Universidad Nacional de Educación a Distancia

Escuela Técnica Superior de Ingeniería Informática

Proyecto de fin de Grado en Ingeniería Informática

Algoritmo de soporte a la decisión para bombas de insulina en pacientes con diabetes tipo 1

Paloma Llorente La Orden

Dirigido por: Alfonso Urquía Moraleda

Codirigido por: Carla Martín Villalba

Curso 2024/2025, convocatoria junio

Algoritmo de soporte a la decisión para bombas de insulina en pacientes con diabetes tipo 1

Proyecto de fin de Grado en Ingeniería Informática

de modalidad específica

Realizado por: Paloma Llorente La Orden

Dirigido por: Alfonso Urquía Moraleda

Codirigido por: Carla Martín Villalba

Fecha de lectura y defensa: junio 2025

Agradecimientos

En septiembre de 2019 me diagnosticaron diabetes tipo 1. Un año después terminé el grado y máster en Ingeniería Aeronáutica y empecé mi vida laboral desarrollando software para flight dynamics, algo así como mecánica orbital para satélites. Cuando pensé que mi vida ya no podía complicarse más, decidí comenzar, como hobby, la carrera de Ingeniería Informática.

Desde entonces he hecho auténticos malabares para compaginar mi tiempo entre los míos, los estudios y una carrera profesional que no ha parado de moverse —con cambios de trabajo y vida en el extranjero incluidos. Al empezar a tratarme la diabetes con bomba de insulina, me surgieron un montón de ideas para mejorar el software que la gestiona. Gracias a Alfonso y Carla, he tenido la oportunidad de desarrollar esas ideas en este proyecto.

Durante más de un año he dedicado gran parte de mi tiempo a dos cosas que jamás imaginé que convivirían: operar satélites geoestacionarios y diseñar un sistema de control para una bomba de insulina. De nuevo, gracias Alfonso y Carla por confiar en que esto valía la pena.

También quiero agradecer a mi endocrino por enseñarme tanto, por compartir su experiencia y por contagiarme esa ambición de entender y dominar esta enfermedad.

Gracias a mis amigos —tanto los de Soria como los de Madrid— por aguantar mis chapas, mis ausencias y mis quejas constantes por no tener tiempo. A mis padres, porque aunque estén cansados de que siempre esté estudiando, sé que están orgullosos. Y a mis hermanos, por estar siempre ahí, cuidándome sin condiciones.

Gracias Paula y Ariel, por formar parte de mi vida. Os quiero.

Resumen

Este Proyecto de Fin de Grado se enmarca en el ámbito de la Ingeniería Informática, con un enfoque multidisciplinar que combina el desarrollo de sistemas dinámicos, teoría del control, modelización matemática, programación de aplicaciones software y despliegue embebido. El objetivo principal es desarrollar un sistema de soporte a la decisión para el control automático de insulina en personas con diabetes tipo 1, integrando un modelo fisiológico de la enfermedad con un algoritmo de administración adaptativa de insulina.

Para ello, se ha diseñado e implementado un modelo computacional de la evolución de la glucosa en sangre, teniendo en cuenta factores fisiológicos y externos. Este modelo ha sido conectado con un algoritmo de control híbrido en bucle semi-cerrado: el usuario introduce manualmente los hidratos de carbono ingeridos, mientras que el sistema ajusta automáticamente la insulina basal mediante un controlador PID adaptativo. El sistema también incluye funcionalidades como predicción de glucosa, administración de bolos de corrección, manejo de unidades grasa-proteína (UGP) y tratamiento de eventos externos como ejercicio, alcohol, enfermedad o estrés.

El desarrollo del sistema se ha materializado en una herramienta propia denominada DiabetesTool, implementada en Python. Esta herramienta integra un modelo fisiológico de diabetes tipo 1, permite el acceso y procesamiento de datos reales descargados desde la nube CareLink™ de Medtronic, y ofrece funcionalidades de simulación, estimación y validación para analizar el comportamiento tanto del modelo como del algoritmo de control automático de insulina.

Finalmente, se ha implementado una interfaz gráfica interactiva sobre una Raspberry Pi con pantalla táctil, que permite ejecutar el sistema en tiempo real y simula el comportamiento de una bomba de insulina portátil. Esta integración hardware-software proporciona una plataforma funcional y extensible, orientada al desarrollo de sistemas de páncreas artificial de código abierto, más inteligentes, personalizables y seguros.

Abstract

This Bachelor's Thesis is framed within the field of Computer Engineering, with a multidisciplinary approach that combines dynamic systems development, control theory, mathematical modeling, software programming, and embedded system deployment. The main objective is to develop a decision support system for the automatic control of insulin in people with type 1 diabetes, integrating a physiological model of the disease with an adaptive insulin delivery algorithm.

To achieve this, a computational model of blood glucose dynamics has been designed and implemented, taking into account both physiological and external factors. This model is integrated with a hybrid semi-closed-loop control algorithm: the user manually inputs carbohydrate intake, while the system automatically adjusts basal insulin using an adaptive PID controller. The system also includes functionalities such as glucose prediction, correction boluses, support for fat-protein units (FPU), and event handling for physical activity, alcohol intake, illness, or stress.

The system has been implemented as a custom tool named DiabetesTool, developed in Python. This tool integrates a physiological model of type 1 diabetes, allows access to and processing of real data downloaded from Medtronic's CareLink™ cloud, and provides simulation, parameter estimation, and validation functionalities to analyze the behavior of both the model and the automatic insulin control algorithm.

Finally, an interactive graphical user interface (GUI) has been deployed on a Raspberry Pi with a touchscreen, enabling real-time execution and simulating the behavior of a portable insulin pump. This hardware-software integration provides a functional and extensible platform aimed at supporting the development of open-source artificial pancreas systems that are more intelligent, customizable, and safe.

From a control engineering perspective, the manipulated variable in this system is insulin, while the monitored variable is blood glucose. The controller continuously adjusts insulin delivery based on the error relative to a target value, aiming to keep glucose levels within the desired range.

Palabras clave

Diabetes tipo 1; Bomba de insulina; Controlador PID; Algoritmo de control; Modelo fisiológico; Simulación; Insulina basal; Bolos de insulina; Raspberry Pi; CareLink.

Keywords

Type 1 Diabetes; Insulin Pump; PID Controller; Control Algorithm; Physiological Model; Simulation; Basal Insulin; Insulin Bolus; Raspberry Pi; CareLink.

Índice

Índice de tablas XIX

Índice de figuras XXX

Índice de pseudocódigos XXXI

1. Introducción, objetivos y estructura **1**

1.1. Introducción 1

1.2. Objetivos 2

1.2.1. Tareas a realizar 3

1.2.2. Resultados esperados 5

1.3. Estructura 5

2. Estado de la cuestión **7**

2.1. Introducción 7

2.2. La diabetes tipo 1 8

2.2.1. Impacto de factores adicionales 10

2.2.2. Limitaciones de las bombas de insulina 13

2.2.3. Posibles soluciones: input manual del usuario 14

2.3. Modelos y simulación de diabetes 16

2.3.1. UVA/PADOVA Type 1 Diabetes Simulator (T1DMS) 16

2.3.2. Cambridge Model 17

2.4. Sistemas de control de diabetes y suministro de insulina 17

2.4.1. Sistemas de bucle cerrado de código abierto 17

2.4.2. Algoritmos comerciales de bucle cerrado 18

2.4.3. Sistemas de bucle abierto (OpenLoop) 18

2.4.4. Variantes al control PID 19

2.5. Conclusiones 21

3. Desarrollo de la herramienta DiabetesTool 23

3.1. Introducción 23

3.2. Análisis del sistema DiabetesTool 23

3.2.1. Requisitos funcionales 24

3.2.2. Requisitos no funcionales 24

3.3. Diseño de la arquitectura y módulos principales 24

3.4. Implementación del modelo y motor de simulación 25

3.4.1. Parámetros y variables de estado 25

3.4.2. Modelización de los eventos 30

3.4.3. Solver y simulación del modelo 33

3.4.4. Pseudocódigo del algoritmo 34

3.5. Visualización de resultados, ejemplos de uso y validación 36

3.5.1. Ingesta de alimentos con bolo de insulina 37

3.5.2. Efecto de los eventos 46

3.5.3. Cambio en los parámetros del paciente 54

3.6. Conclusiones 61

4. Algoritmo de la bomba de insulina 63

4.1. Introducción 63

4.2. Lectura de glucosa 64

4.2.1. La variable tiempo 64

4.2.2. Los datos de glucosa 65

4.3. Entradas de usuario 66

4.3.1. Parámetros 66

4.3.2. Comidas 68

4.3.3. Eventos 69

4.4. Insulina suministrada 76
4.4.1. Insulina basal 76
4.4.2. Bolo de corrección 84
4.4.3. Bolos por comida 87
4.5. Arquitectura del algoritmo 89
4.6. Ejemplos de uso y validación 92
4.6.1. Simulación sin eventos 92
4.6.2. Simulación con eventos 112
4.7. Conclusiones 130

5. Integración del algoritmo en dispositivo **133**

5.1. Introducción 133
5.2. Diagrama de flujo de datos 133
5.3. Raspberry Pi como bomba de insulina 135
5.3.1. Preparación del entorno de desarrollo 135
5.3.2. Medios necesarios 136
5.3.3. Compartión de datos 136
5.4. Arquitectura del programa 137
5.4.1. Pseudocódigo del bucle principal de actualización 141
5.5. Interfaz gráfica 142
5.5.1. Gráfica 143
5.5.2. Comidas 143
5.5.3. Eventos 144
5.5.4. Configuración 147
5.6. Conclusiones 149

6. Obtención de datos reales **151**

6.1. Introducción 151

6.2. API de Carelink . . . 152
6.2.1. Arquitectura del sistema . . . 152
6.2.2. Requisitos del sistema . . . 152
6.3. Datos descargados . . . 152
6.3.1. Conexión con CareLink . . . 153
6.3.2. Estructura y contenido del archivo JSON . . . 153
6.3.3. Datos utilizados por el algoritmo . . . 155
6.4. Diagrama de flujo de datos . . . 156
6.5. Estimador de parámetros . . . 156
6.5.1. Parámetros ajustables . . . 157
6.5.2. Algoritmo de estimación basado en RMS . . . 157
6.5.3. Resultados del estimador . . . 157
6.6. Visualización . . . 158
6.6.1. Descarga de datos . . . 158
6.6.2. Estimador de parámetros . . . 158
6.6.3. Comparador de modelo y datos reales . . . 159
6.6.4. Simulación dinámica con datos reales . . . 160
6.7. Conclusiones . . . 161

7. Validación **163**

7.1. Introducción . . . 163
7.2. Modelo de diabetes . . . 163
7.2.1. Metodología . . . 163
7.2.2. Escenarios de validación . . . 164
7.3. Algoritmo de la bomba de insulina . . . 175
7.3.1. Metodología . . . 175
7.3.2. Escenarios de validación . . . 177

7.4. Satisfacción de los requisitos del sistema . . . 183
7.5. Conclusiones . . . 184

8. Conclusiones y trabajos futuros **185**
8.1. Introducción . . . 185
8.2. Conclusiones . . . 185
8.3. Trabajos futuros . . . 186

Bibliografía **189**

A. Manual de usuario de DiabetesTool **193**
A.1. Introducción . . . 193
A.2. Requisitos del sistema . . . 193
A.3. Ejecución de la herramienta . . . 194
A.4. Descripción de la interfaz . . . 195
A.4.1. Panel superior: selección de modo . . . 195
A.4.2. Panel izquierdo: parámetros del modo . . . 195
A.4.3. Panel derecho: pestañas de archivos . . . 195
A.4.4. Panel central . . . 197
A.5. Casos de uso . . . 197
A.5.1. Dynamic Simulator . . . 198
A.5.2. Static Simulator . . . 199
A.5.3. Estimator . . . 201
A.5.4. Downloader . . . 202
A.5.5. Comparator . . . 203
A.6. Conclusiones . . . 204

B. Manual de usuario de la bomba de insulina **205**
B.1. Introducción . . . 205

B.2. Requisitos del sistema . . . 205
B.3. Ejecución . . . 206
B.4. Interfaz de usuario . . . 206
B.4.1. Panel izquierdo: estado actual . . . 206
B.4.2. Panel derecho: controles y configuración . . . 207
B.5. Uso general . . . 209
B.6. Conclusiones . . . 210

C. Preparación del dispositivo Raspberry Pi 211
C.1. Introducción . . . 211
C.2. Instalación del sistema operativo . . . 211
C.3. Compartición de datos mediante Samba . . . 212
C.3.1. Instalación de Samba en Raspberry Pi . . . 212
C.3.2. Configuración del archivo `smb.conf` . . . 212
C.3.3. Creación de usuario y contraseña de Samba . . . 213
C.3.4. Reinicio del servicio . . . 213
C.3.5. Acceso desde Windows . . . 213

D. Código fuente de DiabetesTool 217
D.1. Estructura . . . 217
D.2. main.py . . . 218
D.3. carelink_carepartner_api_login.py . . . 218
D.4. app . . . 224
D.4.1. app.py . . . 224
D.4.2. input_section.py . . . 226
D.4.3. logging_handler.py . . . 239
D.4.4. run_programs.py . . . 240
D.5. carelink . . . 243

D.5.1. carelink_client2.py . . . 243
D.5.2. data.py . . . 251
D.5.3. json_carelink.py . . . 254
D.5.4. pump.py . . . 258
D.6. inputs . . . 263
D.6.1. input_auxi.py . . . 263
D.6.2. json_data.py . . . 264
D.7. model . . . 267
D.7.1. ambridge_model.py . . . 267
D.7.2. events.py . . . 272
D.7.3. glucose.py . . . 276
D.7.4. nsulin.py . . . 278
D.7.5. meal.py . . . 280
D.8. programs . . . 281
D.8.1. comparator_main.py . . . 281
D.8.2. download_main.py . . . 284
D.8.3. estimator_main.py . . . 285
D.8.4. pump_main.py . . . 289
D.8.5. simulator_main.py . . . 291
D.9. simulator . . . 295
D.9.1. auxi.py . . . 295
D.9.2. simulator.py . . . 296
D.9.3. solver.py . . . 302
D.10.util . . . 304
D.10.1. auxi.py . . . 304
D.10.2. glucose_plot.py . . . 307
D.10.3. residuals.py . . . 316

E. Código fuente del algoritmo de la bomba de insulina **321**

E.1. Estructura . . . 321

E.2. main.py . . . 322

E.3. app . . . 322

E.3.1. app.py . . . 322

E.3.2. meal_screen.py . . . 329

E.3.3. options_screen.py . . . 333

E.3.4. screen.py . . . 339

E.4. pump . . . 342

E.4.1. basal.py . . . 342

E.4.2. correction.py . . . 349

E.4.3. data.py . . . 352

E.4.4. events.py . . . 354

E.5. util . . . 360

E.5.1. plot.py . . . 360

E.5.2. time.py . . . 361

E.5.3. utils.py . . . 365

Índice de tablas

3.1. Resumen de parámetros del modelo 26

3.2. Variables de estado del modelo 27

Índice de figuras

1.1. Relación entre objetivos, tareas y capítulos del proyecto. 4

2.1. Metabolismo del carbohidrato en una persona sin diabetes [1]. 8

2.2. Bomba de insulina Minimed 780. 9

2.3. Vías biológicas por las que las partículas PM2.5 pueden promover la enfermedad cardiometabólica, según [2]. 13

2.4. Interfaz de usuario del simulador de LoopInsighT1. 16

2.5. Modelo de distribución de insulina de acuerdo al algoritmo PID, según [3]. . . . 20

3.1. Diagrama del modelo separado en compartimentos [1]. 30

3.2. Visualización de Inputs en Static Simulator. 37

3.3. Visualización de Events en Static Simulator. 38

3.4. Visualización de Parameters en Static Simulator. 39

3.5. Evolución de glucosa Ejemplo Inicial. 39

3.6. Evolución de variables de estado Ejemplo Inicial. 40

3.7. Retardo en acción de la insulina de 20 minutos. 40

3.8. Retardo en acción de la comida de 10 minutos. 41

3.9. Retardo en acción de la insulina de 20 minutos y en la comida de 10 minutos. . 41

3.10. Hipoglucemia por más insulina. 42

3.11. Hipoglucemia por menos HC. 42

3.12. Hipoglucemia por demasiado tiempo de espera. 42

3.13. Hipoglucemia por demasiada basal. 43

3.14. Hiperglucemia por menos insulina. 44

3.15. Hiperglucemia por más HC. 44

3.16. Hiperglucemia por comida antes que bolo. 44

3.17. Hiperglucemia por insuficiente basal. 45

3.18. Comida sin UGPs. 45
3.19. Comida con 2 UGPs. 46
3.20. Comida con 5 UGPs. 46
3.21. Comida con 5 UGPs más 1 bolo de corrección. 47
3.22. Comida con 5 UGPs más 2 bolo de corrección. 47
3.23. Variables de estado para comida con 5 UGPs más 1 bolo de corrección. 48
3.24. Evolución de la glucosa sin eventos. 48
3.25. Glucosa con ejercicio de intensidad baja. 49
3.26. Parámetros fisiológicos con ejercicio de intensidad baja. 49
3.27. Glucosa con ejercicio de intensidad alta. 50
3.28. Parámetros fisiológicos con ejercicio de intensidad alta. 50
3.29. Glucosa con ejercicio de intensidad alta e ingesta posterior de hidratos. 51
3.30. Glucosa con ingesta de alcohol. 51
3.31. Parámetros fisiológicos con ingesta de alcohol. 52
3.32. Glucosa con ingesta de alcohol y alguna bebida azúcarada. 52
3.33. Glucosa tras bolo de insulina con pleno efecto. 53
3.34. Glucosa tras bolo de insulina con menor efecto. 53
3.35. Glucosa tras bolo de insulina y estado relajado. 53
3.36. Parámetros fisiológicos tras bolo de insulina y estado relajado. 54
3.37. Glucosa tras bolo de insulina y estado de estrés. 54
3.38. Parámetros fisiológicos tras bolo de insulina y estado de estrés. 55
3.39. Glucosa tras bolo de insulina e infección. 55
3.40. Parámetros fisiológicos tras bolo de insulina e infección. 56
3.41. Glucosa tras bolo de insulina y fiebre. 56
3.42. Parámetros fisiológicos tras bolo de insulina y fiebre. 57
3.43. Glucosa tras bolo de insulina y deshidratación. 57
3.44. Parámetros fisiológicos tras bolo de insulina y deshidratación. 58

3.45. Glucosa con variación en el parámetro SIT. 58

3.46. Variables de estado con variación en el parámetro SIT. 59

3.47. Variación en el parámetro SIT. 59

3.48. Glucosa con variación en el parámetro SID 59

3.49. Glucosa con variación en el parámetro SIE. 60

3.50. Glucosa con variación en el parámetro EGP0. 60

3.51. Variables de estado con variación en el parámetro EGP0. 60

3.52. Variación en el parámetro EGP0. 61

4.1. Detalle de pump. 90

4.2. Detalle de correction. 90

4.3. Detalle de basal. 91

4.4. Glucosa, insulina y carbohidratos para el caso de comida sin UGPs I. 93

4.5. Detalle de la bomba de insulina en el momento de la Figura 4.4. 94

4.6. Glucosa, insulina y carbohidratos para el caso de comida sin UGPs II. 94

4.7. Detalle de la bomba de insulina en el momento de la Figura 4.6. 94

4.8. Evolución completa de glucosa, insulina y carbohidratos para el caso de comida sin UGPs. 95

4.9. Evolución completa de variables de estado para el caso de comida sin UGPs. . . 95

4.10. Pantalla de comida para carbohidratos con UGPs. 96

4.11. Glucosa, insulina y carbohidratos para el caso de comida con UGPs I. 96

4.12. Detalle de la bomba de insulina en el momento de la Figura 4.11. 96

4.13. Glucosa, insulina y carbohidratos para el caso de comida con UGPs II. 97

4.14. Detalle de la bomba de insulina en el momento de la Figura 4.13. 97

4.15. Glucosa, insulina y carbohidratos para el caso de comida con UGPs III. 97

4.16. Glucosa, insulina y carbohidratos para el caso de comida con UGPs IV. 98

4.17. Detalle de la bomba de insulina en el momento de la Figura 4.16. 98

4.18. Evolución completa de glucosa, insulina y carbohidratos para el caso de comida con UGPs. 98

4.19. Evolución completa de variables de estado para el caso de comida con UGPs. . . 99

4.20. Glucosa, insulina y carbohidratos para el caso de correcciones automáticas I. . . 100

4.21. Detalle de la bomba de insulina en el momento de la Figura 4.20. 100

4.22. Glucosa, insulina y carbohidratos para el caso de correcciones automáticas II. . . 101

4.23. Detalle de la bomba de insulina en el momento de la Figura 4.22. 101

4.24. Evolución completa de glucosa, insulina y carbohidratos para el caso de correcciones automáticas. 101

4.25. Evolución completa de variables de estado para el caso de correcciones automáticas. 102

4.26. Evolución completa de glucosa e insulina para el caso de basal por control PID. 105

4.27. Evolución completa de variables de estado para el caso de basal por control PID. 105

4.28. Pantalla targets de glucosa. 106

4.29. Pantalla ratios. 106

4.30. Pantalla sensibilidad a la insulina. 106

4.31. Detalle de la bomba de insulina en el momento previo a administrar un bolo por comida sin UGPs. 107

4.32. Glucosa, insulina y carbohidratos para el caso de diferentes parámetros I. 107

4.33. Detalle de la bomba de insulina en el momento de la Figura 4.32. 107

4.34. Detalle de la bomba de insulina para administrar un bolo por comida antes de cambiar su ratio. El ratio es 8.5 g/U. No se administra este bolo. 108

4.35. Detalle de la bomba de insulina para administrar un bolo por comida después de cambiar su ratio. El ratio es 10 g/U. Se administra este bolo. 108

4.36. Glucosa, insulina y carbohidratos para el caso de diferentes parámetros II. . . . 109

4.37. Detalle de la bomba de insulina en el momento de la Figura 4.36. 109

4.38. Glucosa, insulina y carbohidratos para el caso de diferentes parámetros III. . . . 109

4.39. Detalle de la bomba de insulina en el momento de la Figura 4.38. 109

4.40. Evolución completa de glucosa, insulina y carbohidratos para el caso de diferentes objetivo, sensibilidad y ratio. 110

4.41. Evolución completa de variables de estado para el caso de diferentes objetivo, sensibilidad y ratio. 111

4.42. Glucosa, insulina y carbohidratos para el caso de antes pero inyectando la insulina 40 min antes I. 111

4.43. Detalle de la bomba de insulina en el momento de la Figura 4.42. 111

4.44. Glucosa, insulina y carbohidratos para el caso de antes pero inyectando la insulina 40 min antes II. 112

4.45. Glucosa e insulina para un caso sin eventos. 112

4.46. Pantalla de ingestión de alcohol. 113

4.47. Bolo de insulina por 15 g iniciales para 3 unidades de alcohol. 113

4.48. Glucosa, insulina y carbohidratos para el caso de 3 unidades de alcohol I. 113

4.49. Detalle de la bomba de insulina en el momento de la Figura 4.48. 114

4.50. Glucosa, insulina y carbohidratos para el caso de 3 unidades de alcohol II. . . . 114

4.51. Bolo de insulina por 15 g finales para 3 unidades de alcohol. 114

4.52. Glucosa, insulina y carbohidratos para el caso de 3 unidades de alcohol III. . . . 115

4.53. Detalle de la bomba de insulina en el momento de la Figura 4.52. 115

4.54. Evolución completa de glucosa, insulina y carbohidratos para el caso de 3 unidades de alcohol. 115

4.55. Evolución completa de variables de estado para el caso de 3 unidades de alcohol. 116

4.56. Evolución completa de parámetros fisiológicos para el caso de 3 unidades de alcohol.116

4.57. Bolo de insulina por 15 g iniciales para 6 unidades de alcohol. 117

4.58. Bolo de insulina por 15 g finales para 6 unidades de alcohol. 117

4.59. Evolución completa de glucosa, insulina y carbohidratos para el caso de 6 unidades de alcohol. 117

4.60. Evolución completa de variables de estado para el caso de 6 unidades de alcohol. 118

4.61. Evolución completa de EGP0 para el caso de 6 unidades de alcohol. 118

4.62. Pantalla de ejercicio físico. 118

4.63. Evolución completa de glucosa e insulina para el caso de ejercicio físico. 119

4.64. Evolución completa de variables de estado para el caso de ejercicio físico. 121

4.65. Evolución completa de parámetros fisiológicos para el caso de ejercicio físico. . . 121

4.66. Pantalla de enfermedad. 122

4.67. Evolución completa de glucosa e insulina para el caso de infección. 122

4.68. Evolución completa de variables de estado para el caso de infección. 123

4.69. Evolución completa de parámetros fisiológicos para el caso de infección. 123

4.70. Pantalla de estado anímico (estrés). 124

4.71. Evolución completa de glucosa e insulina para el caso de estrés. 124

4.72. Evolución completa de variables de estado para el caso de estrés. 125

4.73. Evolución completa de parámetros fisiológicos para el caso de estrés. 126

4.74. Pantalla de estado anímico (relajación). 126

4.75. Evolución completa de glucosa e insulina para el caso de relajación. 127

4.76. Evolución completa de variables de estado para el caso de relajación. 128

4.77. Evolución completa de parámetros fisiológicos para el caso de relajación. 128

4.78. Pantalla de días de máximo efecto de insulina. 129

4.79. Pantalla de cambio de reservorio. 129

4.80. Evolución completa de glucosa, insulina y carbohidratos para el caso de insulina que ha perdido efecto. 129

4.81. Evolución completa de glucosa, insulina y carbohidratos para el caso de insulina con pleno efecto. 130

5.1. Diagrama de flujo de datos. 134

5.2. Raspberry Pi como bomba de insulina. 135

5.3. Raspberry Pi con pantalla táctil, bomba de Medtronic a la izquierda. 136

5.4. Contenido de la carpeta compartida. 137

5.5. Arquitectura del programa. 138

5.6. Detalle de app. 139

5.7. Detalle de util. 141

5.8. Ejecución del algoritmo de la bomba de insulina en la Raspberry Pi. 143

5.9. Trazas de la terminal de la DiabetesTool referenciando a los archivos de la carpeta compartida. . . . 143

5.10. Pantalla de bomba sin selección. . . . 143

5.11. Pantalla de bomba en la Raspberry Pi. . . . 144

5.12. Pantalla de bomba, selección de gráfica. . . . 144

5.13. Pantalla de bomba, selección de comidas. . . . 144

5.14. Pantalla de bomba, selección de eventos. . . . 144

5.15. Pantalla de ejercicio, lista vacía. . . . 145

5.16. Pantalla de ejercicio, añadir ejercicio y modificar valor. . . . 145

5.17. Pantalla de ejercicio, añadir nuevo ejercicio. . . . 145

5.18. Pantalla de ejercicio, borrar ejercicio. . . . 145

5.19. Pantalla de alcohol. . . . 146

5.20. Pantalla de estado anímico. . . . 146

5.21. Pantalla de estado anímico, selección de tipo. . . . 146

5.22. Pantalla de enfermedad, selección de tipo. . . . 146

5.23. Pantalla de cambio de reservorio. . . . 147

5.24. Pantalla de bomba, selección de configuración. . . . 147

5.25. Pantalla de BG target. . . . 147

5.26. Pantalla de BG target, modificar valor. . . . 148

5.27. Pantalla de ratio de HC. . . . 148

5.28. Pantalla de factor de sensibilidad a la insulina. . . . 148

5.29. Pantalla de efecto del reservorio de insulina. . . . 148

5.30. Pantalla de basal máxima. . . . 149

6.1. Arquitectura de comunicación de datos de CareLink. . . . 152

6.2. Diagrama de flujo de datos con datos reales. . . . 156

6.3. Trazas de la terminal durante la descarga de datos y visualización de las últimas 24h de glucosa. . . . 158

6.4. Estimación de parámetros: residuos. 159

6.5. Comparación entre la curva de glucosa simulada y la real tras la estimación. . . 159

6.6. Comparación entre la curva del modelo y los datos reales de glucosa. 160

6.7. Simulación dinámica basada en datos reales de la bomba de insulina. 161

7.1. Parámetros iniciales para la comida estándar. 165

7.2. Inputs comparador inicial para la comida estándar. 165

7.3. Plots de la comparación inicial para la comida estándar. 165

7.4. Modificación tiempo de comida para la comida estándar. 166

7.5. Inputs estimador para la comida estándar: m_delay. 166

7.6. Estimación m_delay para comida estándar: evolución de glucosa. 167

7.7. Estimación m_delay para comida estándar: evolución de residuos. 167

7.8. Parámetros para la comida estándar: modificación de m_delay. 167

7.9. Estimación SIT para comida estándar: evolución de glucosa. 168

7.10. Estimación SIT para comida estándar: evolución de residuos. 168

7.11. Parámetros para la comida estándar: modificación de SIT. 169

7.12. Plots de la comparación final para la comida estándar. 169

7.13. Plots de la comparación para la comida alta UGP. 171

7.14. Modificación inputs para la comida alta UGP. 171

7.15. Resultados estimación SIT para la comida alta UGP. 171

7.16. Evolución de glucosa para la comida alta UGP. 172

7.17. Residuos para la comida alta UGP. 172

7.18. Evento asociado al ejercicio físico. 173

7.19. Plots de la comparación para el ejercicio físico. 173

7.20. Estimación de SIT para el ejercicio físico: evolución de glucosa. 174

7.21. Estimación de SIT para el ejercicio físico: evolución de residuos. 174

7.22. Estimación de EGP0 para el ejercicio físico: terminal. 175

7.23. Ratios de CH en función de la hora del día. 176

7.24. Target de glucosa en función de la hora del día. 176

7.25. Factor de sensibilidad de la insulina en función de la hora del día. 176

7.26. Ratios de CH en función de la hora del día en la bomba real. 176

7.27. Target de glucosa en función de la hora del día en la bomba real. 177

7.28. Factor de sensibilidad de la insulina en función de la hora del día en la bomba real. 177

7.29. Comidas del archivo de inputs de los datos reales para el escenario de comida estándar. 178

7.30. Simulador al inicio de la simulación para el escenario de comida estándar. 178

7.31. Primera comida para el escenario de comida estándar. 179

7.32. Segunda comida para el escenario de comida estándar. 179

7.33. Comparación de insulinas para el escenario de comida estándar. 179

7.34. Comparación de insulinas para el escenario de comida alta UGP. 181

7.35. Explicación decisiones del algoritmo para el escenario de comida alta UGP. . . . 181

7.36. Panel de ejercicio en la aplicación de la bomba de insulina. 182

7.37. Comparación de insulinas para el escenario de ejercicio físico. 182

A.1. Panel superior de DiabetesTool con botones de selección de modo. 195

A.2. Panel izquierdo de DiabetesTool con campos de entrada para el modo seleccionado.196

A.3. Panel derecho de DiabetesTool con pestañas de archivos. 197

A.4. Panel central de DiabetesTool con la consola y gráficos. 198

B.1. Panel izquierdo de la interfaz de usuario de la bomba de insulina simulada. . . . 207

B.2. Panel derecho de la interfaz de usuario de la bomba de insulina simulada. 208

B.3. Gráfico de evolución de glucosa en la bomba. 208

B.4. Panel de comidas en la bomba de insulina. 208

B.5. Panel de eventos en la bomba de insulina. 209

B.6. Panel de configuración de la bomba de insulina. 210

C.1. comando ip addr: direacción 192.168.0.17 de la red wlan0. 214

C.2. Credenciales para conectar a la carpeta compartida. 214

C.3. Carpeta home de la Raspberry Pi. 214

Índice de pseudocódigos

3.1. Solver adaptativo RK1-RK2 (Heun-Euler). 34
3.2. Solver adaptativo RK1-RK2 (Heun-Euler) implementado. 35
3.3. Funciones auxiliares para el solver adaptativo RK1-RK2 (Heun-Euler) implementado. 36
4.1. Ajuste de parámetros por eventos (I). 74
4.2. Ajuste de parámetros por eventos (II). 75
4.3. Decisión de aplicación de eventos. 75
4.4. Cálculo del factor de eficacia. 76
4.5. Cálculo de la insulina basal mediante PID adaptativo (I). 80
4.6. Cálculo de la insulina basal mediante PID adaptativo (II). 81
4.7. Cálculo de la derivada del error y decisión de adaptación. 81
4.8. Adaptación de parámetros PID. 82
4.9. Tiempo en rango. 83
4.10. Análisis de estabilidad. 83
4.11. Ajuste de K_P. 83
4.12. Ajuste de K_I. 84
4.13. Ajuste de K_D. 84
4.14. Cálculo de la tasa basal promedio. 84
4.15. Cálculo del bolo corrector. 86
4.16. Cálculo del bolo por ingesta de comida. 88
4.17. Ajuste del bolo por corrección de glucosa precomida. 89
5.1. Bucle principal de actualización (I). 141
5.2. Bucle principal de actualización (II). 142

Capítulo 1

Introducción, objetivos y estructura

1.1. Introducción

El control de la diabetes tipo 1 representa uno de los mayores desafíos en el ámbito de la medicina personalizada, debido a la gran variabilidad inter e intraindividual que presentan los niveles de glucosa en sangre y a la necesidad constante de ajustar la dosis de insulina. La evolución tecnológica de los dispositivos de medición continua de glucosa (CGM) y las bombas de insulina ha permitido importantes avances en los sistemas de administración automatizada de insulina (AID, por sus siglas en inglés), pero todavía existen numerosas limitaciones.

Entre ellas se encuentran la escasa capacidad de estos sistemas para adaptarse a factores externos (como ejercicio, emociones, enfermedades o ingesta de alcohol), la falta de personalización real basada en los datos del propio paciente, y una limitada transparencia sobre el funcionamiento interno de los algoritmos comerciales, los cuales no están disponibles como código abierto.

Desde una perspectiva personal, la motivación principal de este trabajo surge tras el diagnóstico de diabetes tipo 1 y la experiencia directa con el uso de una bomba de insulina. Esta vivencia reveló una serie de carencias en la lógica de decisión de estos dispositivos, así como un gran potencial de mejora si se aplicaban conocimientos de ingeniería, control automático y programación. Este proyecto es, por tanto, una combinación de interés personal, experiencia práctica y conocimientos técnicos, con el objetivo de construir una herramienta útil y ampliable que pueda servir de base para futuros desarrollos en el campo de los sistemas de páncreas artificial.

El presente trabajo se enmarca dentro del ámbito de la ingeniería informática, con una orientación multidisciplinar que combina conocimientos de sistemas dinámicos, teoría del control, desarrollo software y electrónica. El objetivo principal es diseñar e implementar un sistema completo capaz de modelar el comportamiento de la glucosa en pacientes con diabetes tipo 1 y de tomar decisiones automáticas para administrar insulina de forma precisa, adaptativa y personalizada.

Para comprender correctamente los objetivos propuestos en la Sección 1, es importante tener claros algunos conceptos clave. En primer lugar, un *modelo de diabetes* es una representación matemática que describe cómo evoluciona la glucosa en sangre en función de variables como la insulina administrada, la ingesta de alimentos o el ejercicio físico. Dicho modelo se implementa resolviendo un conjunto de ecuaciones diferenciales.

Por otro lado, una *bomba de insulina* es un dispositivo que administra insulina de manera continua (insulina basal) y permite administrar dosis adicionales (bolos) en momentos específicos, como antes de las comidas. Los sistemas más avanzados integran sensores de glucosa y algo-

ritmos de control que automatizan parcialmente estas decisiones, en lo que se conoce como *sistemas híbridos en bucle cerrado*.

En este contexto, un *controlador PID* (Proporcional-Integral-Derivativo) es una técnica ampliamente utilizada en sistemas de control industrial que se ha aplicado con éxito a la administración automática de insulina. En este trabajo, el controlador PID se adapta dinámicamente a la evolución de la glucosa y a las condiciones del paciente.

1.2. Objetivos

El objetivo principal de este Proyecto de Fin de Grado es desarrollar un sistema de soporte a la decisión para el control automático de insulina en personas con diabetes tipo 1, integrando un modelo fisiológico de la enfermedad con un algoritmo de administración automática de insulina. El proyecto combinará elementos de modelización matemática, simulación, control automático, desarrollo software y despliegue embebido.

Este sistema estará concebido como una herramienta de código abierto que, además de simular el comportamiento de un paciente diabético, permitirá evaluar distintos enfoques de control automatizado y visualizar su impacto en los niveles de glucosa. El objetivo final será facilitar el desarrollo de bombas de insulina más inteligentes, personalizables y seguras.

Los objetivos específicos planteados son los siguientes:

1. Crear un modelo de diabetes tipo 1 que incluya:
   - Ingesta de hidratos de carbono.
   - Dosis de insulina inyectada (bolos y basal).
   - Otros inputs relevantes como el estado anímico, enfermedad, consumo de alcohol y ejercicio físico.
2. Desarrollar un algoritmo de bucle cerrado que genere bolos de insulina automáticos basados en el modelo creado. Este algoritmo se ajusta en tiempo real según la retroalimentación de los niveles de glucosa, mediante un controlador PID adaptativo, asegurando una administración precisa y continua de insulina.
3. Obtener y utilizar datos reales de glucosa mediante una API conectada a la nube de Medtronic, lo que permite evaluar y validar el comportamiento del sistema frente a escenarios reales.
4. Simular el funcionamiento de una bomba de insulina en una Raspberry Pi con pantalla táctil, permitiendo la interacción directa del usuario con la interfaz del sistema y la visualización en tiempo real de los datos.
5. Comparar el rendimiento del modelo y del algoritmo propuesto con el de una bomba de insulina real, tomando como referencia el dispositivo MiniMed 780G.

El presente proyecto propone una herramienta que, desde un enfoque modular y abierto, permitirá tanto simular como visualizar el comportamiento de la glucosa un paciente. Por otro lado, también se desarrollará un algoritmo de una bomba de insulina en tiempo real, con una interfaz gráfica intuitiva para su despliegue en una Raspberry Pi, que simulará el funcionamiento de una bomba de insulina.

La validación del sistema se realizará tanto con datos sintéticos como con datos reales descargados desde la nube de Medtronic, permitiendo así contrastar el funcionamiento del modelo y del algoritmo frente a situaciones reales.

En la Figura 1.1 se muestra la relación entre los objetivos, las tareas a realizar y los capítulos del proyecto. Esta figura ilustra cómo cada objetivo se descompone en tareas específicas que se desarrollarán a lo largo de los distintos capítulos de la memoria.

1.2.1. Tareas a realizar

Para cumplir estos objetivos, se llevarán a cabo las siguientes tareas:

- Se desarrollará un modelo de diabetes basado en el Cambridge Model, implementado en Python, con posibilidad de personalizar sus parámetros fisiológicos para adaptarse a distintos pacientes.
- Se diseñará un método numérico numérico adaptativo de tipo Runge-Kutta (RK1–RK2), que permitirá simular la evolución de la glucosa en sangre con precisión y eficiencia.
- Se implementará una herramienta software llamada `DiabetesTool`, que incluirá funcionalidades de simulación estática y dinámica, estimación de parámetros, descarga de datos desde CareLink, comparación de resultados y visualización. En el Apéndice A se incluirá un manual de usuario detallado.
- Se desarrollará un algoritmo de la bomba de insulina con un controlador PID adaptativo, integrando módulos para la administración de insulina basal, bolos prandiales y bolos de corrección, además de mecanismos para considerar factores externos (eventos) como el ejercicio o el estrés.
- Para facilitar su uso en un entorno realista, se creará una interfaz gráfica (GUI) desplegada en una Raspberry Pi con pantalla táctil, que simulará el comportamiento de una bomba de insulina. El Apéndice B contendrá el manual completo de esta interfaz.
- Se descargarán y procesarán datos reales de glucosa mediante la API de CareLink de Medtronic, integrando estos datos en el modelo para realizar validaciones y pruebas comparativas.
- Finalmente, se ejecutarán simulaciones y escenarios de validación, comparando los resultados del modelo y del algoritmo de la bomba con los datos reales disponibles.

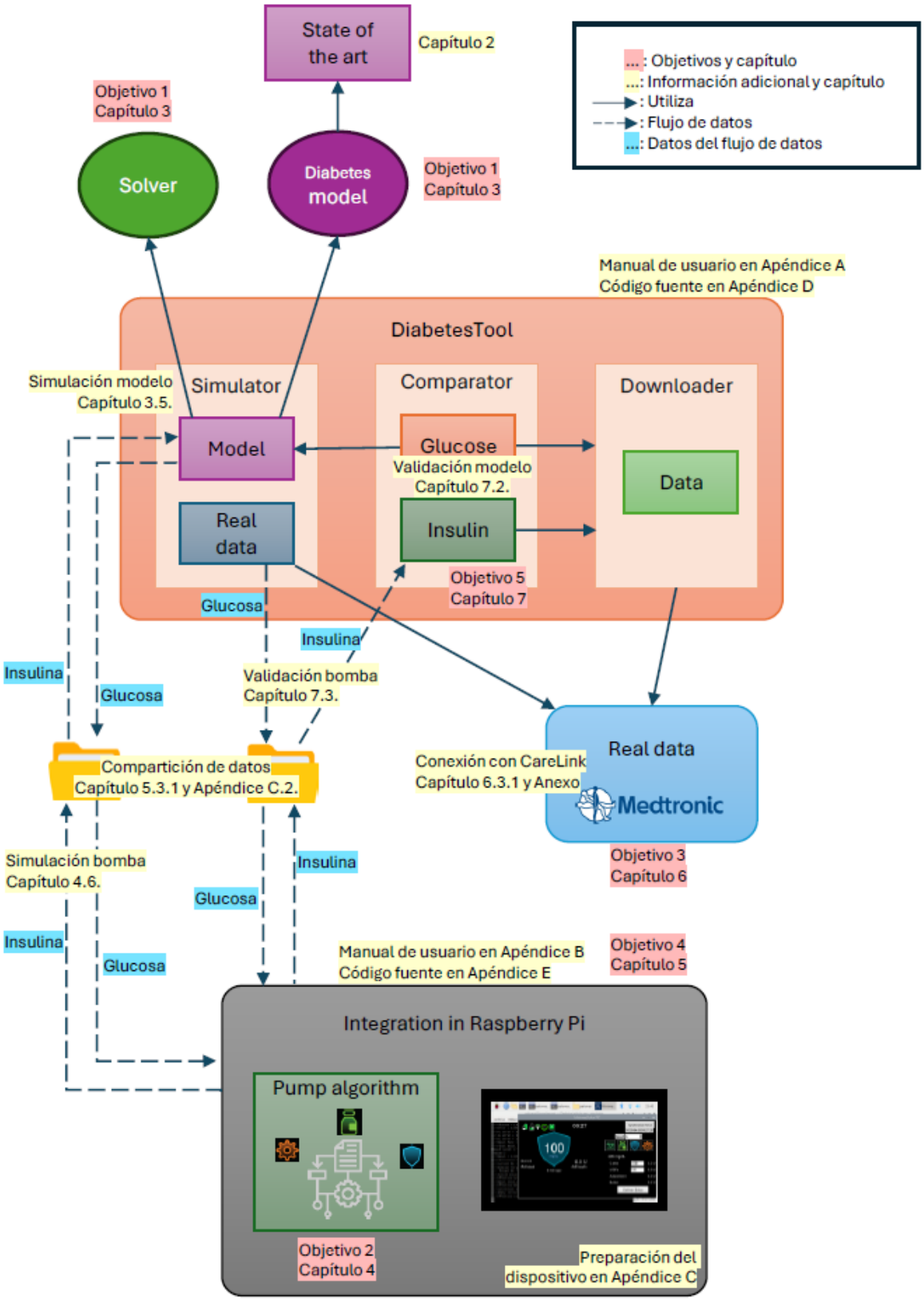


Figura 1.1: Relación entre objetivos, tareas y capítulos del proyecto.

1.2.2. Resultados esperados

Aunque el número de escenarios de validación que se implementarán será limitado, se espera que los resultados muestren un comportamiento prometedor, tanto en la evolución de la glucosa simulada como en las decisiones tomadas por el algoritmo de la bomba. Se verificará que el sistema será capaz de ajustar la administración de insulina de forma coherente ante diferentes situaciones (comidas, ejercicio, alcohol, estrés), reproduciendo fenómenos observados en la experiencia clínica real.

Este trabajo sentará así las bases para futuras ampliaciones y mejoras en sistemas de páncreas artificial de código abierto, con un enfoque centrado en la personalización, la flexibilidad y la validación frente a datos reales.

1.3. Estructura

Este documento se estructura en ocho capítulos principales, complementados por varios apéndices.

El **Capítulo** 1 presenta la introducción al problema, los objetivos del proyecto y una visión general de la estructura del documento.

El **Capítulo** 2 describe la enfermedad de la diabetes tipo 1 y los principales factores que afectan al control de la glucosa y las limitaciones actuales de las bombas de insulina. También ofrece una revisión del estado de la cuestión, incluyendo modelos y simuladores de diabetes tipo 1, así como distintos enfoques de control de la insulina (sistemas en bucle cerrado, abierto, híbrido, controladores PID y variantes). Esta sección justifica las decisiones tomadas para el diseño del modelo y del algoritmo del presente trabajo.

En el **Capítulo** 3 se presenta el desarrollo de la herramienta *DiabetesTool*, cuyo objetivo es implementar un modelo de simulación personalizado de la diabetes tipo 1 basado en el Cambridge Model. Se describen los parámetros fisiológicos, las variables de estado y la modelización de eventos como comidas, ejercicio o enfermedades. Además, se detalla el método numérico utilizado, el diseño de la arquitectura software y los módulos desarrollados, así como las funcionalidades de visualización y ejemplos de simulación.

El **Capítulo** 4 está dedicado al algoritmo de la bomba de insulina. Se describen su funcionamiento, las lecturas de glucosa, las entradas de usuario, la generación de insulina basal y bolos de corrección, y su arquitectura. Incluye también un conjunto amplio de pruebas de simulación que permiten analizar el comportamiento del sistema.

El **Capítulo** 5 aborda la integración del algoritmo en un dispositivo hardware. Se describe el uso de una Raspberry Pi con pantalla táctil para simular el entorno de una bomba de insulina portátil. Se explica el flujo de datos, la arquitectura software implementada, el pseudocódigo del bucle principal de control y la interfaz gráfica diseñada.

El **Capítulo** 6 trata sobre la obtención y uso de datos reales. Se documenta el uso de la API de CareLink de Medtronic para descargar datos de glucosa y se detalla cómo se integran en el

sistema para su análisis, estimación de parámetros y validación frente al modelo.

El **Capítulo** 7 está dedicado a la validación. Se establecen distintas metodologías y escenarios, tanto para el modelo de diabetes como para el algoritmo de la bomba de insulina, incluyendo comparaciones con datos reales y análisis de sensibilidad a distintos parámetros.

Por último, el **Capítulo** 8 recoge las conclusiones generales del trabajo y plantea líneas de trabajo futuras para continuar mejorando el sistema.

La memoria incluye también los siguientes apéndices:

- **Apéndice** A: Manual de usuario de `DiabetesTool`, herramienta software desarrollada para ejecutar simulaciones, comparaciones y descargas de datos.
- **Apéndice** B: Manual de usuario de la aplicación que simula la bomba de insulina con interfaz gráfica.
- **Apéndice** C: Guía de instalación y configuración del sistema en una Raspberry Pi.
- **Apéndices** D y E: Código fuente detallado de los distintos módulos desarrollados, tanto para `DiabetesTool` como para la aplicación de la bomba de insulina.

La entrega del proyecto incluye:

- El archivo PDF de esta memoria.
- El código fuente completo de la herramienta `DiabetesTool`.
- El código fuente completo de la aplicación con el algoritmo de la bomba de insulina.

Capítulo 2

Estado de la cuestión

2.1. Introducción

La diabetes tipo 1 es una enfermedad crónica con múltiples factores fisiológicos, ambientales y de comportamiento que dificultan su control automatizado. En el Capítulo 1 se introdujo la motivación general del proyecto, centrada en desarrollar un sistema de control de insulina personalizado. En este capítulo se profundiza en los aspectos médicos, tecnológicos y de ingeniería que conforman el estado actual del conocimiento sobre el tratamiento automatizado de la diabetes tipo 1.

Este capítulo se va a proporcionar una visión general del estado de la cuestión en cuanto a modelado, simulación y control de la diabetes tipo 1. Se realiza una revisión de las principales plataformas y modelos utilizados en la literatura para simular la evolución de la glucosa en pacientes, así como de los enfoques más relevantes en el diseño de algoritmos de control de insulina.

En primer lugar, se abordan las herramientas de simulación disponibles, como LoopInsighT1, el simulador UVA/Padova y el Cambridge Model. Se analizan sus características, aplicaciones y limitaciones.

A continuación, se exploran los distintos enfoques para el diseño de sistemas de control de la glucosa, desde los sistemas en bucle abierto hasta los algoritmos comerciales de bucle cerrado, incluyendo también soluciones de código abierto como OpenAPS. Se dedica una sección especial a los controladores PID y sus variantes más avanzadas, por ser estos los métodos más utilizados en la práctica clínica y en la investigación aplicada.

Por último, se describen los conceptos fundamentales del control híbrido en lazo cerrado, sobre los que se construirá el sistema de control que se desarrollará en los siguientes capítulos.

La estructura de este capítulo es la siguiente:

- En la Sección 2.2, se presenta una introducción a la **diabetes tipo 1**, incluyendo su fisiopatología, tratamiento y factores que afectan los niveles de glucosa en sangre.
- En la Sección 2.3, se presentan los **modelos y herramientas de simulación** utilizados para representar la evolución de la glucosa en pacientes con diabetes tipo 1, incluyendo plataformas de código abierto y simuladores clínicamente validados.
- En la Sección 2.4, se analizan los **sistemas de control de insulina**, incluyendo enfoques comerciales y de código abierto, el papel de los controladores PID y sus variantes, y el concepto de control híbrido en lazo cerrado.

2.2. La diabetes tipo 1

La diabetes tipo 1 es una enfermedad autoinmune crónica en la cual el sistema inmunológico del cuerpo ataca y destruye las células beta del páncreas, que son responsables de producir insulina. Como resultado, las personas con diabetes tipo 1 no pueden producir suficiente insulina, una hormona clave que regula los niveles de glucosa (azúcar) en sangre. Sin insulina, la glucosa no puede entrar en las células para ser utilizada como fuente de energía, lo que lleva a hiperglucemia (niveles altos de azúcar en sangre). Si no se controla, esto puede causar complicaciones graves, como daño a los nervios, problemas renales, y enfermedades cardiovasculares.

La Figura 2.1 muestra el metabolismo de los carbohidratos en el cuerpo humano. Este metabolismo tiene dos funciones principales. Primero, asegura un suministro constante de energía a las células del organismo. En segundo lugar, garantiza que los niveles de glucosa en sangre (BG) se estabilicen después de influencias externas como la ingesta de alimentos, el ayuno o el ejercicio. La insulina y el glucagón son las dos hormonas clave que regulan este sistema, asegurando que los niveles de glucosa en sangre se mantengan en un rango de 70-180 mg/dL. Cuando el nivel de glucosa en sangre baja por debajo de lo normal, las células α del páncreas liberan glucagón, lo cual estimula la liberación de glucosa desde el hígado al torrente sanguíneo y, por lo tanto, normaliza el nivel de glucosa. Si el nivel de glucosa en sangre es demasiado alto, las células β del páncreas secretan insulina, que estimula la captación y almacenamiento de glucosa en varios tipos de células del cuerpo, reduciendo así los niveles de glucosa en sangre. El metabolismo de la glucosa, por ende, actúa como un mecanismo de retroalimentación fisiológica que mantiene los niveles de glucosa dentro del rango mencionado anteriormente, proporcionando al cuerpo humano un suministro constante de energía.

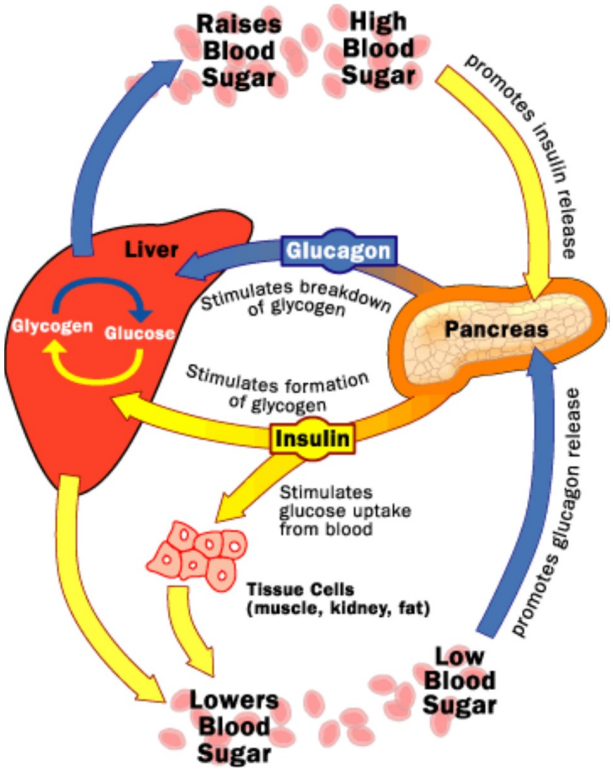


Figura 2.1: Metabolismo del carbohidrato en una persona sin diabetes [1].

Dado que en los pacientes con diabetes tipo 1 (T1D) las células β están destruidas, el paciente no puede secretar insulina. Esto provoca un metabolismo de la glucosa disfuncional, con niveles de glucosa en sangre altamente variables. Por lo tanto, los pacientes con T1D dependen completamente de inyecciones externas de insulina para sobrevivir. El Grupo de Investigación de Control y Complicaciones de la Diabetes (DCCT) ha identificado dos factores importantes para minimizar las complicaciones a largo plazo en pacientes con T1D. El primer factor es mantener los niveles de glucosa en sangre dentro del rango glucémico. El segundo factor es reducir la variabilidad glucémica, que se refiere a las fluctuaciones entre niveles bajos y altos de glucosa [4]. Así, el DCCT establece que cuanto más controlado esté el nivel de glucosa en sangre de un paciente con T1D, menos complicaciones se pueden esperar.

Niveles de glucosa fuera del rango glucémico pueden llevar a complicaciones graves. Un nivel de glucosa en sangre demasiado bajo, conocido como hipoglucemia, puede provocar convulsiones o coma, lo que puede derivar en daño cerebral e incluso la muerte, debido a la falta de suministro de energía a las células nerviosas del cuerpo humano. Un nivel de glucosa en sangre demasiado alto, denominado hiperglucemia, conlleva complicaciones más sutiles y a largo plazo, incluyendo enfermedades vasculares, trastornos de la visión, degeneración nerviosa, entre otros. Las personas que padecen T1D tienen un riesgo de sufrir este tipo de complicaciones entre tres y cinco veces mayor en comparación con las personas sanas.

2.2.0.1. Tratamiento de la diabetes tipo 1

El tratamiento de la diabetes tipo 1 se basa principalmente en la administración de insulina exógena para compensar la falta de producción natural. Este tratamiento incluye varias estrategias:

1. **Inyecciones de insulina**:La mayoría de los pacientes usan múltiples inyecciones diarias de insulina. Se administran insulinas de acción rápida antes de las comidas para controlar los picos de glucosa postprandial y insulinas de acción prolongada para mantener niveles basales de insulina a lo largo del día.

2. **Bombas de insulina**:Estos dispositivos suministran insulina de manera continua a lo largo del día a través de un catéter subcutáneo. Algunas bombas utilizan algoritmos de bucle cerrado, también conocidos como "páncreas artificial", que ajustan automáticamente la dosis de insulina en función de los niveles de glucosa medidos por sensores. Un ejemplo de bomba de insulina, que se va a utilizar para contrastar los resultados del trabajo, es la bomba Minimed 780 G, la cual se puede ver en la Figura 2.2

Figura 2.2: Bomba de insulina Minimed 780.

3. **Monitores continuos de glucosa (CGM)**:Estos sensores permiten medir los niveles de glucosa en tiempo real y alertar al paciente en caso de niveles fuera del rango normal. Los CGM se han convertido en una herramienta clave para mejorar el control glucémico y reducir el riesgo de episodios de hipoglucemia e hiperglucemia.

4. **Mediciones con glucómetros**:Para aquellos que no usan CGM, se puede medir la glucosa a través de glucómetros convencionales que analizan una pequeña gota de sangre extraída con una lanceta. Aunque no proporciona monitoreo continuo, es un método efectivo para controlar los niveles de glucosa varias veces al día.

El control adecuado de la diabetes tipo 1 requiere un equilibrio constante entre la administración de insulina, la ingesta de carbohidratos, la actividad física y la monitorización regular de los niveles de glucosa en sangre. Además, otros factores como el estrés, las enfermedades y la calidad del sueño también pueden influir en los niveles de glucosa, lo que requiere ajustes constantes en las dosis de insulina y en las decisiones diarias del tratamiento.

2.2.1. Impacto de factores adicionales

El estado anímico, el consumo de alcohol y el ejercicio físico tienen un impacto significativo en el control de la diabetes. El ejercicio puede reducir los niveles de glucosa en sangre, mientras que el alcohol puede causar variaciones en el metabolismo de la glucosa, aumentando el riesgo de hipoglucemias o hiperglucemias. Los modelos de bucle cerrado más avanzados permiten ajustar los objetivos de glucosa en función del ejercicio, mientras que el estado anímico y el estrés, aunque menos controlados por los algoritmos actuales, son áreas en desarrollo

2.2.1.1. Factores principales que afectan los niveles de glucosa

A continuación, se enumeran los principales factores que influyen en los niveles de glucosa en personas con diabetes tipo 1, y cómo afectan:

1. **Alimentación**: Los carbohidratos, especialmente los simples (azúcares refinados), tienen un impacto directo en el aumento de la glucosa en sangre. Los carbohidratos complejos, la grasa y las proteínas también aumentan la glucosa, pero de manera más lenta. Por eso, ajustar la cantidad de insulina en función de la ingesta de alimentos es crucial para el control adecuado de los niveles de glucosa.

2. **Ejercicio físico**: El ejercicio reduce los niveles de glucosa en sangre, ya que el cuerpo utiliza la glucosa como energía durante la actividad física. Sin embargo, el efecto puede ser variable dependiendo de la intensidad, duración y tipo de ejercicio. El ejercicio de alta intensidad puede causar una elevación temporal de la glucosa, seguido de una disminución.

3. **Estrés y estado anímico**: El estrés emocional o físico puede aumentar los niveles de glucosa en sangre debido a la liberación de hormonas como el cortisol y la adrenalina, que incrementan la glucosa hepática. Las emociones fuertes, ya sean positivas o negativas, también pueden afectar el control de la glucosa.

4. **Consumo de alcohol**: El alcohol puede tener efectos mixtos en la glucosa. Puede reducir los niveles de glucosa en sangre al inhibir la producción de glucosa por el hígado, lo que puede causar hipoglucemia horas después de su consumo. Sin embargo, si se mezcla con bebidas azucaradas, puede causar picos de glucosa iniciales.

5. **Insulina exógena**: La insulina exógena es fundamental para el control de la glucosa en personas con diabetes tipo 1. La dosis y el momento de la administración de la insulina afectan directamente la cantidad de glucosa en la sangre. Un error en la dosificación puede provocar hipoglucemia (si se administra demasiada) o hiperglucemia (si no es suficiente).

6. **Enfermedades**: Las infecciones y otras enfermedades agudas pueden aumentar los niveles de glucosa debido al estrés que el cuerpo experimenta durante la enfermedad. La fiebre y las infecciones provocan la liberación de hormonas de estrés, lo que incrementa la glucosa en sangre.

7. **Medicamentos**: Algunos medicamentos, como los corticosteroides, pueden aumentar los niveles de glucosa, mientras que otros, como los betabloqueantes, pueden enmascarar los síntomas de la hipoglucemia. Es importante que las personas con diabetes tipo 1 monitoreen su glucosa cuando toman nuevos medicamentos.

8. **Ciclo menstrual y cambios hormonales**: En las mujeres, las fluctuaciones hormonales durante el ciclo menstrual pueden causar variaciones en los niveles de glucosa. Por ejemplo, los niveles de estrógeno y progesterona afectan la sensibilidad a la insulina, lo que puede provocar hiperglucemia en la segunda mitad del ciclo.

9. **Sueño**: La falta de sueño puede afectar el metabolismo de la glucosa y la sensibilidad a la insulina, lo que lleva a un aumento de los niveles de glucosa en sangre. Un mal descanso también puede causar desajustes hormonales que impactan negativamente en el control de la diabetes.

10. **Temperaturas extremas**: Tanto el calor como el frío extremos pueden afectar los niveles de glucosa. El calor puede aumentar la absorción de insulina y llevar a hipoglucemias, mientras que el frío extremo puede causar un aumento de los niveles de glucosa debido a la liberación de hormonas de estrés.

2.2.1.2. Factores menos evidentes que afectan la glucosa

Además de los factores más conocidos, hay otros menos evidentes que también influyen en los niveles de glucosa en personas con diabetes tipo 1:

1. **Fenómeno del Alba (Dawn Phenomenon)**: Durante las primeras horas de la mañana, muchas personas experimentan un aumento en los niveles de glucosa en sangre debido a la liberación de hormonas del crecimiento, cortisol y adrenalina. Estas hormonas reducen la sensibilidad a la insulina, lo que provoca una subida de la glucosa. Esto es común en personas con diabetes tipo 1 y puede requerir ajustes en la dosis de insulina basal [5].

2. **Efecto Somogyi**: Este fenómeno ocurre cuando una hipoglucemia nocturna provoca una respuesta de rebote en la que el cuerpo libera hormonas que elevan los niveles de glucosa. Las personas pueden despertar con niveles de glucosa anormalmente altos sin darse cuenta de que experimentaron una hipoglucemia nocturna [6].

3. **Viajes y cambios de zona horaria**: Viajar, especialmente cruzando varias zonas horarias, puede alterar el ciclo circadiano del cuerpo, lo que afecta la administración de insulina y los niveles de glucosa. Es posible que se necesiten ajustes en los horarios de administración de insulina, especialmente en vuelos largos [7].

4. **Microbioma intestinal**: Investigaciones recientes sugieren que la composición del microbioma intestinal puede influir en la respuesta del cuerpo a la insulina y en los niveles de glucosa. Aunque es un campo de investigación emergente, ciertos patrones de bacterias intestinales están asociados con una mejor regulación de la glucosa [8].

5. **Calidad del aire**: Estudios han demostrado que la contaminación del aire, en particular la exposición a partículas finas (PM2.5 - Materia particulada de un diámetro menor de 2.5 micrómetros), puede aumentar el riesgo de hiperglucemia y resistencia a la insulina. Las personas con diabetes tipo 1 en áreas con alta contaminación del aire podrían tener más dificultades para controlar sus niveles de glucosa [2].

6. **Sensibilidad a la insulina según la hora del día**: La sensibilidad a la insulina no es constante a lo largo del día. Por la mañana, la sensibilidad a la insulina tiende a ser más baja, lo que significa que el cuerpo necesita más insulina para procesar la glucosa. Esto es algo que se debe tener en cuenta al ajustar las dosis diarias.

7. **Altitud**: La altitud puede afectar el rendimiento de los dispositivos de monitoreo continuo de glucosa (CGM) y alterar la forma en que el cuerpo metaboliza la glucosa y la insulina. Las personas que viven o viajan a grandes alturas pueden necesitar ajustar su régimen de insulina [9].

8. **Vuelos**: Un factor menos evidente que afecta el uso de bombas de insulina durante los vuelos es el cambio rápido de presión atmosférica en los despegues y aterrizajes. Este cambio puede provocar el desplazamiento de la insulina dentro del catéter, debido a la expansión o contracción del aire atrapado en el sistema. Como resultado, puede administrarse una dosis incorrecta de insulina, lo que podría llevar a un exceso o déficit inesperado. Este fenómeno es importante para los usuarios de bombas de insulina, ya que podrían experimentar hipoglucemias o hiperglucemias inesperadas si no se tiene en cuenta este efecto durante los viajes aéreos.

9. **Deshidratación**: Otro factor menos evidente que afecta a los niveles de glucosa en personas con diabetes tipo 1 es la deshidratación. Cuando el cuerpo no tiene suficiente agua, la concentración de glucosa en sangre aumenta, ya que el volumen de líquido en el torrente sanguíneo se reduce. Además, los riñones, que juegan un papel crucial en la eliminación del exceso de glucosa a través de la orina, no pueden funcionar adecuadamente en condiciones de deshidratación, lo que exacerba el aumento de glucosa en sangre. Esto puede dificultar el control de la glucosa y llevar a hiperglucemias si no se corrige adecuadamente.

10. **Estado de la insulina**: Un factor adicional menos evidente relacionado con el uso de insulina es el estado de la insulina cuando se expone a condiciones ambientales inadecuadas. La insulina, al estar expuesta al aire o a temperaturas no frías durante varios días, puede perder efectividad. La insulina debe mantenerse en un rango de temperatura específico (generalmente entre 2 y 8 °C) para asegurar su estabilidad y eficacia. Si se expone a temperaturas cálidas o al aire durante mucho tiempo, su estructura se degrada, lo que puede resultar en una menor capacidad para controlar los niveles de glucosa en sangre.

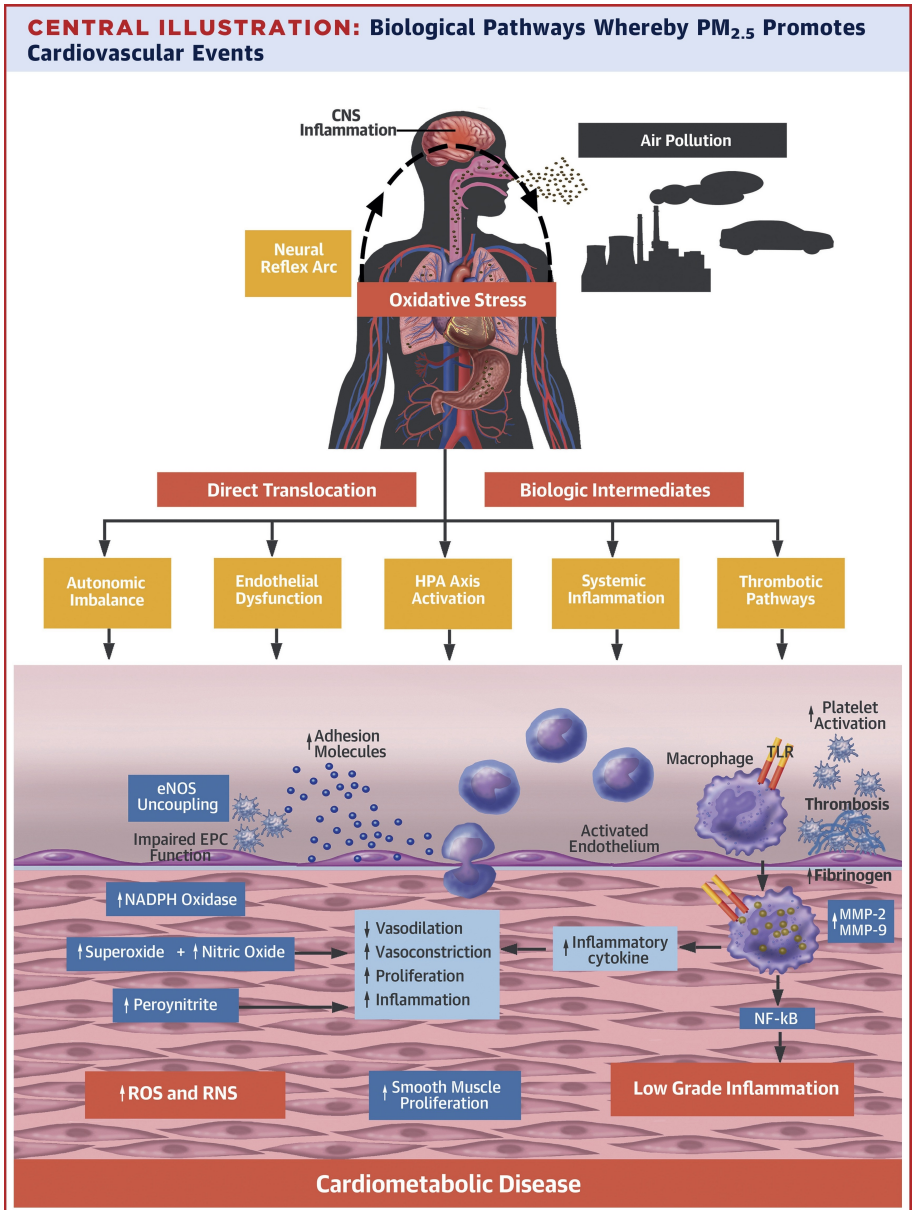


Figura 2.3: Vías biológicas por las que las partículas PM2.5 pueden promover la enfermedad cardiometabólica, según [2].

2.2.2. Limitaciones de las bombas de insulina

Las bombas de insulina actuales, incluidas aquellas con algoritmos de bucle cerrado o híbrido, no contemplan algunos de los factores menos evidentes que afectan los niveles de glucosa, como el estrés, la altitud o la calidad del aire, por varias razones:

1. **Limitaciones en los sensores disponibles**: Las bombas de insulina dependen de los monitores continuos de glucosa (CGM), los cuales miden los niveles de glucosa en sangre pero no tienen la capacidad de detectar factores como el estrés, la calidad del aire o las variaciones en el microbioma intestinal. Estos sensores están diseñados específicamente para controlar los niveles de glucosa en tiempo real y no pueden monitorear factores fisiológicos o ambientales complejos.

2. **Dificultad para cuantificar datos subjetivos**: Factores como el estrés o la falta de sueño afectan los niveles de glucosa, pero su impacto es difícil de cuantificar de manera precisa. Las bombas no pueden medir los niveles de hormonas como el cortisol, que aumentan durante el estrés, ni interpretar la calidad del sueño de una persona. Mientras que la alimentación y el ejercicio pueden ser datos que los usuarios ingresan de forma manual, factores menos evidentes no tienen una representación cuantificable fácil para que los algoritmos de la bomba lo consideren.

3. **Complejidad en la integración de datos externos**: Aunque sería útil integrar datos adicionales como el ciclo circadiano o la altitud, esto requeriría la inclusión de múltiples fuentes de datos y sensores adicionales, lo que incrementaría la complejidad del sistema. Integrar información de dispositivos de seguimiento del sueño o de sensores de altitud, por ejemplo, complicaría el algoritmo de control de la insulina, lo que podría introducir más incertidumbre y potenciales errores en los cálculos de la dosis.

4. **Falta de evidencia sólida**: Aunque se ha demostrado que factores como el microbioma intestinal o la contaminación del aire pueden afectar los niveles de glucosa, estos factores aún no están completamente comprendidos ni son predecibles de manera constante. Los sistemas de bombas de insulina dependen de algoritmos basados en evidencia sólida para realizar ajustes automáticos, y muchos de estos factores menos evidentes aún no tienen modelos predictivos suficientemente robustos para ser integrados en los dispositivos actuales.

5. **Seguridad y regulaciones**: Las bombas de insulina están sujetas a estrictas normativas y aprobaciones de entidades reguladoras como la FDA. Incluir factores menos evidentes requeriría pruebas extensas para garantizar que el sistema pueda manejar de manera segura situaciones impredecibles. Añadir capas adicionales de complejidad sin un claro beneficio podría retrasar la adopción de estas tecnologías debido a las preocupaciones de seguridad[10].

6. **No contemplación del estado de la insulina**: Una limitación importante de las bombas de insulina es que no contemplan el estado de la insulina que se utiliza en el reservorio. Los fabricantes de dispositivos como Medtronic recomiendan cambiar el reservorio de insulina cada 2 a 3 días, ya que la insulina puede perder efectividad si se expone al aire o a temperaturas no adecuadas. Aunque existen reservorios extendidos que permiten su uso hasta 7 días, en los últimos días la insulina puede comenzar a degradarse, lo que afecta su capacidad para controlar adecuadamente los niveles de glucosa. Esta degradación no es monitoreada por la bomba, lo que requiere que el paciente esté atento a los tiempos de cambio.

2.2.3. Posibles soluciones: input manual del usuario

Permitir que los pacientes o usuarios ingresen manualmente datos adicionales, como el estado anímico, la calidad del sueño o el ejercicio físico, podría ser útil para personalizar los algoritmos de las bombas de insulina. Sin embargo, hay varios factores que hacen que esta idea sea compleja y potencialmente arriesgada:

1. **Precisión de los datos ingresados manualmente:** Los datos que ingresa el usuario

pueden no ser precisos o consistentes. Por ejemplo, la percepción del nivel de estrés, el cansancio o la cantidad de ejercicio pueden variar ampliamente de una persona a otra. Ingresar estos datos subjetivos puede llevar a errores en la interpretación del algoritmo, lo que podría resultar en una administración inadecuada de insulina, causando hipoglucemia o hiperglucemia.

2. **Riesgo de sobrecarga de información:** Ingresar múltiples variables adicionales aumenta la complejidad del sistema, lo que podría saturar tanto al usuario como al algoritmo. Esto no solo implica más trabajo para el paciente, sino que también aumenta las posibilidades de error humano. Además, hay un riesgo de sobreajuste del algoritmo a factores que pueden no ser completamente comprensibles o predecibles en cada situación, lo que podría generar respuestas inadecuadas del sistema.

3. **Seguridad:** Los sistemas automatizados de insulina están diseñados para priorizar la seguridad del usuario, y cualquier cambio en los algoritmos de bucle cerrado debe pasar por rigurosas pruebas para garantizar que no aumente el riesgo de hipoglucemias o hiperglucemias peligrosas. Permitir que el usuario influya directamente en el sistema mediante la introducción de datos adicionales podría hacer que el comportamiento del sistema sea menos predecible y aumentar los riesgos para la salud.

4. **Investigación limitada por financiamiento:** La falta de integración de estos factores puede estar relacionada con una combinación de falta de interés y financiación limitada. Muchos proyectos de investigación en diabetes están enfocados en mejorar la precisión de los sensores de glucosa y la eficacia de los algoritmos de bucle cerrado actuales, y no tanto en la integración de datos subjetivos. Además, desarrollar dispositivos y algoritmos que incorporen estas variables menos evidentes requeriría financiación adicional para investigación y ensayos clínicos, lo que no siempre es fácil de obtener.

5. **Falta de estandarización y evidencia sólida:** Aún no existe una evidencia sólida o una estandarización sobre cómo ciertos factores, como el estado anímico o la altitud, afectan a la glucosa de manera predecible. Esto podría explicar la falta de interés en desarrollarlos más ampliamente en las bombas de insulina comerciales. Se están investigando estos factores, pero es posible que las conclusiones sean difíciles de traducir en algoritmos prácticos sin una comprensión más profunda.

6. **Estado de la insulina:** Una posible solución para abordar la limitación relacionada con el estado de la insulina en las bombas sería implementar ajustes automáticos en la dosis administrada durante los últimos días de uso del reservorio, cuando la insulina podría empezar a perder efectividad debido a su exposición al aire o temperaturas inadecuadas. Esto compensaría la degradación y mantendría un control más estable de los niveles de glucosa. Además, la integración de un sensor interno en la bomba que monitoree la calidad de la insulina permitiría detectar posibles cambios en su estabilidad, ajustando la dosis automáticamente o alertando al usuario para que cambie el reservorio. Esta solución optimizaría el control del tratamiento y reduciría la carga de gestión del paciente.

2.3. Modelos y simulación de diabetes

Existen plataformas como LoopInsighT1 [11], un marco de simulación de código abierto que permite simular la interacción entre el cuerpo de una persona con diabetes tipo 1 y un sistema AID (administración automática de insulina). Este modelo incluye la respuesta fisiológica a factores externos como la comida, la insulina administrada y la actividad física. Los datos se visualizan en gráficos interactivos y permiten probar diferentes configuraciones de los sistemas AID. Este tipo de herramienta es útil para la educación, el entrenamiento médico y la investigación en diabetes, ofreciendo una visión detallada del control de la glucosa mediante bucles cerrados. Un ejemplo de la simulación se puede ver en la Figura 2.4

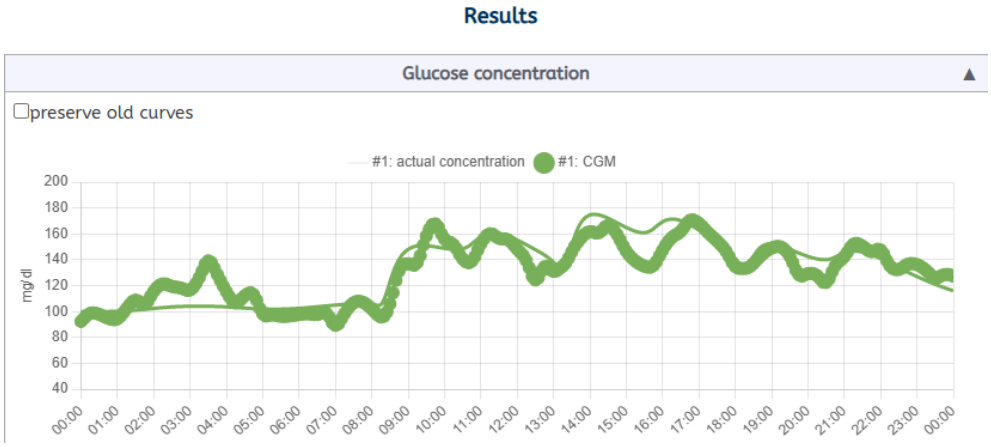


Figura 2.4: Interfaz de usuario del simulador de LoopInsighT1.

El código fuente de LoopInsighT1 está disponible en GitHub [12] bajo una licencia MIT, lo que lo convierte en una opción totalmente de código abierto. Esta plataforma simula la interacción entre una persona con diabetes tipo 1 y un sistema automatizado de administración de insulina (AID). El simulador está programado en JavaScript y usa herramientas como Vue.js para la visualización gráfica, Chart.js para los gráficos y Webpack para empaquetar los módulos de JavaScript y servir como servidor local. El simulador permite realizar simulaciones directamente en el navegador, lo que significa que los cálculos se ejecutan en el dispositivo del usuario sin necesidad de enviar datos externos. El código se puede modificar para adaptarse a distintos escenarios de simulación. Además, el proyecto incluye algoritmos AID como OpenAPS (oref0) y controladores PID para el ajuste de insulina (ver siguientes secciones).

A continuación se comentan dos de los modelos de diabetes que implementan el modelo de LoopInsightT1.

2.3.1. UVA/PADOVA Type 1 Diabetes Simulator (T1DMS)

El UVA/Padova es uno de los simuladores más utilizados para modelar la diabetes tipo 1. Fue desarrollado en colaboración entre la Universidad de Virginia y la Universidad de Padova, cuyas ecuaciones han sido publicadas por Dalla Man et al [13]. Este simulador permite investigar la respuesta fisiológica de pacientes virtuales a diferentes estrategias de administración de insulina, incluyendo algoritmos AID. Se utiliza en estudios clínicos y regulatorios, siendo aprobado

por la FDA como una plataforma para probar sistemas de páncreas artificial. Aunque no está disponible como código abierto, ha sido ampliamente utilizado en investigaciones y publicaciones científicas, particularmente para el desarrollo de tecnologías de control automático de la diabetes.

Además, estudia también como modelar la actividad física dentro de este modelo [14]. Describe cómo cambia la dinámica glucosa-insulina en dos fases: una variación rápida de incremento y disminución en la eliminación de glucosa que no depende de la insulina, junto con un cambio rápido al aumentar y lento al disminuir en la sensibilidad a la insulina.

Muchos estudios se han basado en este modelo para ampliarlo o centrarse en algún área específica. Es el caso del trabajo publicado por la Revista Iberoamericana de Automática e Informática Industrial [15], donde mejoran el modelo de generador de alimentos incluyendo comidas altas en proteínas y grasas. Esto se simula introduciendo el concepto de Unidad Grasa-Proteína (UGP). A diferencia de los carbohidratos, que se miden en gramos o raciones, las UGP se expresan en kcal. Un gramo de grasa equivale a 9 kcal, mientras que un gramo de proteína equivale a 4 kcal.

2.3.2. Cambridge Model

El Cambridge Diabetes Simulator, desarrollado por el Dr. Roman Hovorka y su equipo en la Universidad de Cambridge, es otro simulador ampliamente reconocido en la investigación sobre la diabetes tipo 1. Su enfoque es modelar la dinámica de la glucosa y la insulina en pacientes con diabetes, y ha sido fundamental en el desarrollo de algoritmos de administración de insulina. Este simulador es utilizado en ensayos clínicos y estudios regulatorios para investigar la efectividad de sistemas de páncreas artificial. El código no es de acceso libre, pero se utiliza con frecuencia en colaboración académica y está documentado en múltiples investigaciones [16] [17].

2.4. Sistemas de control de diabetes y suministro de insulina

2.4.1. Sistemas de bucle cerrado de código abierto

El proyecto OpenAPS es uno de los más conocidos en esta área. OpenAPS es una iniciativa de código abierto que permite a los usuarios de bombas de insulina crear su propio sistema de control automatizado. Este sistema ajusta automáticamente la administración de insulina en función de los niveles de glucosa medidos por sensores de glucosa en tiempo real. Aunque el sistema no es comercial y requiere hardware específico, ha sido adoptado por muchas personas que buscan una solución personalizada para la diabetes [18].

OpenAPS es uno de los sistemas de bucle cerrado de código abierto más conocidos en la comunidad de diabetes. Está diseñado para que los usuarios construyan sus propios sistemas de administración automatizada de insulina utilizando componentes de hardware específicos y el algoritmo oref0 desarrollado por la comunidad DIY. El código fuente de OpenAPS también

está disponible en GitHub [19] y está escrito principalmente en Python. Además, el proyecto está bien documentado para ayudar a los usuarios en la configuración del sistema, proporcionando instrucciones detalladas sobre cómo integrar las bombas de insulina compatibles y los monitores de glucosa continuos (CGM). OpenAPS es parte del movimiento "Do It Yourselfz ofrece a los usuarios la capacidad de personalizar su tratamiento, aunque requiere cierto nivel de conocimientos técnicos para implementarlo correctamente.

Sistemas Do It Yourself (DIY) El movimiento Do It Yourself (DIY) en la gestión de la diabetes ha crecido gracias a comunidades de código abierto que buscan soluciones personalizadas y no comerciales para el control de la insulina. El proyecto OpenAPS es una de las iniciativas más representativas de este movimiento, donde los usuarios diseñan y ensamblan sus propios sistemas de bucle cerrado o abierto, usando bombas de insulina más antiguas, monitores continuos de glucosa y software de código abierto. Estos sistemas DIY ofrecen una alternativa a las soluciones comerciales, permitiendo a los usuarios tener control total sobre su tratamiento. Aunque los sistemas DIY no están regulados por organismos como la FDA, son usados por muchos pacientes debido a su flexibilidad y su capacidad para mejorar el manejo de la diabetes en aquellos que desean una mayor personalización.

El movimiento DIY en el manejo de la diabetes permite a los usuarios crear sus propios sistemas de administración de insulina, basados en herramientas y algoritmos de código abierto. Incluye el desarrollo de soluciones como OpenAPS, AndroidAPS y Loop, ha permitido a los usuarios compartir sus experiencias y contribuir al desarrollo continuo de sistemas automatizados de control de insulina. Sin embargo, estos sistemas requieren un nivel avanzado de conocimiento técnico y la responsabilidad de gestionar riesgos, ya que los usuarios mismos son responsables de su construcción y uso.

2.4.2. Algoritmos comerciales de bucle cerrado

Los sistemas híbridos de bucle cerrado (HCL) comerciales, como el Control-IQ de Tandem [20] y el MiniMed 780G de Medtronic [21], utilizan algoritmos avanzados para predecir los niveles de glucosa y ajustar la administración de insulina automáticamente. Estos sistemas permiten la configuración de diferentes objetivos de glucosa para momentos específicos, como el ejercicio físico o durante la noche. Aunque estos modelos no son de código abierto, representan un avance significativo en la automatización del manejo de la diabetes.

2.4.3. Sistemas de bucle abierto (OpenLoop)

El concepto de sistemas de bucle abierto, como **OpenLoop**, se centra en proporcionar control manual a los usuarios de diabetes para ajustar la administración de insulina en lugar de confiar en un algoritmo completamente automatizado. A diferencia de los sistemas de bucle cerrado que ajustan automáticamente las dosis de insulina en función de los datos de los sensores de glucosa en tiempo real, en OpenLoop los usuarios deben analizar la información de los sensores y decidir cuánta insulina administrar en función de sus niveles de glucosa, su ingesta de carbohidratos y su actividad física. Aunque ofrece más flexibilidad y control, este enfoque requiere un monitoreo constante y un conocimiento detallado por parte del usuario sobre cómo ajustar correctamente las dosis de insulina.

Un sistema de OpenLoop puede ser útil para usuarios que prefieren mayor autonomía sobre sus decisiones terapéuticas, pero también puede incrementar el riesgo de errores si no se ajusta adecuadamente. Este enfoque se encuentra en plataformas de código abierto, como Loop [22] y AndroidAPS [23], que proporcionan herramientas para configurar sistemas de bucle abierto.

AndroidAPS, por ejemplo, incluye un modo OpenLoop en el cual la app sugiere cambios en la insulina basal, pero el usuario debe confirmarlos e ingresarlos manualmente en la bomba. Esto ofrece más control al usuario, pero también requiere un monitoreo constante y conocimientos técnicos. AndroidAPS está disponible como código abierto en GitHub [24] y está escrito en Kotlin y Java, lo que permite a los usuarios personalizarlo y ajustarlo a sus necesidades específicas. Además, la plataforma puede integrarse con otros sistemas como xDrip+ y Nightscout para el monitoreo continuo de la glucosa (CGM) y la gestión de datos.

2.4.4. Variantes al control PID

El éxito del PID en la industria se debe a la facilidad con la que se puede sintonizar, su simplicidad y flexibilidad. Además, no necesita una explicación del modelo, sino que se pueden emplear reglas empíricas. Para la dinámica glucsa-insulina, las 3 componentes del controlador PID pueden responder a estos comportamientos, en función de la variable de error, que es la diferencia entre la glucosa medida por el sensor y la glucosa objetivo.

$$e = G_{sensor} - G_{target} \tag{2.1}$$

El controlador PID se basa en tres componentes que responden a diferentes aspectos de la dinámica glucosa-insulina:

- Proporcional: es una respuesta inmediata al cambio en la glucosa. Considera la diferencia entre la glucosa actual y la deseada (objetivo). Es decir, se aumenta la cantidad de insulina cuando la glucosa medida (por el sensor) excede la glucosa objetivo, y se disminuye la inyección de insulina cuando la diferencia es negativa, ya que es un sistema unidireccional y solo se puede administrar insulina, no extraerla del cuerpo.

$$P(t) = K_P \cdot e \tag{2.2}$$

- Integral: es una respuesta a la reacción lenta de la glucosa. Permite mantener el nivel de insulina basal cuando se igualan la glucosa medida y la glucosa objetivo en estado estacionario.

$$I(t) = \int K_I \cdot e \cdot dt \tag{2.3}$$

- Derivativo: es una respuesta al cambio de velocidad en la glucosa. Además, estabiliza el sistema frente a cambios de concentración de glucosa.

$$D(t) = K_D \cdot \frac{de}{dt} \tag{2.4}$$

La Figura 2.5 esquematiza las componentes del controlador PID para la administración de insulina frente al tiempo. En la literatura se pueden encontrar las siguientes variantes al control PID:

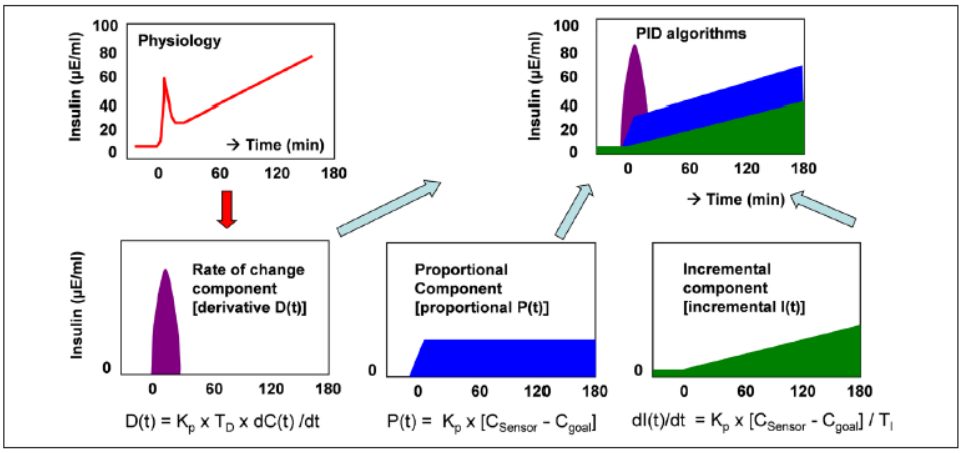


Figura 2.5: Modelo de distribución de insulina de acuerdo al algoritmo PID, según [3].

1. Adaptación: Las ganancias del sistema se ajustan automáticamente para adaptarse a perturbaciones y cambios en la dinámica del sistema. Existen varias estrategias de adaptación:

   - Ajuste de tabla: Usa una tabla que determina las ganancias en función de una variable auxiliar que clasifica los niveles de glucosa en diferentes áreas [25].
   - MRAS (Model Reference Adaptive Systems): Usa un sistema de gradientes para minimizar el error cuadrado.
   - STR (Self-Tuning Regulator): Ajusta los parámetros en tiempo real usando un estimador de parámetros recursivos y un cálculo de diseño.

2. PID inteligente (iPID): Fliess (2008) [26] propuso un controlador que sustituye el modelo matemático por ecuaciones diferenciales ultra-locales y fenomenológicas. Esto permite compensar dinámicas no modeladas y adaptarse a cada paciente, especialmente para evitar hipoglucemias nocturnas.

3. NPID en error (visión de Chen [27]): Un controlador no lineal que se coloca en cascada antes del PID clásico para mejorar el rendimiento, acelerar el tiempo de estabilización y reducir sobreoscilaciones. Introduce el error y el parámetro α en el NNTPID, mientras que en el iNNTPID cada rama del PID tiene su propio α.

4. NPID en ganancias (visión de Yousefzadeh): Combina el PID con una superficie de conmutación no lineal para mejorar la eficiencia y reducir los tiempos transitorios. Las ganancias del sistema dependen de la señal de error.

5. PID fraccional: Maneja ecuaciones diferenciales de orden no entero, introduciendo dos parámetros (λ, μ) entre 0 y 2. Esta idea fue introducida por Tustin y desarrollada por Podlubny en 1999 [28].

6. Fuzzy-PID: Un sistema de control PID donde las variables Kp, Ki y Kd se ajustan según los niveles de glucosa mediante un sistema de regulación basado en reglas difusas [29].

7. PID con control de seguridad, que protege al paciente ante posibles sobredosis de insulina y, por lo tanto, hipoglucemias, dosis negativas (ya que no se puede sustraer insulina al paciente) y obstrucciones en el catéter. También contempla el uso de ganancias reducidas

para evitar sobreoscilaciones en la glucosa. El sistema de seguridad del PID de Steil [30] anula la acción derivativa ante el descenso de la glucosa.

Controlador híbrido en lazo cerrado
En el año 2008, Weinzimer y Steil [31] propusieron un controlador híbrido en lazo cerrado basado en el controlador PID previamente desarrollado por Steil. Este sistema de control semicerrado (SCL) integra prealimentación, lo que permite la anticipación de ingestas alimentarias y la administración de bolos prandiales. Estos bolos prandiales representan entre el 25 % y el 50 % de la dosis diaria total de insulina del paciente. El controlador fue diseñado y probado específicamente en pacientes pediátricos con diabetes tipo 1.

El controlador híbrido introdujo varias mejoras al PID original. Estas mejoras incluyen un objetivo glucémico variable en función del momento del día, la administración del bolo prandial 15 minutos antes de las comidas, la saturación del componente integral del controlador y la adaptación de los tiempos derivativos e integrales según las necesidades del paciente. Además, se implementó un bolo corrector, utilizado en la transición de un control en lazo abierto a un control en lazo cerrado, mejorando la estabilidad del sistema.

Estas modificaciones permitieron una mejora significativa en la regulación de la glucosa, sobre todo en el periodo postprandial, evitando picos de hiperglucemia que normalmente ocurren después de las comidas. El controlador híbrido demostró ser eficaz al proporcionar un control más preciso y seguro en entornos ambulatorios, manteniendo los niveles de glucosa dentro de un rango adecuado y reduciendo las complicaciones metabólicas en pacientes con diabetes tipo 1.

2.5. Conclusiones

En este capítulo se ha presentado una revisión exhaustiva de las metodologías existentes para la simulación y el control de la diabetes tipo 1. Se han analizado los principales simuladores utilizados en investigación, como el UVA/Padova Simulator y el Cambridge Model, que permiten modelar con gran fidelidad la evolución de la glucosa en pacientes virtuales. También se ha introducido la herramienta LoopInsighT1, una solución de código abierto orientada a la simulación y visualización educativa e investigadora.

En cuanto a los algoritmos de control, se ha distinguido entre sistemas en bucle cerrado comerciales, como el Control-IQ de Tandem o el MiniMed 780G de Medtronic, y soluciones abiertas como OpenAPS o AndroidAPS, que han sido impulsadas por comunidades DIY. Estas últimas destacan por su flexibilidad y personalización, aunque requieren conocimientos técnicos por parte del usuario.

Una parte central del capítulo ha sido la revisión del controlador PID y sus variantes adaptativas, inteligentes y no lineales, así como su papel en el diseño de sistemas híbridos de administración automática de insulina. Este análisis justifica la elección de un controlador PID adaptativo como núcleo del algoritmo de control desarrollado en este trabajo.

Capítulo 3

Desarrollo de la herramienta DiabetesTool

3.1. Introducción

Este capítulo presenta el desarrollo de la herramienta **DiabetesTool**, diseñada para simular la evolución de los niveles de glucosa en sangre en personas con diabetes tipo 1. Se trata de una aplicación implementada en Python que integra un modelo fisiológico basado en el Cambridge Model con un motor numérico adaptativo, interfaces gráficas de simulación y análisis visual de resultados.

Además del modelado matemático y fisiológico, este capítulo describe el proceso completo de ingeniería de software seguido: desde el análisis de requisitos funcionales y no funcionales, hasta el diseño modular de la arquitectura y la implementación detallada de sus principales componentes.

La estructura del capítulo es la siguiente:

- En la Sección 3.2, se presentan los requisitos de la aplicación DiabetesTool, tanto funcionales como técnicos.
- En la Sección 3.3, se describe la arquitectura de la herramienta, así como los principales módulos software y su interacción.
- En la Sección 3.4, se detallan los aspectos del modelo de simulación, la lógica de eventos y el método numérico adaptativo usado.
- En la Sección 3.5, se muestran ejemplos de uso de la herramienta y simulaciones ilustrativas.

3.2. Análisis del sistema DiabetesTool

DiabetesTool es una herramienta de simulación orientada a emular el comportamiento de la glucosa en sangre en pacientes con diabetes tipo 1. Está concebida como un software modular y personalizable, con capacidad para integrar datos reales o simulados, eventos externos, y estrategias de administración de insulina.

3.2.1. Requisitos funcionales

- RF1: Simular la evolución de la glucosa en sangre a partir de un conjunto de parámetros fisiológicos del paciente.
- RF2: Integrar eventos externos como comidas, ejercicio, alcohol o estados emocionales, y reflejar su impacto.
- RF3: Mostrar gráficamente los resultados, tanto en modo estático como en modo dinámico.
- RF4: Permitir el uso de diferentes configuraciones a través de ficheros JSON de entrada (params, inputs, events).

3.2.2. Requisitos no funcionales

- RNF1: Implementación eficiente del motor de simulación mediante métodos numéricos adaptativos.
- RNF2: Diseño extensible y modular que permita añadir nuevos eventos o parámetros.
- RNF3: Interfaz gráfica clara y funcional para la visualización de resultados.

3.3. Diseño de la arquitectura y módulos principales

La herramienta está estructurada en diferentes módulos software, desarrollados en Python, que encapsulan las funcionalidades del sistema. La arquitectura general se basa en una separación clara entre el *modelo de simulación*, el *motor numérico*, los *eventos externos*, los *ficheros de configuración* y la *interfaz gráfica*.

Los principales componentes son los siguientes:

- `cambridge_model.py`: Implementación del modelo fisiológico (variables de estado, parámetros, ecuaciones).
- `solver.py`: Motor numérico adaptativo (RK1–RK2).
- `events.py`: Lógica de aplicación de eventos externos sobre parámetros del modelo.
- `main.py` y `plot.py`: Módulos para lanzar simulaciones estáticas o dinámicas y visualizar los resultados.
- `inputs.json`, `params.json`, `events.json`: Archivos de entrada que definen las condiciones de simulación.

3.4. Implementación del modelo y motor de simulación

El modelo de simulación implementado se basa en una serie de parámetros fisiológicos y variables de estado que permiten representar de forma precisa la evolución de la glucosa en sangre en respuesta a diferentes estímulos. Esta sección describe en detalle tanto los parámetros que caracterizan el comportamiento del sistema (como sensibilidades, volúmenes o retardos), como las variables dinámicas que representan su evolución temporal.

3.4.1. Parámetros y variables de estado

El modelo utiliza varios parámetros fisiológicos y variables de estado que permiten calcular la evolución de la glucosa en sangre en respuesta a diferentes estímulos, tanto internos como externos. Estos parámetros y variables permiten una personalización del modelo para adaptarse a las características individuales de cada paciente.

3.4.1.1. Parámetros

A continuación, en la Tabla 3.1, se enumeran los principales parámetros utilizados en el modelo, junto con una breve definición, valores típicos y las unidades de los parámetros:

3.4.1.2. Variables de estado

Las variables de estado principales utilizadas en el modelo incluyen:

- Q_1 y Q_2: números de moles de glucosa en los compartimentos accesible (plasma) y no accesible, respectivamente (mmol).
- I: concentración de insulina en plasma (mU/L).
- S_1 y S_2: cantidades de insulina en los compartimentos de absorción subcutánea (mU).
- x_1, x_2, x_3: efectos de la insulina sobre el transporte de glucosa (1/min), la eliminación de glucosa (1/min) y la producción endógena de glucosa (adimensional), respectivamente.
- D_1 y D_2: glucosa en los compartimentos del sistema digestivo, representando la absorción intestinal (mmol).

En la Tabla 3.2 se muestran estas variables de estado.

Tabla 3.1: Resumen de parámetros del modelo

Parámetro	**Valor típico**	**Unidades**	**Significado**
V_G	0.16	l/kg	Volumen de distribución de glucosa
SIT	0.025	$\text{min}^{-1}/(\text{mU/L})$	Sensibilidad insulínica tisular (tejidos periféricos)
SID	0.0021	$\text{min}^{-1}/(\text{mU/L})$	Sensibilidad insulínica hepática (inhibición de producción de glucosa hepática)
SIE	0.012	$1/(\text{mU/L})$	Sensibilidad insulínica del compartimento de eliminación
EGP_0	0.0161	mmol/kg/min	Producción endógena basal de glucosa
F_{01}	0.0097	mmol/kg/min	Tasa de consumo de glucosa independiente de la insulina
V_I	0.12	l/kg	Volumen de distribución de insulina
k_e	0.138	min^{-1}	Tasa de eliminación de insulina
$t_{max,I}$	55	min	Tiempo hasta el pico de absorción de insulina
$t_{max,G}$	40	min	Tiempo hasta el pico de absorción de glucosa
A_G	0.8	-	Biodisponibilidad de glucosa (fracción absorbida)
BW	70	kg	Peso corporal del paciente
$Gpeq$	140	mg/dL	Glucosa basal en equilibrio (objetivo)
k_{12}	0.066	min^{-1}	Tasa de transferencia de glucosa entre compartimentos
k_{a1}	0.006	min^{-1}	Tasa de absorción de insulina en el primer compartimento
k_{a2}	0.06	min^{-1}	Tasa de absorción de insulina en el segundo compartimento
k_{a3}	0.3	min^{-1}	Tasa de absorción de insulina en el tercer compartimento
i_delay	30	min	Retardo en la acción de la insulina sobre la glucosa
m_delay	-5	min	Retardo en la acción de la comida sobre la glucosa

Tabla 3.2: Variables de estado del modelo

Símbolo	**Significado**	**Unidades**
Q_1	Glucosa en el compartimento accesible (plasma)	mmol
Q_2	Glucosa en el compartimento no accesible (tejido intersticial)	mmol
S_1	Insulina subcutánea en el compartimento de absorción 1	mU
S_2	Insulina subcutánea en el compartimento de absorción 2	mU
I	Insulina en plasma	mU/L
x_1	Acción de la insulina sobre el transporte de glucosa	1/min
x_2	Acción de la insulina sobre la eliminación de glucosa	1/min
x_3	Acción de la insulina sobre la producción endógena de glucosa	(adimensional)
D_1	Glucosa en el compartimento digestivo 1	mmol
D_2	Glucosa en el compartimento digestivo 2	mmol

3.4.1.3. Absorción de glucosa

El subsistema de glucosa tiene como input la cantidad de carbohidratos d que son ingeridos por el individuo por unidad de tiempo. Aunque sus unidades son g/min, el modelo realiza internamente la conversión entre gramos y milimoles ($g \rightarrow mmol$). La absorción y posterior conversión de los carbohidratos a glucosa se realiza mediante los compartimentos D_1 y D_2. Hay que considerar la biodisponibilidad de carbohidratos A_G, por lo que $d = Meal \cdot A_G$, y la tasa de absorción máxima $t_{max,G}$. La ecuación que describe la derivada de la glucosa en el compartimento D_1 es:

$$\frac{dD_1}{dt} = d \cdot A_G - \frac{D_1}{t_{max,G}} \tag{3.1}$$

Para el compartimento D_2:

$$\frac{dD_2}{dt} = \frac{D_1}{t_{max,G}} - \frac{D_2}{t_{max,G}} \tag{3.2}$$

Finalmente, la absorción de glucosa desde el tracto gastrointestinal, calculada con la cantidad de glucosa en el compartimento digestivo D_2 y el tiempo de absorción máxima de glucosa $t_{max,G}$ es:

$$U_G = \frac{D_2}{t_{max,G}} \quad [\text{mmol/min}] \tag{3.3}$$

3.4.1.4. Absorción de insulina

El modelo incluye un sistema de absorción de insulina subcutánea representado por S_1 y S_2, donde $t_{max,I}$ es el tiempo hasta alcanzar la máxima absorción de insulina. La tasa de cambio de insulina en el compartimento S_1 se describe mediante la siguiente ecuación:

$$\frac{dS_1}{dt} = \text{IIR} - \frac{S_1}{t_{max,I}} \tag{3.4}$$

donde IIR es la tasa de infusión de insulina en mU/min. Este parámetro es un input del sistema, ya que se trata del ratio de la insulina administrada al paciente. La tasa de cambio en el compartimento S_2 se describe por:

$$\frac{dS_2}{dt} = \frac{S_1}{t_{max,I}} - \frac{S_2}{t_{max,I}} \tag{3.5}$$

La derivada de la concentración de insulina en plasma I se modela como:

$$\frac{dI}{dt} = \frac{S_2}{t_{max,I} \cdot V_I \cdot BW} - I \cdot k_e \tag{3.6}$$

donde V_I es el volumen de distribución de la insulina, BW es el peso corporal del paciente y k_e es la tasa de eliminación de la propia insulina.

3.4.1.5. Regulación de la glucosa

La glucosa absorbida se distribuye dentro del sistema de regulación de la glucosa de la siguiente forma. Las variables de estado Q_1 [mmol] y Q_2 [mmol] representan la cantidad de glucosa en el flujo sanguíneo principal y en el tejido periférico, respectivamente. La ecuación que describe la tasa de cambio del compartimento uno se establece de la siguiente manera:

$$\frac{dQ_1}{dt} = k_{12} \cdot Q_2 - x_1 \cdot Q_1 + \text{EGP} - F_{01} - FR + U_G \tag{3.7}$$

Es interesante describir cada uno de los términos implicados:

- k_{12} es la tasa de transferencia de glucosa entre compartimentos [1/min]. Por lo tanto $k_{12} \cdot Q_2$ esla glucosa que proviene del compartimento Q_2.
- x_1 es la acción de la insulina sobre el transporte de glucosa, calculada mediante:

  $$x_1 = \text{SIT} \cdot I \quad [1/\text{min}] \tag{3.8}$$

  Entonces, $x_1 \cdot Q_1$ es la cantidad de glucosa "transportada"desde $Q1$, cuya magnitud depende de la sensibilidad al transporte por parte de la insulina, y de la cantidad de insulina.

- EGP representa la producción endógena de glucosa por parte del hígado:

$$\text{EGP} = \text{EGP}_0 \cdot BW \cdot (1 - x_3) \quad [\text{mmol/min}] \tag{3.9}$$

  x_3 se obtiene como:

$$x_3 = \text{SIE} \cdot I \quad [1] \tag{3.10}$$

  SIE es la sensibilidad de la insulina a la producción endógena [L/mU].

- F_{01} es el flujo de glucosa no dependiente de insulina y se calcula como:

$$F_{01} = F_{01}^{\text{c}} \cdot BW \quad [\text{mmol/min}] \tag{3.11}$$

  donde F_{01}^{c} es un coeficiente corregido por la concentración de glucosa:

$$F_{01}^{\text{c}} = F_{01} \cdot \min\left(\frac{G}{4.5}, 1\right) \tag{3.12}$$

  y G es la concentración de glucosa en plasma [mmol/L]. Esto quiere decir, que si la glucosa es mayor a 4.5 mmol/L (81 mg/dL), el parámetro F_{01} aumenta ligeramente.

- FR es la liberación renal de glucosa, que se activa cuando la glucosa en plasma supera un los 9 mmol/L (162 mg/dL) y se modela como:

$$FR = \max(0, 0.003 \cdot (G - 9) \cdot V_G) \quad [\text{mmol/min}] \tag{3.13}$$

- U_G es la absorción de glucosa desde el tracto gastrointestinal, calculada como:

$$U_G = \frac{D_2}{t_{max,G}} \quad [\text{mmol/min}] \tag{3.14}$$

La derivada para el numero de milimoles en el compartimento no accesible Q_2 es:

$$\frac{dQ_2}{dt} = x_1 \cdot Q_1 - (k_{12} + x_2) \cdot Q_2 \tag{3.15}$$

donde x_2 es la acción de la insulina sobre la eliminación de glucosa.

Las ecuaciones para la acción de la insulina sobre el transporte, la eliminación y la producción endógena de glucosa (x_1, x_2, x_3) se modelan como:

$$\frac{dx_i}{dt} = I \cdot k_{bi} - x_i \cdot k_{ai} \quad \text{para } i = 1, 2, 3 \tag{3.16}$$

donde k_{bi} y k_{ai} son los coeficientes de acción de la insulina correspondientes a cada efecto.

Estos detalles, basados en el modelo de Hovorka [16], aseguran que el sistema pueda simular de manera precisa la respuesta del cuerpo a diferentes estímulos, como la ingesta de alimentos y la administración de insulina. En la Figura 3.1 se muestra el modelo en forma de diagrama y separado por compartimentos.

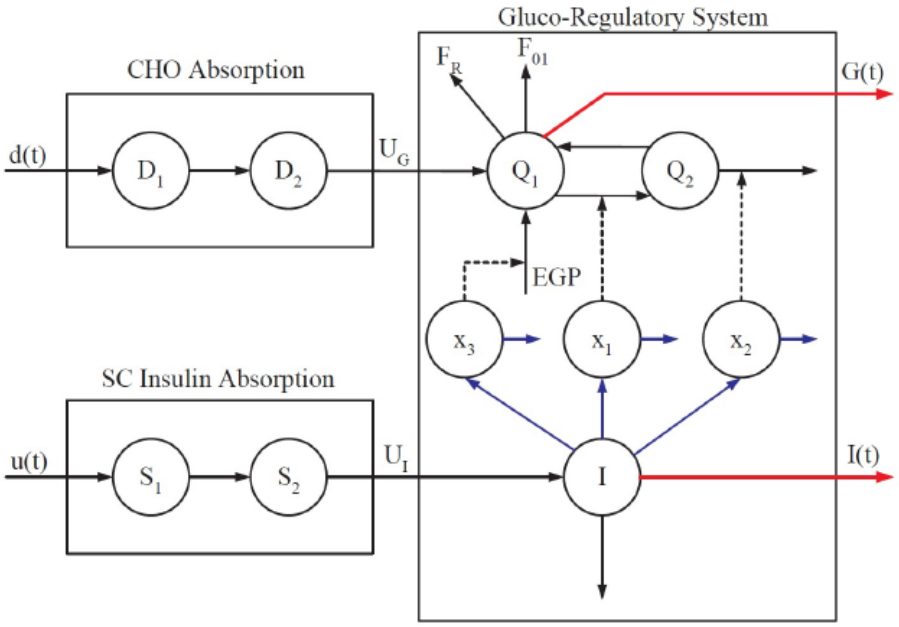


Figura 3.1: Diagrama del modelo separado en compartimentos [1].

3.4.2. Modelización de los eventos

Para reflejar situaciones de la vida diaria, el modelo integra una serie de eventos externos que afectan los niveles de glucosa en sangre:

- **Alimentos**: La ingesta de alimentos se representa a través de la cantidad de carbohidratos y la absorción de grasas y proteínas (UGP), permitiendo calcular su impacto en los niveles de glucosa.
- **Insulina**: Se integran tanto bolos como infusiones basales de insulina, cuyo efecto se modela mediante funciones de absorción y eliminación que se ajustan al perfil farmacocinético de la insulina.
- **Ejercicio**: El modelo incluye tanto ejercicio de alta como de baja intensidad, ajustando la sensibilidad a la insulina y la producción de glucosa en función de la duración e intensidad del ejercicio.
- **Alcohol**: La ingesta de alcohol afecta la producción de glucosa hepática y la sensibilidad a la insulina, y su efecto se modela en función de la cantidad consumida y la duración de su impacto.
- **Estado anímico y enfermedades**: Factores como el estrés, la fiebre o infecciones incrementan la producción de glucosa y reducen la sensibilidad a la insulina.

3.4.2.1. Alimentos

La ingesta de alimentos se modela en términos de la absorción de carbohidratos y la absorción de grasas y proteínas (UGP). La tasa de absorción de carbohidratos se calcula con la siguiente

ecuación:

$$\text{Meal} = \text{carbs (g/min)} \cdot \frac{1000}{180.16} \quad \text{[mmol/min]} \tag{3.17}$$

En esta ecuación, 1000/180.16 es el factor de conversión que permite convertir gramos de carbohidratos a milimoles. El numerador 1000 convierte los gramos a miligramos, y el denominador 180.16 es la masa molar de la glucosa en mg/mmol.

En [15] se introduce una mdodelización de las UGPs. La absorción de grasas y proteínas se modela mediante Unidades Grasa-Proteína (UGP). Una UGP corresponde a un aporte energético que afecta los niveles de glucosa a un ritmo más lento y prolongado en comparación con los carbohidratos. La absorción de UGP se distribuye linealmente durante el tiempo de digestión. La duración total de la digestión es función del número de UGPs. La duración es de 3 horas para 1 UGP, y aumenta 1 hora adicional por cada UGP adicional

$$\text{carbs(mmol/min)} \cdot \left(\frac{t_{\text{elapsed}}}{\text{ugp_duration}} \right) \quad \text{[mmol/min]} \tag{3.18}$$

donde t_{elapsed} es el tiempo transcurrido desde el inicio de la ingesta, ugp_duration es la duración total de la absorción calculada en función del número de UGPs.

No hay muchos artículos en la literatura que se adentren en el tema de las UGPs. Además, algunos endocrinos afirman que no hay una evidencia de que se pueda aplicar una norma para todo paciente. Sin embargo, tanto en [32] como en [33] se explica en detalle cómo se puede administrar insulina en función de este tipo de alimentos altos en proteínas y grasas. Es más, teniendo en cuenta estas UGPs, se pueden explicar las hiperglucemias que se observan en la experiencia tras ingerir pizzas, hamburguesas, mucha comida con pollo etc.

3.4.2.2. Insulina

El modelo integra bolos e infusiones basales de insulina. La cantidad de insulina en el compartimento de absorción subcutánea se calcula mediante la tasa de infusión de insulina (IIR) con la ecuación:

$$\text{IIR (U/h)} \cdot \frac{1000}{60} \cdot t_{max,I} \quad \text{[mU]} \tag{3.19}$$

donde $t_{max,I}$ es el tiempo hasta la absorción máxima de la insulina en minutos, y $\frac{1000}{60}$ es el factor de conversión de U/h a mU/min.

La eficacia de la insulina disminuye con el tiempo y la temperatura. Los viales de insulina se guardan en los frigoríficos ya que tienen que estar a temperaturas bajas. Para las bombas de insulina, se recomienda cambiar el reservorio cada 3 días como mucho, ya que al estar la bomba pegada al cuerpo la temperatura es mucho más alta que la temperatura ambiente. Normalmente empieza a perder eficacia transcurridos 2.5 días. La fórmula para el ajuste de la eficacia es:

$$\text{efficacy_factor} = \max\left(0, 1 - \frac{\text{days_since_change} - 2.5}{2.0}\right) \quad \text{[adimensional]} \tag{3.20}$$

Se ha ajustado esta ecuación para que se cumpla lo siguiente: cuando $2.5 \leq \text{days_since_change} < 4$, y se reduce a 0.25 para más de 4 días sin cambio de reservorio.

3.4.2.3. Ejercicio

El ejercicio físico impacta directamente en la sensibilidad a la insulina y en la producción endógena de glucosa (EGP_0). En el modelo, estos efectos se ajustan en función de la duración y la intensidad del ejercicio, permitiendo una representación precisa de cómo varían las respuestas fisiológicas durante y después del ejercicio.

El impacto en EGP_0 se representa mediante un factor de ajuste que depende de la intensidad del ejercicio. Cuando la intensidad del ejercicio es alta (mayor a 5 en una escala de 1 a 10), EGP_0 se incrementa en los primeros 5 minutos de ejercicio, seguido por una estabilización y luego una reducción gradual en los 10 minutos posteriores al final del ejercicio:

$$\text{EGP}_{0modify} = \text{EGP}_0 \cdot \text{factor de ajuste} \quad \text{[mmol/kg/min]} \tag{3.21}$$

Donde el factor de ajuste varía según la intensidad del ejercicio, siendo proporcional a un valor base de $0.4 \cdot \text{intensidad}$.

La sensibilidad a la insulina se ajusta aumentando los parámetros SIT, SID y SIE, lo que refleja un incremento en la capacidad del cuerpo para transportar, eliminar y reducir la producción endógena de glucosa. Estos ajustes se aplican durante la fase posterior al ejercicio y dependen también de la duración e intensidad del mismo:

- Para ejercicios de alta intensidad, los ajustes en la sensibilidad se realizan en tres fases: un aumento progresivo al inicio de la recuperación, un mantenimiento de la sensibilidad incrementada y una reducción gradual de la sensibilidad hasta volver a los niveles basales.
- Para ejercicios de baja intensidad, los cambios en la sensibilidad se aplican de manera más suave, con un incremento leve y un retorno progresivo a los valores normales.

3.4.2.4. Alcohol

La ingesta de alcohol afecta la producción de glucosa hepática y reduce EGP_0. La fórmula que modela este efecto es:

$$\text{reduction_factor} = \begin{cases} 0.7, & \text{si la cantidad de alcohol} \geq 6 \text{ (unidades)} \\ 1 - 0.05 \cdot \text{cantidad}, & \text{si la cantidad de alcohol} < 6 \text{ (unidades)} \end{cases} \tag{3.22}$$

El efecto de reducción se aplica a EGP_0 por un período definido, seguido de una restauración gradual a los valores normales.

3.4.2.5. Estado anímico y enfermedades

Los factores como el estrés y las enfermedades (p. ej., infecciones, fiebre) modifican la producción de glucosa y la sensibilidad a la insulina. Para modelar estos eventos, se utilizan factores de ajuste:

- **Estrés**: Aumenta EGP_0 y reduce SIE.
- **Relajación**: Incrementa SIT y SIE.
- **Infección**: Aumenta EGP_0 y reduce tanto SIT como SID.
- **Fiebre**: Tiene un mayor impacto que una infección, aumentando EGP_0 y reduciendo significativamente SIT y SID.

3.4.3. Solver y simulación del modelo

El modelo se resuelve utilizando un método numérico basado en el método de Runge-Kutta, que permite una integración precisa y eficiente de las ecuaciones diferenciales que rigen la dinámica de la glucosa en sangre. La simulación se ejecuta en Python, donde cada ciclo de integración representa un avance en el tiempo y una actualización de los niveles de glucosa y de insulina, en función de los parámetros fisiológicos y de los eventos externos definidos en la base de datos.

Para esta tarea se ha desarrollado un método numérico adaptativo explícito, denominado `SolverRK1_2`, que combina los métodos de Runge-Kutta de orden 1 (RK1, también conocido como método de Euler) y de orden 2 (RK2, también conocido como método de Heun). Este enfoque permite ajustar dinámicamente el tamaño del paso de integración en función de una estimación del error relativo entre ambos métodos, optimizando así la precisión y la eficiencia computacional.

Naturaleza del método numérico

El método utilizado es un **método numérico explícito** porque el cálculo del siguiente estado del sistema x_{n+1} se realiza directamente a partir de evaluaciones conocidas de la función derivada $f(t, x)$, sin necesidad de resolver sistemas de ecuaciones en cada paso (como ocurriría en métodos implícitos).

En cada iteración, el método numérico calcula dos aproximaciones del estado futuro:

- $x_{n+1}^{(1)}$ mediante el método de Euler: $x + h \cdot f(t, x)$.
- $x_{n+1}^{(2)}$ mediante el método de Heun: $x + \frac{h}{2}\left[f(t, x) + f(t + h, x + hf(t, x))\right]$.

La diferencia entre estas dos aproximaciones proporciona una estimación del error local, que se utiliza para decidir si se acepta el paso y para ajustar el siguiente tamaño de paso h de forma adaptativa.

3.4.4. Pseudocódigo del algoritmo

El pseudocódigo simplificado del método numérico se muestra en el Pseudocódigo 3.1.

El código completo se encuentra en el apéndice. Un pseudocódigo del código completo es el mostrado en los Pseudocódigos 3.3 y 3.2.

Pseudocódigo 3.1 Solver adaptativo RK1-RK2 (Heun-Euler).

```
Entrada: t0, tf, x0, tolerancia eps, paso inicial h

t ← t0
x ← x0

Mientras t <tf hacer:
    h ← min(h, tf - t)
    /* estimación de Euler (RK1) */
    k1 ← h * f(t, x)
    /* corrección de Heun (RK2) */
    k2 ← h * f(t + h, x + k1)
    /* error relativo estimado */
    e ← ||(k2 - k1)/2|| / ||x||
    Si e >eps entonces:
        h ← h / 2
        continuar
    x ← x + (k1 + k2)/2
    t ← t + h
    h ← max(1, min(60, h * min(2, 0.9 * sqrt(eps / (2e)))))
Fin mientras
```

Pseudocódigo 3.2 Solver adaptativo RK1-RK2 (Heun-Euler) implementado.

```
Clase SolverRK1_2:

    Función init(default_time_step, tol):
        self._time_step ← default_time_step
        self._tol ← tol

    Función reset(default_time_step):
        self._time_step ← default_time_step

    Función solve(derivatives, t_init, x_init, t_final):
        t ← t_init
        x ← x_init
        Mientras t <t_final hacer:
            Mientras Verdadero:
                t_next ← min(t + time_step, t_final)
                dt ← (t_next - t) en minutos
                /* realiza un paso RK1-RK2 */
                step ← perform_time_step(derivatives, t, x, dt)
                Si step['e'] >tol entonces:
                    time_step ← time_step / 2
                Sino:
                    x ← step['x']
                    t ← t_next
                Si step['e'] == 0 entonces:
                    step['e'] ← 1e-10
                    /* ajusta el siguiente paso según el error */
                    time_step ← max(1, min(60, floor(time_step *
                            min(2, 0.9 * sqrt(tol / 2 / step['e'])))))
                    romper
        retornar x

    Función perform_time_step(deriv, t, x, dt):
        k1 ← times_scalar(deriv(t, x), dt)
        k2 ← times_scalar(deriv(t + dt, vector_sum(x, k1)), dt)
        e ← norm(vector_sum(times_scalar(k1, -0.5), times_scalar(k2,
0.5)))
        rel_err ← e / norm(x)
        retornar diccionario con:
            'x': vector_sum(x, times_scalar(k1, 0.5), times_scalar(k2,
0.5))
            'e': rel_err
```

Pseudocódigo 3.3 Funciones auxiliares para el solver adaptativo RK1-RK2 (Heun-Euler) implementado.

```
Función norm(x):
    /* calcula la norma euclídea de un vector */
    retornar raíz_cuadrada(suma de value^2 para cada value en x)

Función times_scalar(vec, scalar):
    /* multiplica cada componente de un vector por un escalar */
    retornar  nuevo vector con value * scalar para cada key, value en
vec

Función vector_sum(v1, v2, ...):
    /* suma varios vectores */
    inicializar result como diccionario vacío
    Para cada vec en vectores:
        Para cada key, value en vec:
            result[key] += value
    retornar result
```

3.5. Visualización de resultados, ejemplos de uso y validación

Para facilitar el análisis del modelo, en la herramienta DiabetesTool, se ha implementado una interfaz gráfica que permite visualizar la evolución de la glucosa en sangre, la tasa de infusión de insulina y otros parámetros relevantes en tiempo real. Esta visualización proporciona al usuario una comprensión clara del comportamiento del sistema ante diferentes condiciones y permite evaluar la efectividad del modelo de simulación en la administración de insulina personalizada.

Los siguientes módulos de DiabetesTool se pueden configurar para modelizar un paciente con diabetes:

- Static Simulator: Se trata de una simulación de visualización estática, a la cual se le introduce una fecha inicial y una duración de la simulación. Al ejecutar este módulo, se generan varias gráficas:
  - Evolución de la glucosa.
  - Evolución de las variables de estado Q1, Q2, S1, S2, I, X1, X2, X3, D1 y D2.
  - Evolución de variables intermedias.
- Dynamic Simulator: Se trata de una simulación de visualización dinámica. Una de las opciones para ejecutar una simulación dinámica es la obtención de datos de glucosa del modelo de diabetes y, la otra posible fuente de datos de glucosa son los datos en tiempo real de la bomba de Medtronic 780g. Los inputs son los mismos que para Static Simulator, además de que se le puede configurar la velocidad de simulación.

El Apéndice A contiene el Manual de usuario de DiabetesTool, donde se explica cómo se pueden configurar los módulos y cómo se pueden ejecutar las simulaciones.

Los siguientes análisis se han realizado con Static Simulator. Cada simulación toma de entrada también 3 ficheros, los cuales si no se indica el nombre de estos, toma los de referencia:

1. Inputs (el archivo de referencia es data/inputs/inputs.json): contiene los datos de comidas, bolos de insulina y ratios de basales.

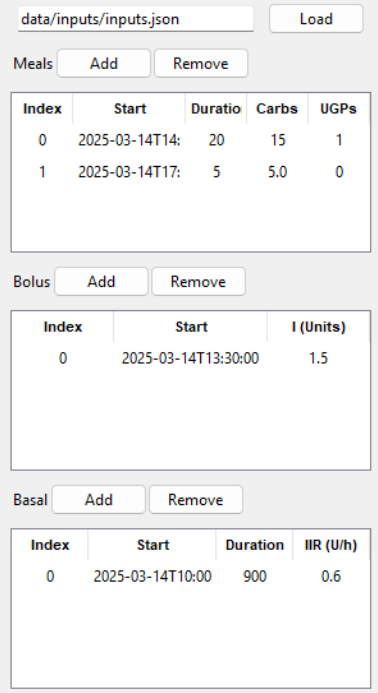


Figura 3.2: Visualización de Inputs en Static Simulator.

2. Events (el archivo de referencia es data/events/events.json): contiene los eventos que pueden ocurrir.
3. Params (el archivo de referencia es data/params/params.json): contiene los parámetros personales del paciente.

Si no se indica lo contrario, en las simulaciones siguientes se muestra la evolución de glucosa para un paciente que presenta los parámetros recogidos en 3.4.

3.5.1. Ingesta de alimentos con bolo de insulina

Se muestran a continuación una serie de ejecuciones en las que el paciente ingiere una cantidad de carbohidratos y se inyecta insulina, variando las cantidades y las horas.

3.5.1.1. Ejemplo inicial

Se parte de un paciente con diabetes tipo 1 cuyos parámetros fisiológicos se muestran en la Figura 3.4, y que presenta una tasa basal constante de 0.6 U/h. En las bombas de insulina, este

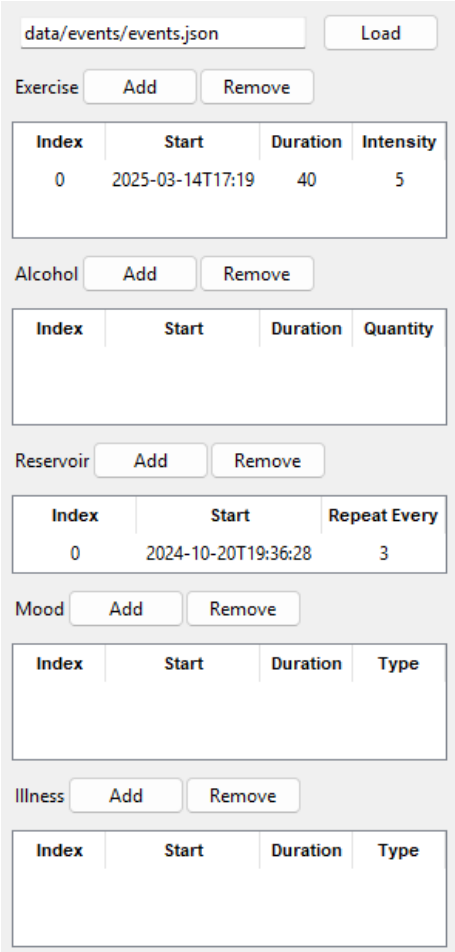


Figura 3.3: Visualización de Events en Static Simulator.

valor puede configurarse directamente especificando la tasa deseada para cada franja horaria. En el caso de pacientes que utilizan plumas de insulina, esta tasa depende del tipo de insulina basal (de acción prolongada). Por ejemplo, si se administra una insulina con efecto aproximado de 24 horas, el paciente suele inyectarse una única dosis diaria a la misma hora. Una dosis de 15 U diarias equivaldría, por tanto, a una tasa de 15 U / 24 h = 0.625 U/h.

En este escenario, el paciente se administra un bolo de 4 U de insulina a las 13:30:00 h en previsión de una comida compuesta por 40 g de hidratos de carbono (HC) y 1 unidad grasa-proteína (UGP), como podría ser una porción de pizza. Este alimento contiene los 40 g de HC en la masa, y proteínas y grasas en el queso y embutidos, lo que justifica la presencia de 1 UGP. Posteriormente, a las 17:00:00 h, el paciente consume un snack de 5 g de HC, como una onza de chocolate. Debido a la baja cantidad de hidratos, decide no administrarse insulina para esta ingesta.

En la Figura 3.5 se observa cómo la insulina comienza a hacer efecto antes del inicio de la comida principal. Al ingerir la pizza, la glucosa en sangre se eleva ligeramente, pero sin llegar a alcanzar valores de hiperglucemia, ya que el bolo de 4 U administrado previamente compensa eficazmente el impacto glucémico del alimento.

En 3.6 se puede ver la evolución de las variables de estado para este caso.

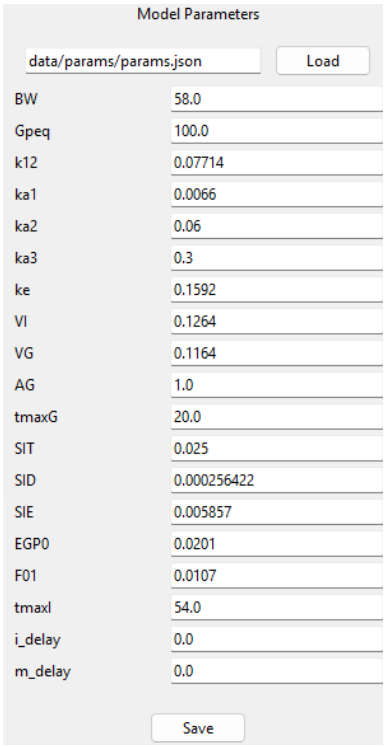


Figura 3.4: Visualización de Parameters en Static Simulator.

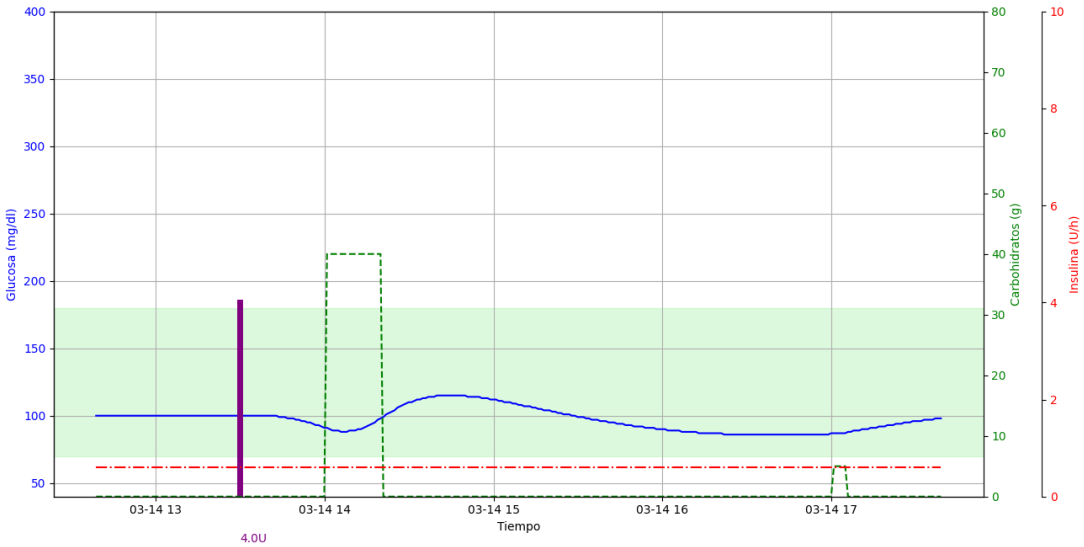


Figura 3.5: Evolución de glucosa Ejemplo Inicial.

3.5.1.2. Retardos en la acción de insulina y en la asimilación de la glucosa

Se partía de unos parámetros del paciente en el que los retardos en la acción de insulina y en la asimilación de la glucosa eran igual a 0 min. En la vida real, el retardo en la acción de la insulina existe y depende de la marca de la insulina que se inyecte. Para los alimentos, suele depender del tipo de alimento y de las condiciones de la persona en ese momento, pero lo normal es que también exista.

Se muestra ahora el ejemplo anterior pero incluyendo estos retardos para hacerlo más realista.

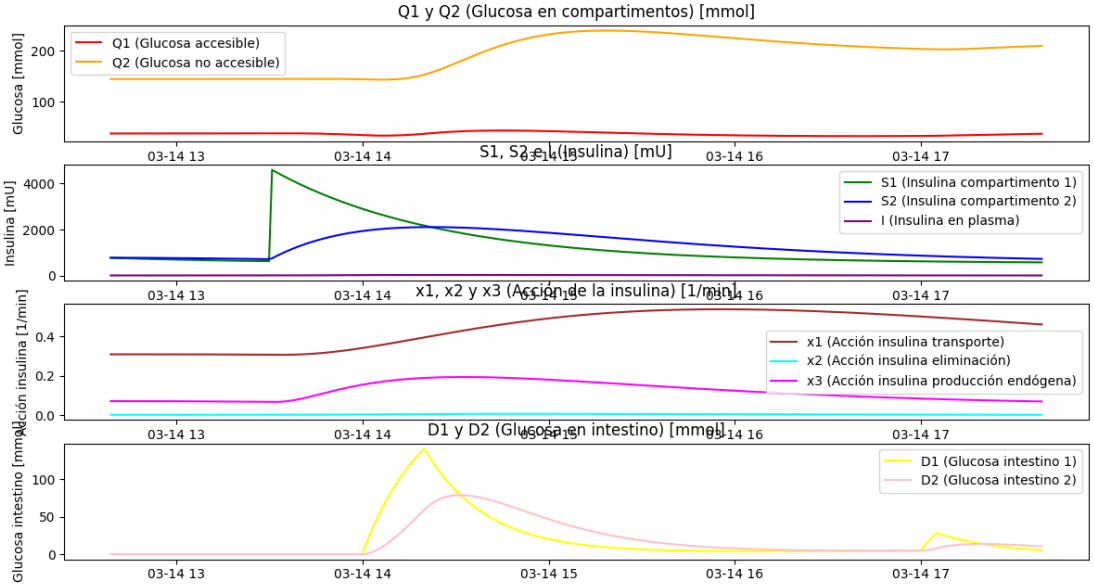


Figura 3.6: Evolución de variables de estado Ejemplo Inicial.

- Retardo en acción de la insulina de 20 minutos: Figura 3.7.

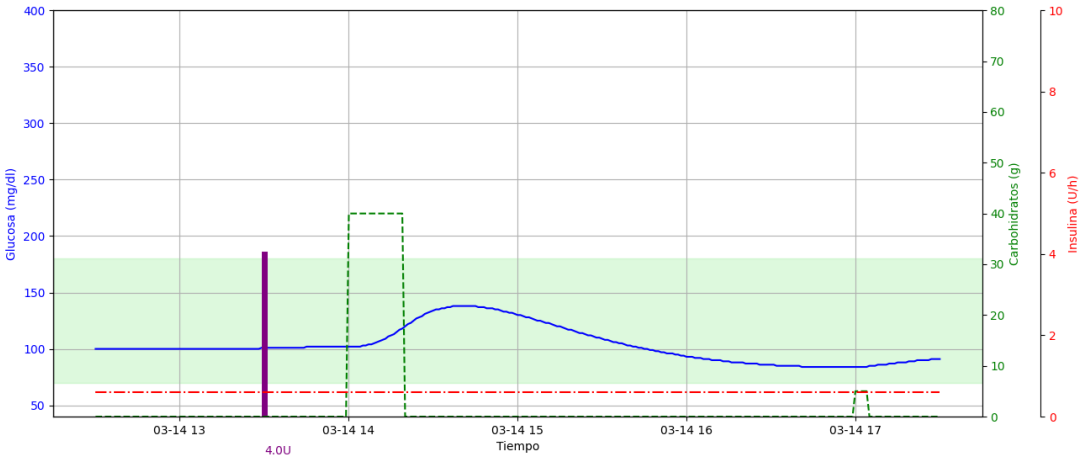


Figura 3.7: Retardo en acción de la insulina de 20 minutos.

- Retardo en acción de la comida de 10 minutos: Figura 3.8.
- Retardos juntos: Figura 3.9.

3.5.1.3. Hipoglucemias

Partiendo del último caso, en el que se implementan los retardos en la acción de comida y de la insulina, se estudia bajo qué efectos se puede llegar a estar en hipoglucemia.

- Más insulina de la necesaria: para los 40 g de HC, antes se inyectaban 4 U, lo que se correspondería a un ratio de 10 g/U. Para ese caso la respuesta de la glucosa era una

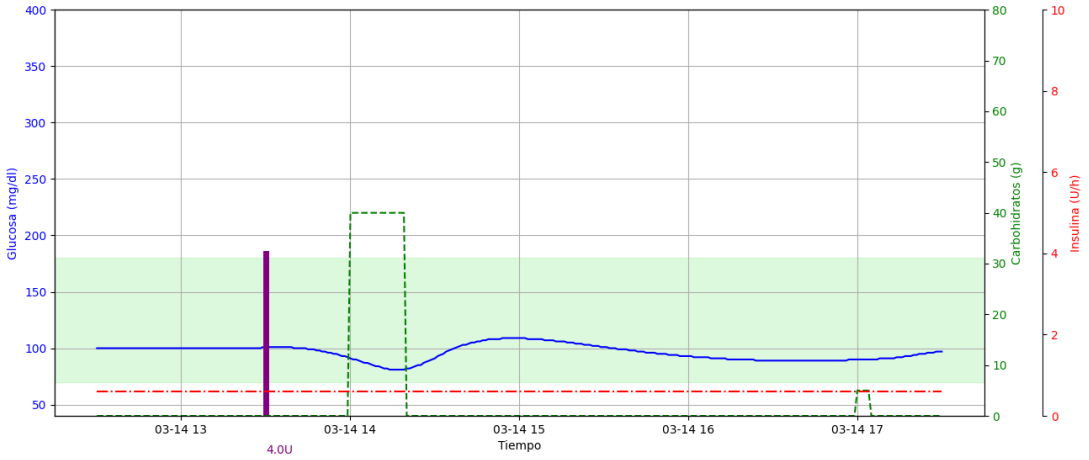


Figura 3.8: Retardo en acción de la comida de 10 minutos.

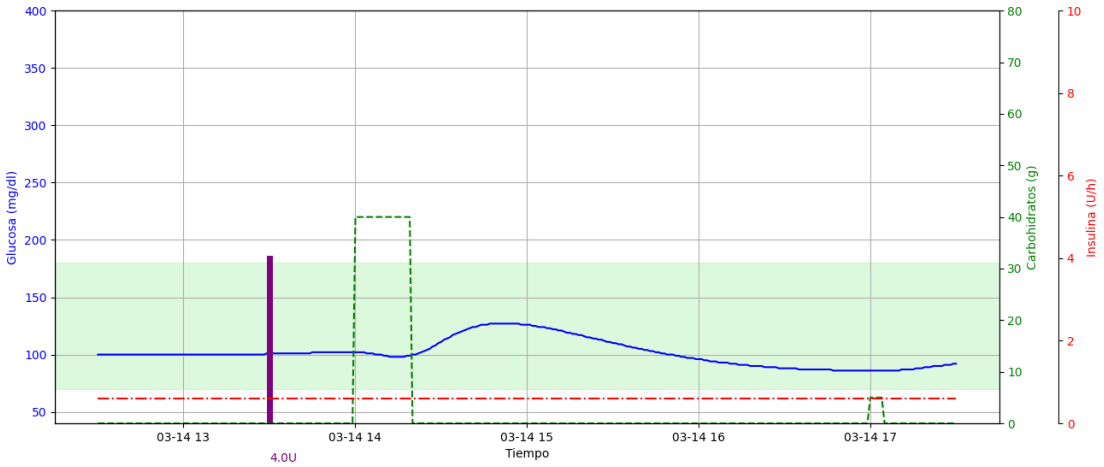


Figura 3.9: Retardo en acción de la insulina de 20 minutos y en la comida de 10 minutos.

normoglucemia, por lo que el ratio se podía considerar correcto. Si bajo este ratio se inyecta más insulina, finalmente se llega a la hipoglucemia. Ahora se han inyectado 5,5 U. La evolución de la glucosa puede verse en 3.10.

- Ingesta de menos HC con la misma insulina (análogo al caso anterior): ahora, en vez de comer 40 g de HC, se ingieren 25 g y se inyectan 4 U. Como se observa, la tendencia es hacia la hipoglucemia aunque no llega a estar por debajo de 70 mg/dL. La evolución de la glucosa puede verse en 3.11.

- Demasiado tiempo de espera: los inputs son los mismos que en el ejemplo inicial, 40 g de HC y 4 U. Pero ahora en vez de empezar a comer a las 14:00:00, empieza 40 minutos más tarde. Además, tarda en comer el doble de tiempo. Como la insulina activa en el cuerpo ya está haciendo efecto, la tendencia es a descender la glucosa antes de que empiece a subir por los alimentos. Normalmente el paciente hace tiempos de espera para comer, pero si estos tiempos de espera son demasiado grandes, puede llevar a hipoglucemia. Bajo ciertos casos no se recomienda hacer estos tiempos de espera. La evolución de la glucosa puede verse en 3.12.

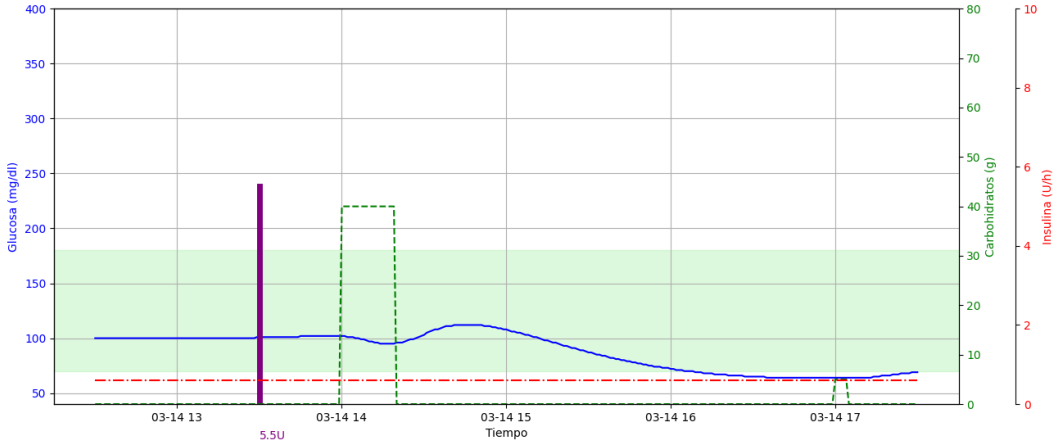


Figura 3.10: Hipoglucemia por más insulina.

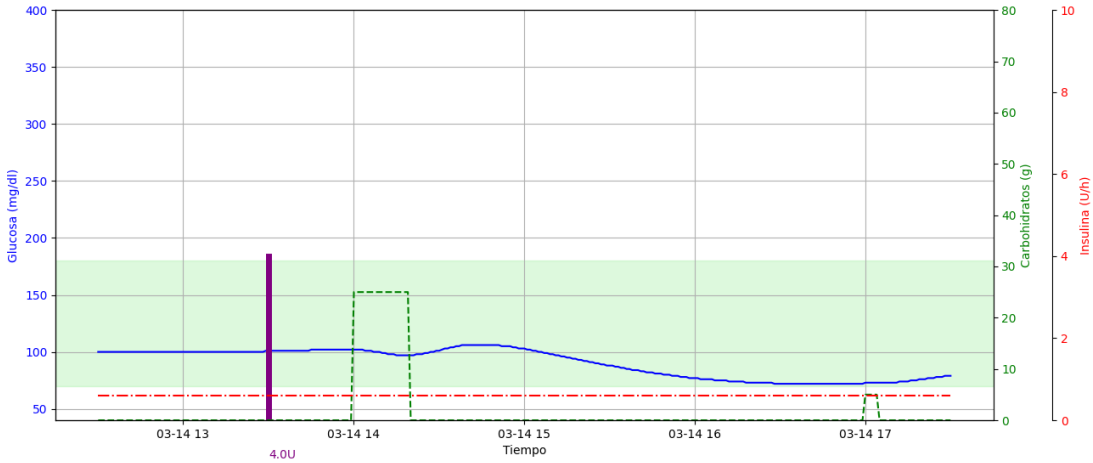


Figura 3.11: Hipoglucemia por menos HC.

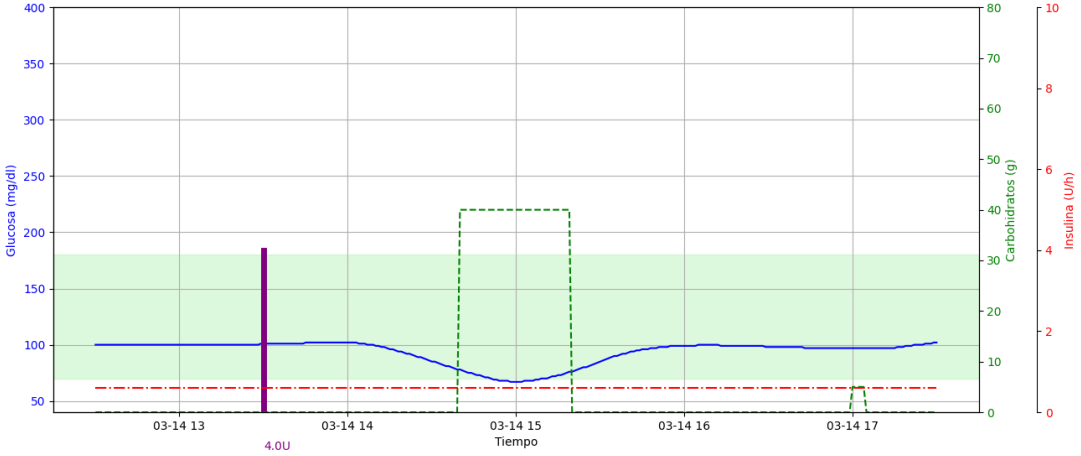


Figura 3.12: Hipoglucemia por demasiado tiempo de espera.

- Demasiado basal en el cuerpo: ahora el input que varía es el ratio de basal. Antes estaba en 0.6 U/h (que se corresponde con 14,4 U al día), y ahora se ha modificado a 0.9 U/h (21.6 U al día). Basales más altas de las que se corresponden al paciente pueden resultar

en glucemias muy buenas tras las comidas, pero en hipoglucemias pasadas unas horas. Es muy importante que el paciente siga las pautas de su endocrino para ajustar sus basales, con cambios muy progresivos en estos ajustes (por ejemplo modificar la basal en 1 unidad cada 4 días como mínimo ya que el efecto no es inmediato). La evolución de la glucosa puede verse en 3.13.

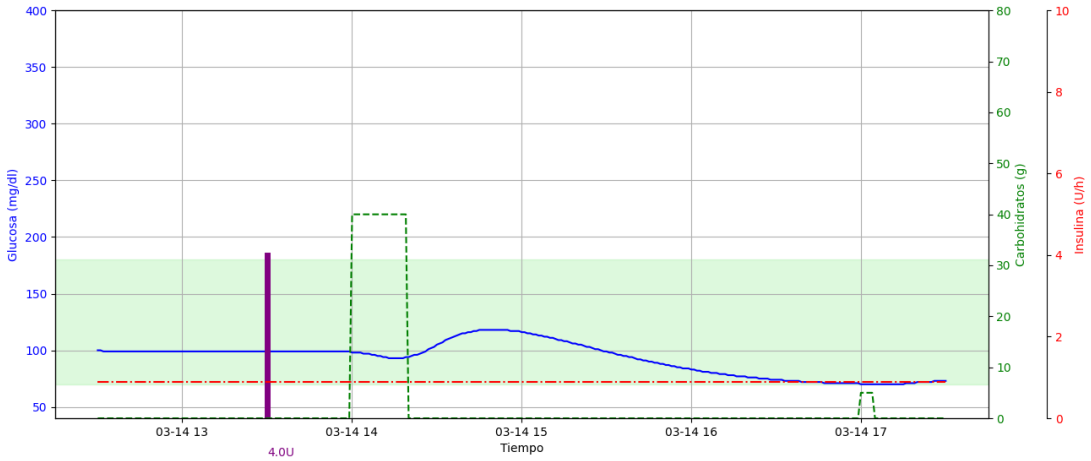


Figura 3.13: Hipoglucemia por demasiada basal.

3.5.1.4. Hiperglucemias

Ahora, partiendo del caso de referencia, en el que se implementan los retardos en la acción de comida y de la insulina, se estudia bajo qué efectos se puede llegar a estar en hiperglucemia.

- Menos insulina de la necesaria: para los 40 g de HC, antes se inyectaban 4 U, lo que se correspondería a un ratio de 10 g/U. Para ese caso la respuesta de la glucosa era una normoglucemia, por lo que el ratio se podía considerar correcto. Si bajo este ratio se inyecta menos insulina, finalmente se llega a la hiperglucemia. Se han inyectado 1,5 U. La evolución de la glucosa puede verse en 3.14.

- Ingesta de más HC con la misma insulina (análogo al caso anterior): ahora, en vez de comer 40 g de HC, se ingieren 75 g y se inyectan 4 U. La evolución de la glucosa puede verse en 3.15.

- Bolo de insulina después de comer: los inputs son los mismos que en el ejemplo inicial, 40 g de HC y 4 U. Pero ahora en vez de empezar a comer a las 14:00:00, come a las 13:00:00 y se pincha insulina 30 minutos después. La evolución de la glucosa puede verse en 3.16.

- Insuficiente basal en el cuerpo: ahora el input que varía es el ratio de basal. Antes estaba en 0.6 U/h (que se corresponde con 14,4 U al día), y ahora se ha modificado a 0.2 U/h (4.8 U al día). Se ha ampliado el tiempo de la simulación en 100 minutos para poder ver como poco a poco la glucosa del paciente va aumentando ya que no presenta suficiente basal en el cuerpo. La evolución de la glucosa puede verse en 3.17.

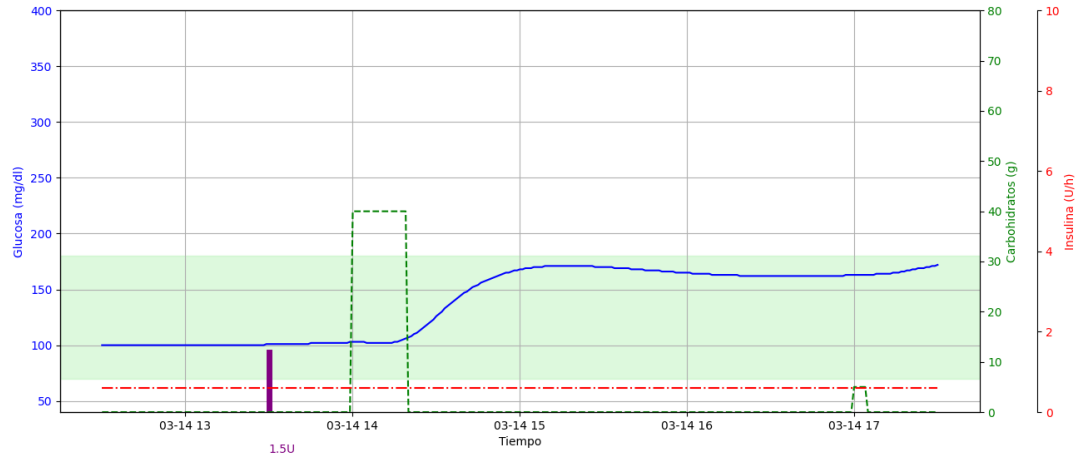


Figura 3.14: Hiperglucemia por menos insulina.

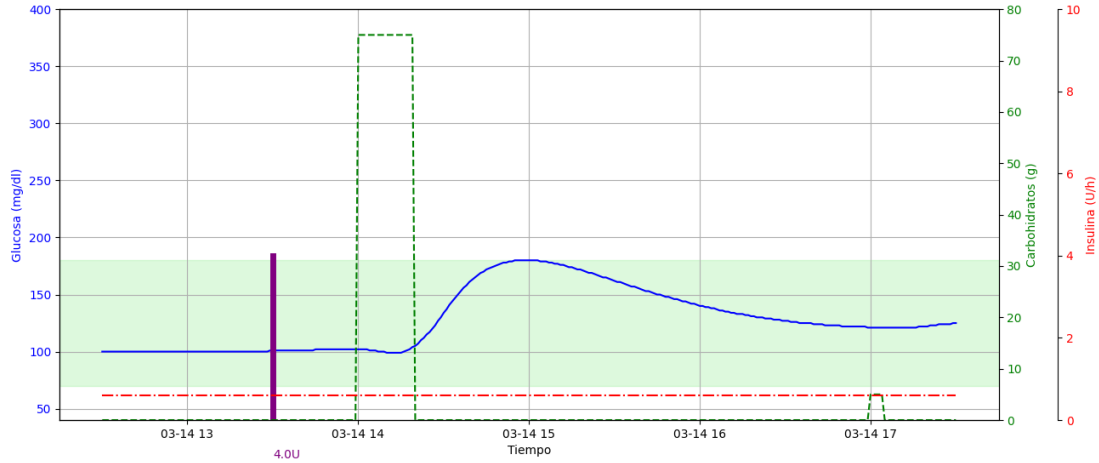


Figura 3.15: Hiperglucemia por más HC.

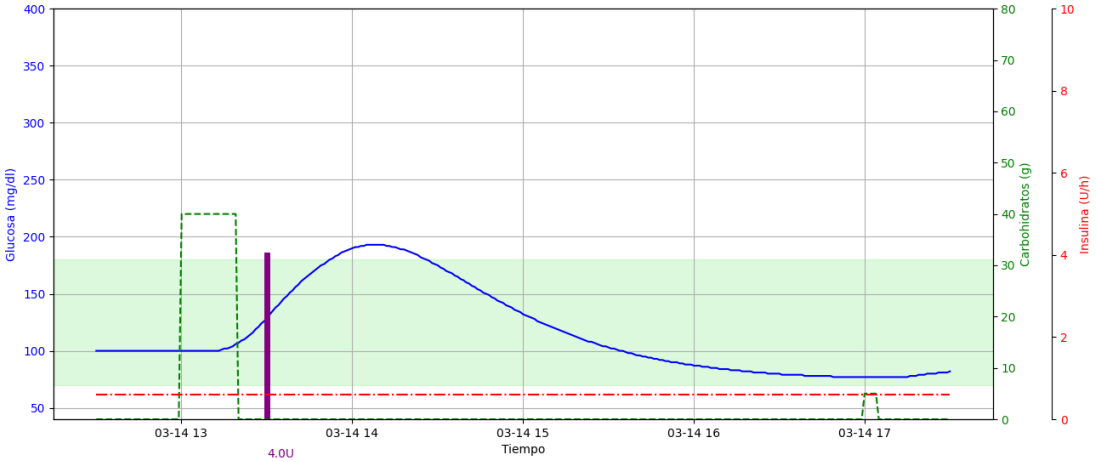


Figura 3.16: Hiperglucemia por comida antes que bolo.

3.5.1.5. Efecto de las UGPs

Ahora se muestran varios ejemplos de comida que presenta la misma cantidad de hidratos de carbono (40g de HC), con variaciones en las cantidades de UGPs. Se inyecta un bolo de 4U un

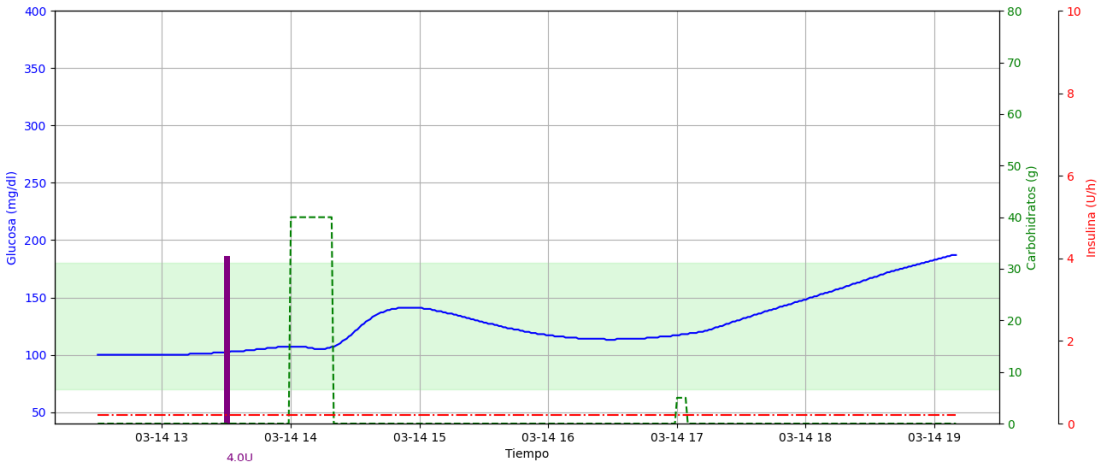


Figura 3.17: Hiperglucemia por insuficiente basal.

poco antes de ingerir el alimento. Los parámetros del paciente son los mismos, pero ahora la insulina basal administrada es de 0.7 U/h.

- 0 UGPs: puede verse la evolución en 3.18

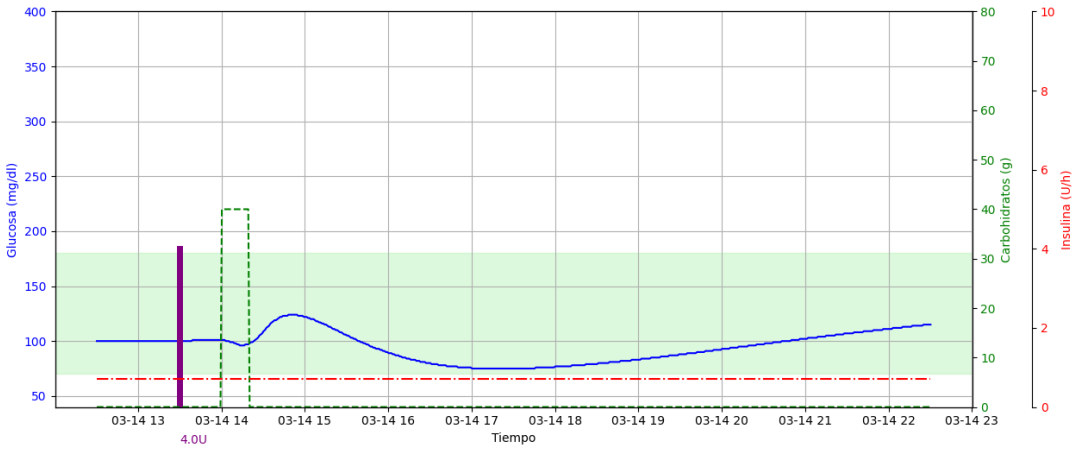


Figura 3.18: Comida sin UGPs.

- 2 UGPs: puede verse la evolución en 3.19. La evolución es bastante similar que sin UGPs, aunque ya se puede intuir como la glucosa va aumentando poco a poco.

- 5 UGPs: puede verse la evolución en 3.20. Pasadas unas horas, la glucosa aumenta considerablemente aun no consumiendo ningún alimento.

  Un alimento que podría contener 40 gramos de hidratos de carbono y 5 UGPs sería una hamburguesa con pan, queso, bacon y salsas. Esto se debe a que el pan de la hamburguesa aportaría alrededor de 40 g de hidratos de carbono (dos rebanadas de pan de hamburguesa pueden contener aproximadamente esa cantidad). Por otro lado, las 5 UGPs provendrían del alto contenido de grasas y proteínas, principalmente del queso, el bacon, la carne de la hamburguesa y las salsas (mayonesa, por ejemplo).

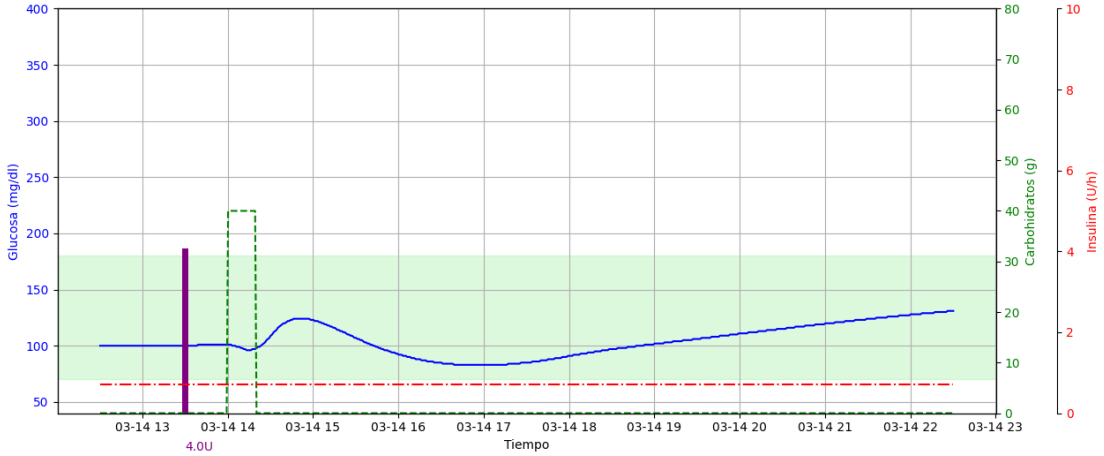


Figura 3.19: Comida con 2 UGPs.

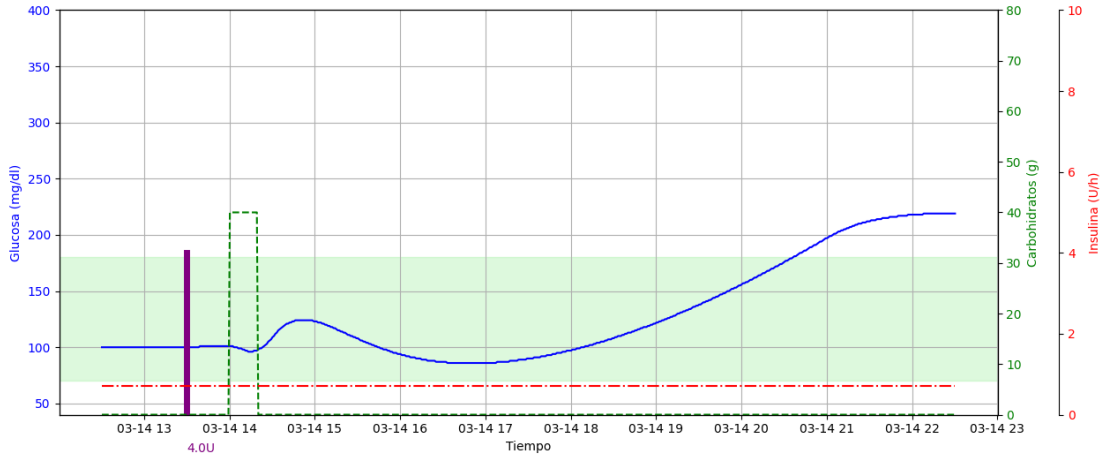


Figura 3.20: Comida con 5 UGPs.

Para mantener la glucosa en valores más bajos, hay que aplicar correcciones. En 3.21 y 3.22 se ve como se aplican 1.5 U a las 17:30 h en ambos casos, y otro bolo de 1.5 U a las 20:00h en el último caso.

En la Figura 3.23 se muestran las variables de estado de este último caso.

3.5.2. Efecto de los eventos

A continuación se muestra el efecto de los distintos eventos sobre la glucosa y las variables intermedias utilizadas en el modelo. La imagen 3.24 muestra la evolución de la glucosa sin eventos, tan solo con insulina basal.

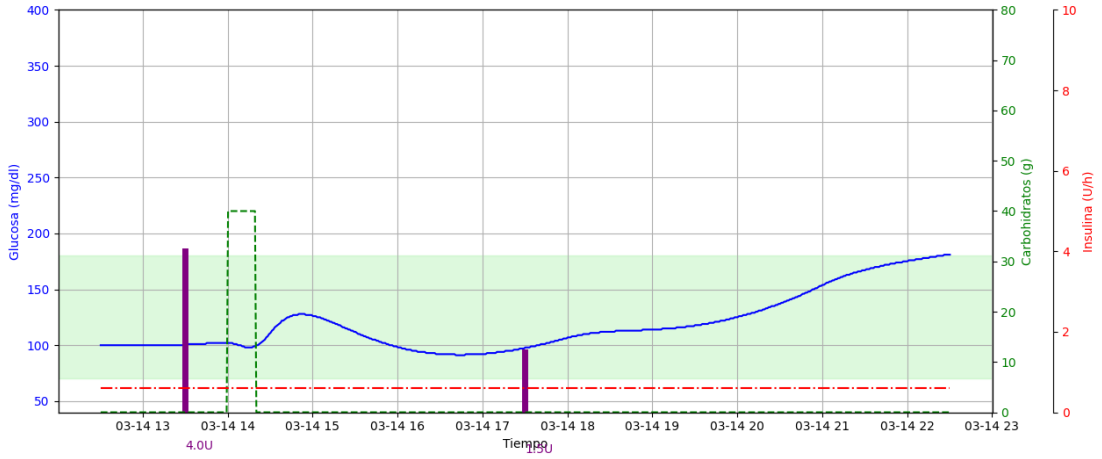


Figura 3.21: Comida con 5 UGPs más 1 bolo de corrección.

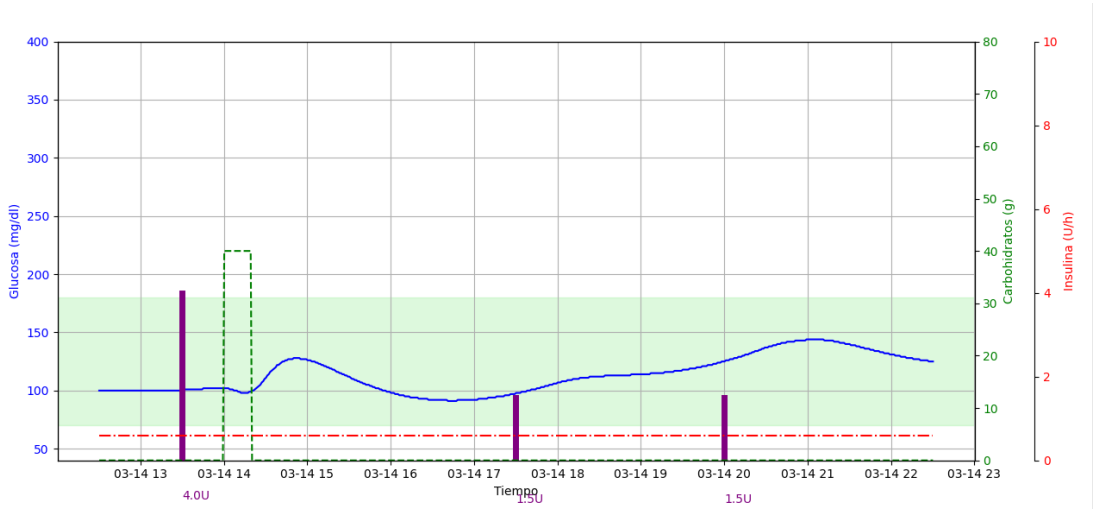


Figura 3.22: Comida con 5 UGPs más 2 bolo de corrección.

3.5.2.1. Efecto del ejercicio físico

En la Figura 3.25 se muestra el efecto de 1h de ejercicio físico de intensidad 3 sobre 10. Un tipo de ejercicio así podría ser salir a correr a un ritmo suave. Se ve que en lugar de continuar aumentando poco a poco la glucosa, empieza a disminuir lenta y suavemente. Los parámetros SIT y SID son los que aumentan durante el ejercicio y varios minutos después. Esto se puede ver en 3.26.

Para el caso de un ejercicio físico más intenso, de unos 45 minutos de intensidad 7 sobre 10, se tiene lo siguiente. Un ejercicio de estas características podría ser la primera mitad de un partido de fútbol. En 3.27 se ve cómo aumenta considerablemente el azúcar en sangre debido a la adrenalina (intensidad) y la hormona del cortisol. Además, se ve como pasado un tiempo, la tendencia es a disminuir rápidamente la glucosa. En 3.28 se ve como durante el ejercicio de intensidad alta aumenta mucho el parámetro EGP0 debido a esta adrenalina, y que horas después aumenta la sensibilidad a la insulina. Específicamente aumentan SIT y SID.

Una de las recomendaciones antes de realizar ejercicio físico aeróbico es tomar hidratos de

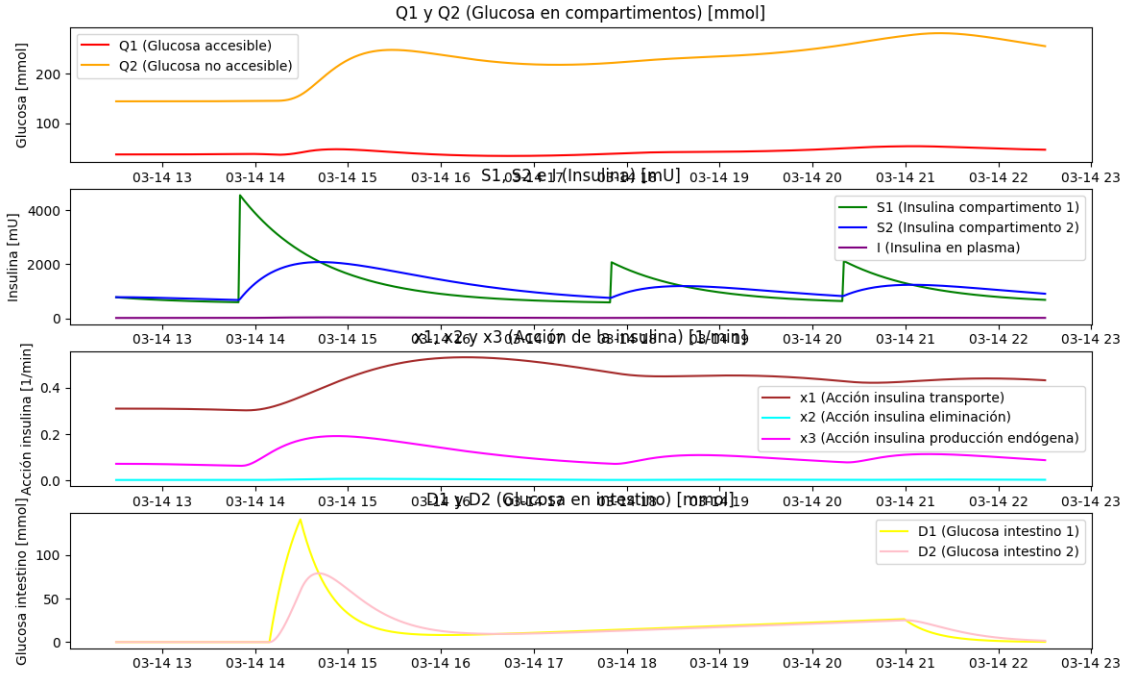


Figura 3.23: Variables de estado para comida con 5 UGPs más 1 bolo de corrección.

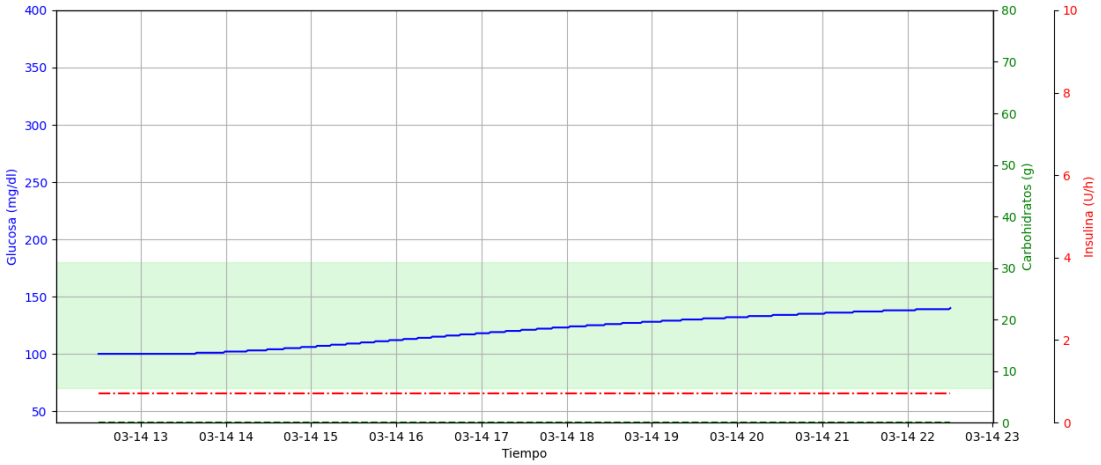


Figura 3.24: Evolución de la glucosa sin eventos.

carbono antes de realizar el ejercicio y cada 30 minutos de esta actividad. Además, es muy importante no tener insulina activa en el cuerpo, ya que al aumentar la sensibilidad a esta, es muy probable que luego de un bajón de azúcar. En 3.29 se aplica otra estrategia, que es tomar hidratos de carbono cuando se pueda percibir que la glucosa está disminuyendo tras realizar deporte. En este caso, se ingieren 20 g de HC.

Otra pauta que indican los endocrinos, es que no se debe corregir una hiperglucemia debida a actividades de ejercicio físico muy intensas, ya que tan pronto la glucosa sube debido a la adrenalina, como luego baja rápidamente debido a la calma posterior y al aumento a la sensibilidad de la insulina. Un aumento grande en la sensibilidad a la insulina puede ser peligroso para casos en los que la insulina basal es constante (como en todos estos ejemplos). La bomba

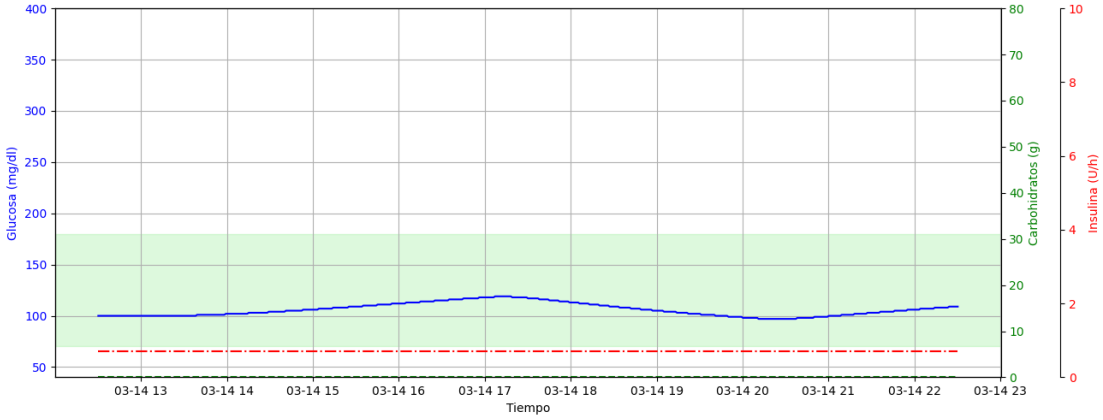


Figura 3.25: Glucosa con ejercicio de intensidad baja.

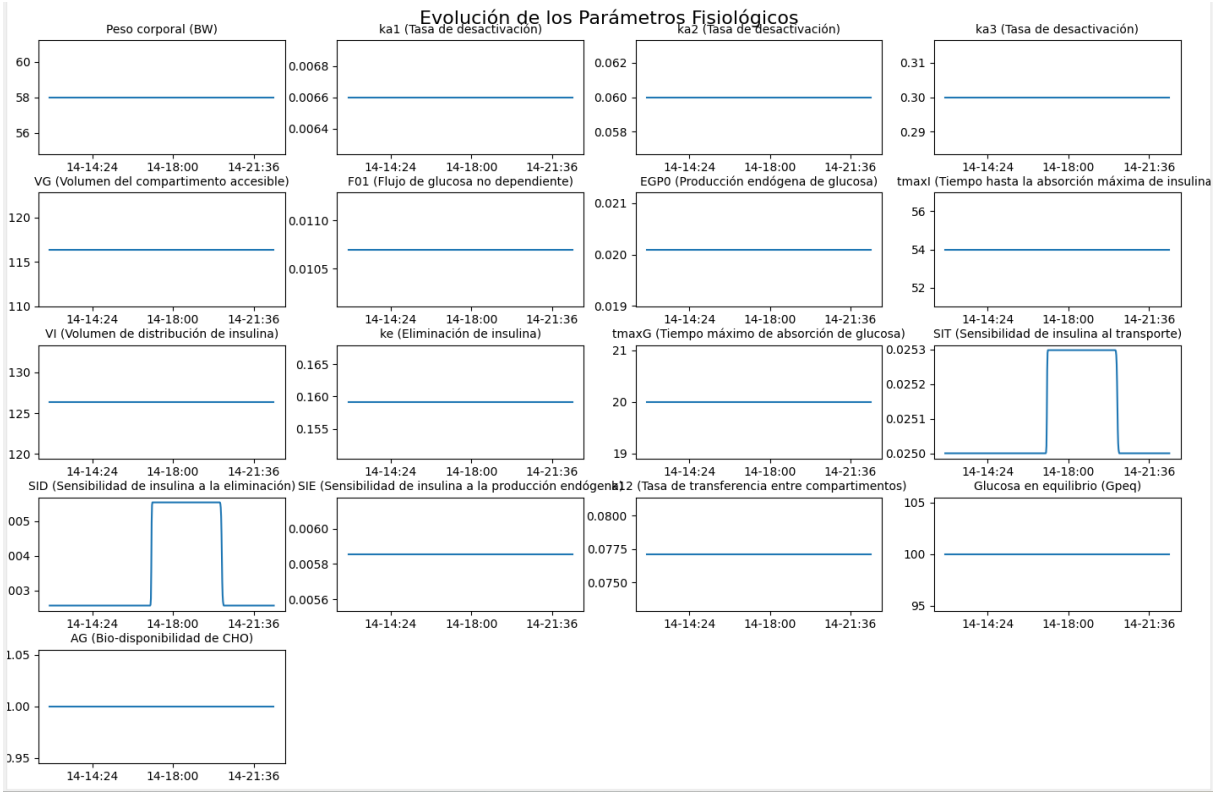


Figura 3.26: Parámetros fisiológicos con ejercicio de intensidad baja.

de insulina de Medtronic 780g puede ser una buena opción frente a esto, ya que la basal sigue un algoritmo que hace que vaya liberando diferentes ratios de insulina basal (incluso no liberar) dependiendo de la glucosa del paciente y de la tendencia de esta.

3.5.2.2. Ingesta de alcohol

Ahora se muestra el efecto del alcohol en la glucosa. En 3.30 se está mostrando la evolución de la glucosa para una ingesta de alcohol de cantidad 6 (lo que se correspondería a 6 copas sin bebida azúcarada), durante 120 minutos, a partir de las 13h. Se observa cómo continúa disminuyendo muy poco a poco la glucosa incluso horas después de haber terminado de consumir alcohol. En

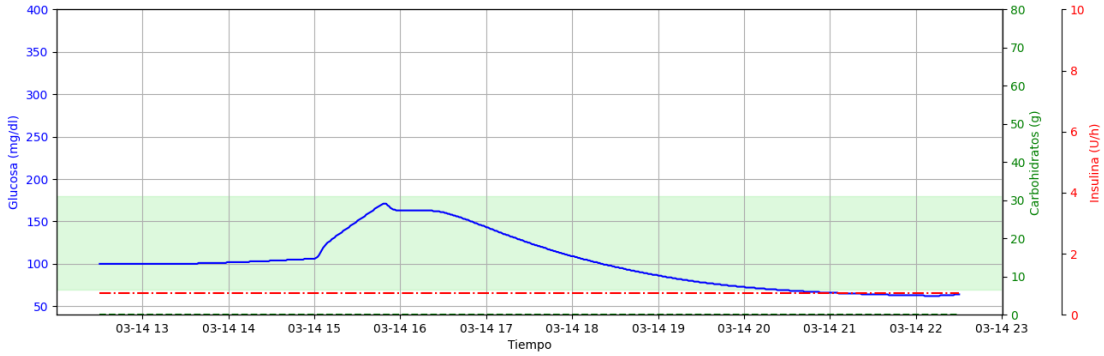


Figura 3.27: Glucosa con ejercicio de intensidad alta.

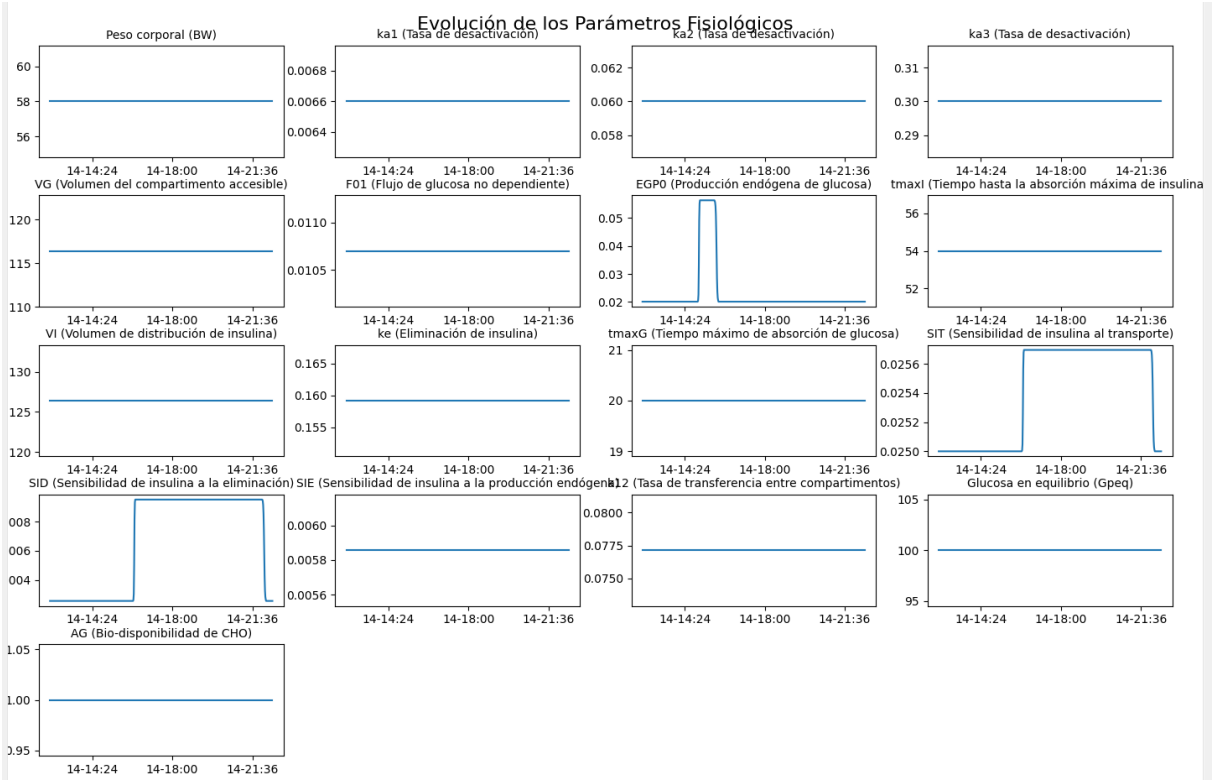


Figura 3.28: Parámetros fisiológicos con ejercicio de intensidad alta.

3.31 se muestran los parámetros fisiológicos con ingesta de alcohol, donde se observa la caída de EGP0, ya que el hígado deja de producir endógenamente glucosa.

Algunos endocrinos recomiendan por lo anterior consumir las bebidas alcohólicas con bebidas con azúcar. La experiencia muestra que no es necesario consumir todas las copas con mezcla con azúcar, si no que se pueden ir alternando. Por ejemplo, en 3.32 se tiene la misma cantidad de alcohol, pero 2 de las copas consumidas con algún refresco con azúcar (12 g de HC cada copa).

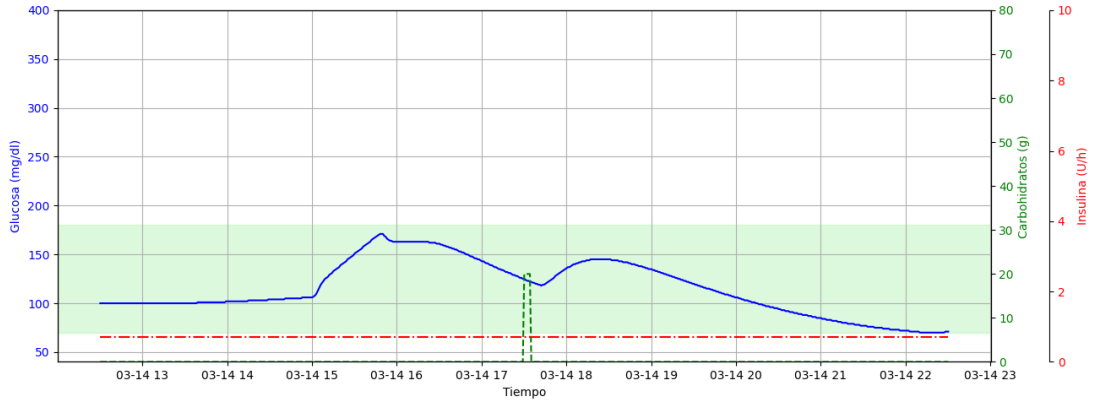


Figura 3.29: Glucosa con ejercicio de intensidad alta e ingesta posterior de hidratos.

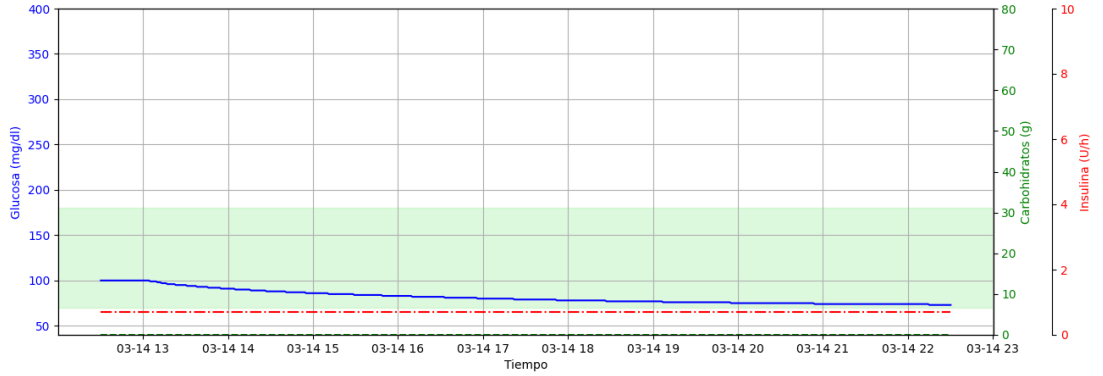


Figura 3.30: Glucosa con ingesta de alcohol.

3.5.2.3. Pérdida de efecto de la insulina por no cambiar el reservorio

Se muestra ahora la evolución de la glucosa para una comida de 50 g de HC, inyectando 10 minutos antes una 4 U de insulina. La fecha de propagación de la glucosa en la simulación es 2025-03-14T12:30:00. En 3.33, el reservorio se cambió el día anterior por la tarde (2025-03-13T19:36:28), por lo que la insulina está haciendo efecto a su máxima eficacia.

En 3.34, el reservorio se cambió hace casi 3 días (2025-03-11T19:36:28). El siguiente día de cambio de reservorio tocaría la tarde del día 14 de marzo, por lo que desde por la mañana hasta el momento de cambio, la insulina ha perdido un poco de efecto. La comparación no es muy evidente, aunque si se puede ver al final de la simulación cómo para este último caso la glucosa es algo mayor.

3.5.2.4. Estado de ánimo y de salud

Se muestran a continuación diferentes estados del paciente y la evolución de la glucosa y parámetros, consumiendo 50 g de HC e inyectando 4 U de insulina 10 min antes.

- Relajación desde las 15 h, durante 1 h (por ejemplo, una siesta). Figuras 3.35 y 3.36.

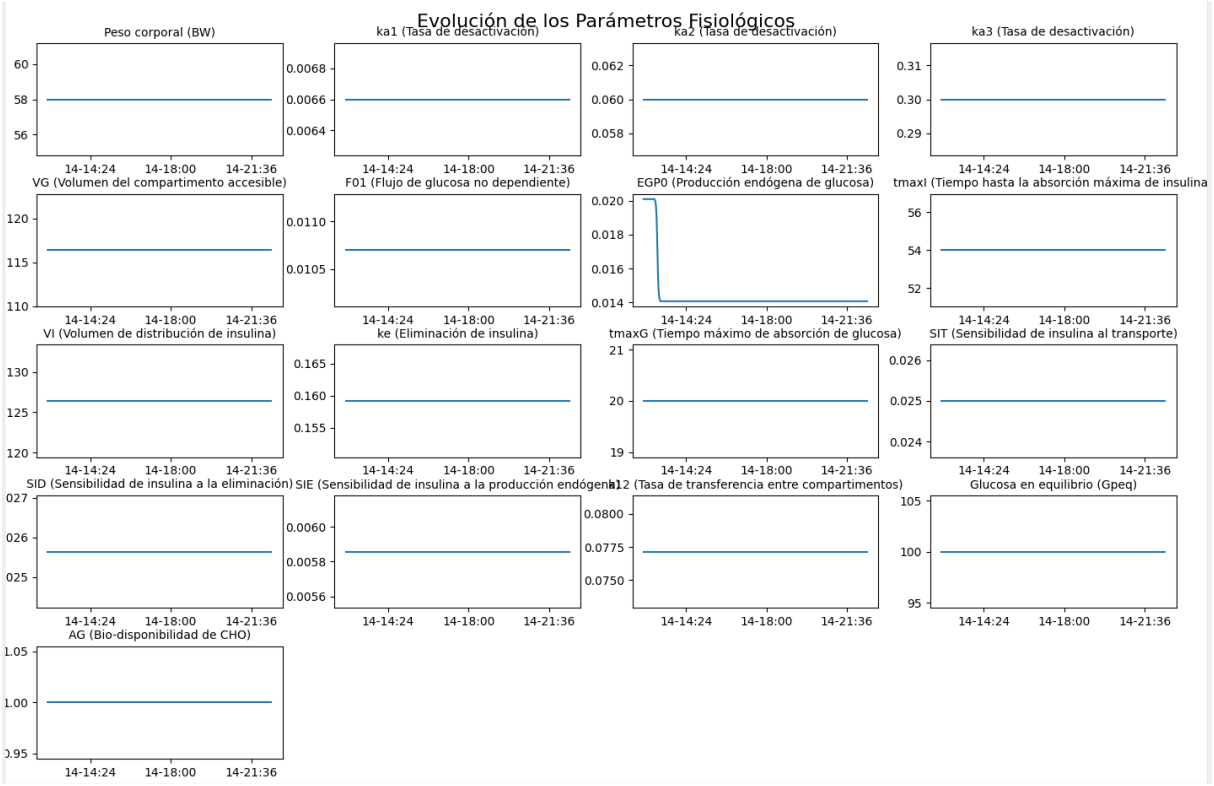


Figura 3.31: Parámetros fisiológicos con ingesta de alcohol.

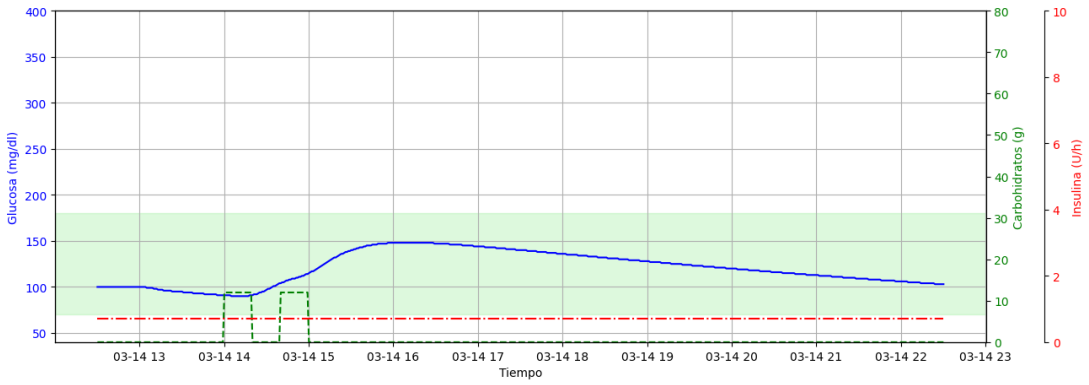


Figura 3.32: Glucosa con ingesta de alcohol y alguna bebida azúcarada.

Aumenta ligeramente la sensibilidad a la insulina. Específicamente los parámetros SIT y SIE.

- Estrés desde las 13 h, durante 10 h. Figuras 3.37 y 3.38. Aumenta EGP0, aumentando así la glucosa desde el inicio del estrés, y disminuye SIE, quedando en hiperglucemia tras comer los 50 g de HC.

- Infección desde las 13 h, durante 10 h. Figuras 3.39 y 3.40. Aumenta EGP0, aumentando así la glucosa desde el inicio, y disminuyen SIT y SID, quedando en hiperglucemia tras comer los 50 g de HC.

- Fiebre desde las 13 h, durante 10 h. Figuras 3.41 y 3.42. Aumenta EGP0, aumentando así la glucosa desde el inicio, y disminuyen SIT y SID, quedando en hiperglucemia tras

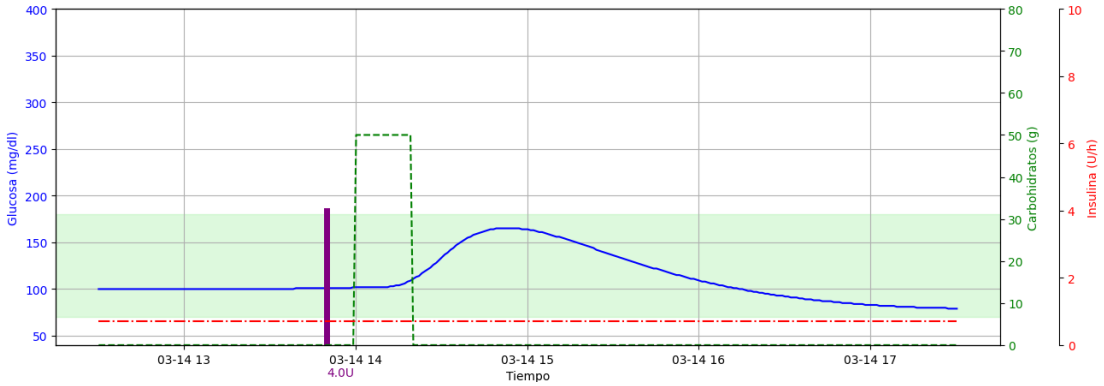


Figura 3.33: Glucosa tras bolo de insulina con pleno efecto.

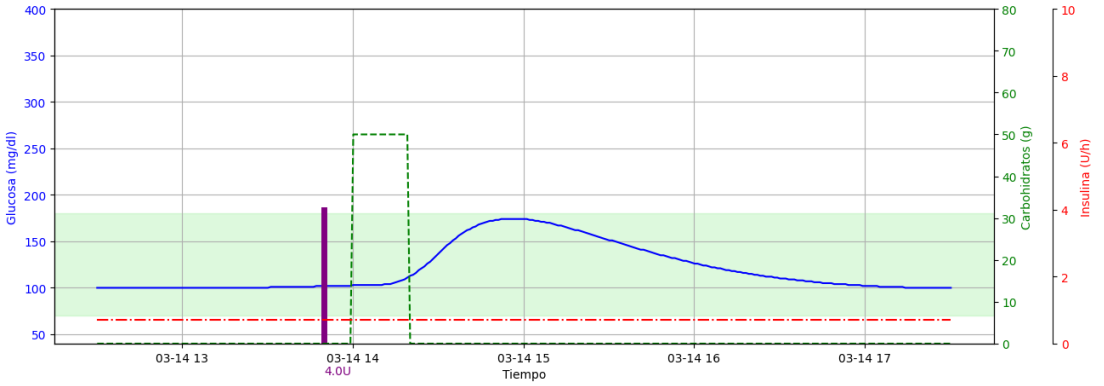


Figura 3.34: Glucosa tras bolo de insulina con menor efecto.

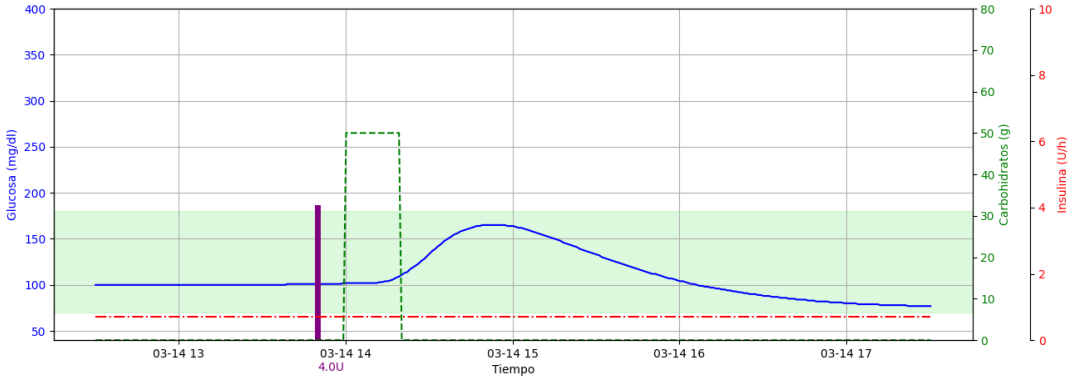


Figura 3.35: Glucosa tras bolo de insulina y estado relajado.

comer los 50 g de HC.

- Deshidratación desde las 13 h, durante 10 h. Figuras 3.43 y 3.44. Disminuye el volumen del compartimento accesible, aumentando ligeramente la glucosa.

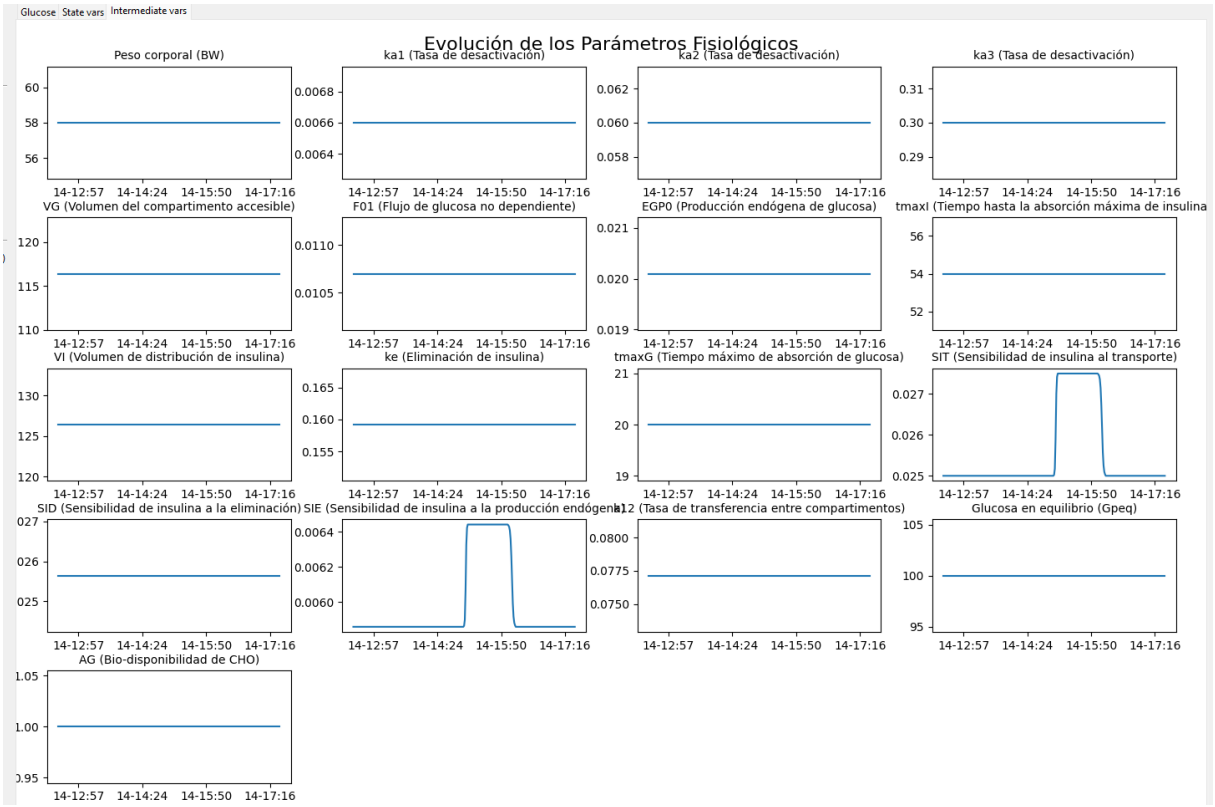


Figura 3.36: Parámetros fisiológicos tras bolo de insulina y estado relajado.

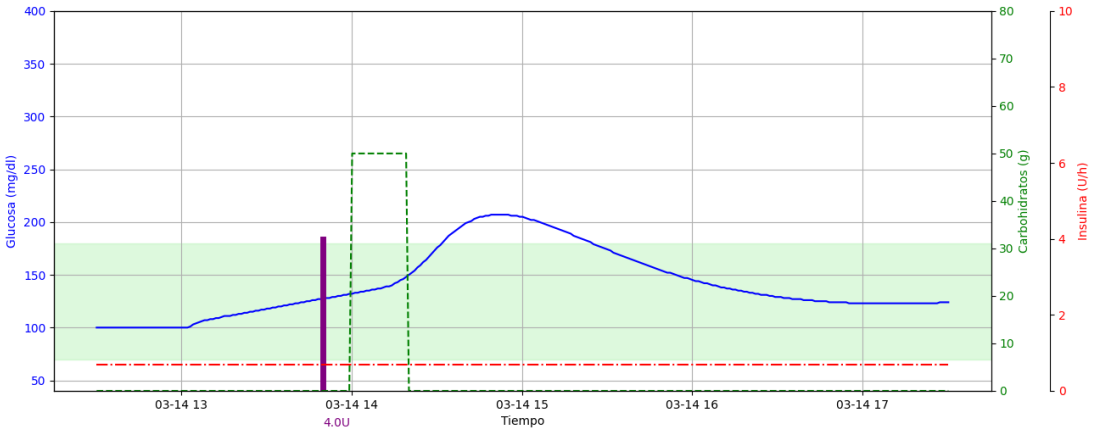


Figura 3.37: Glucosa tras bolo de insulina y estado de estrés.

3.5.3. Cambio en los parámetros del paciente

Una de las funcionalidades de DiabetesTool es realizar simulaciones estáticas variando uno de los parámetros del paciente. Se van a variar los parámetros SIT, SIE, SID y EGP0 para ver como afecta a la evolución.

Nuevamente, se parte de una ingesta de 50 g de HC, 4 U de insulina 30 minutos antes de la ingesta, y una basal del paciente de 0.7 U/h. El resto de parámetros que no se varían son los que se indican en 3.4.

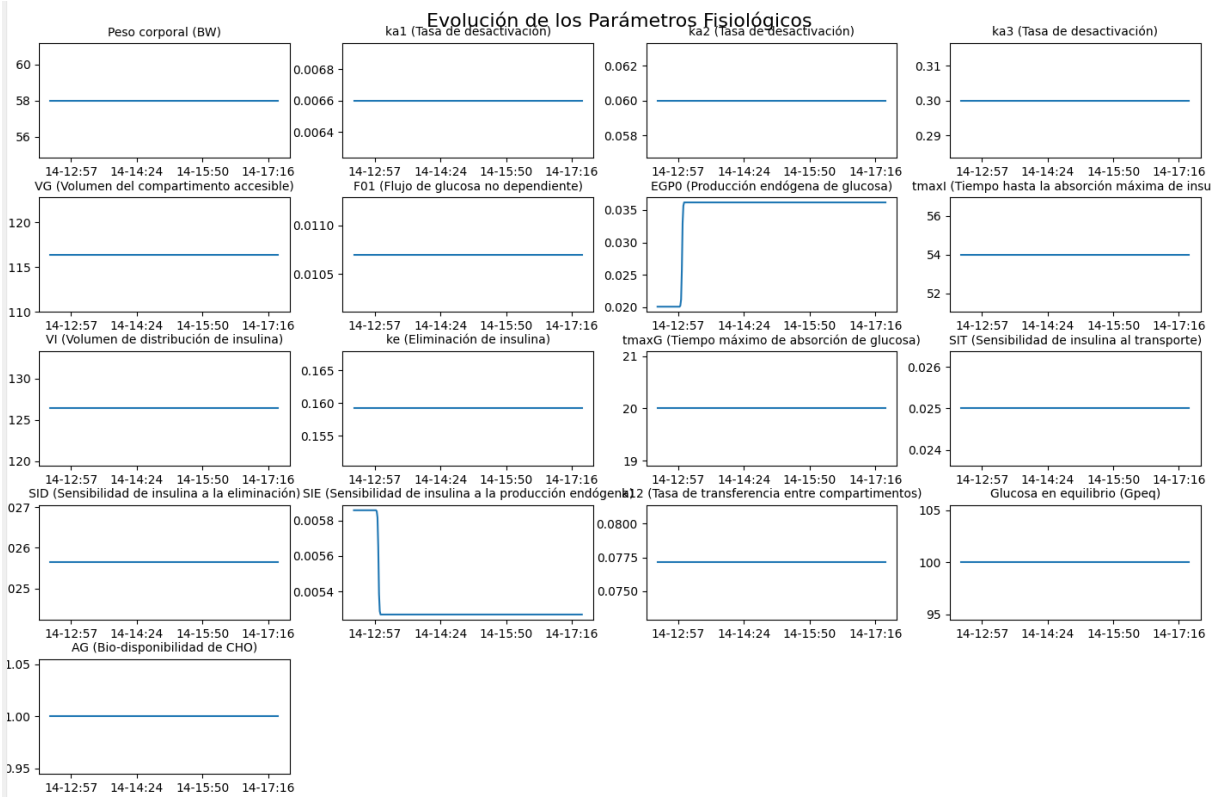


Figura 3.38: Parámetros fisiológicos tras bolo de insulina y estado de estrés.

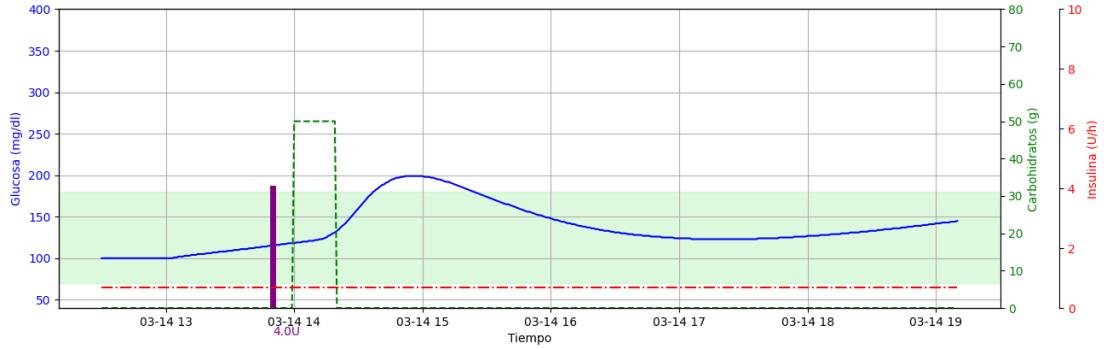


Figura 3.39: Glucosa tras bolo de insulina e infección.

Ya se ha ido viendo en simulaciones anteriores, que dependiendo del IIR (insulin infusion rate) de basal, la glucosa tiende a estar más cercana al objetivo de 100 mg/dL o no. Dados unos parámetros del paciente y una glucosa en equilibrio, se puede obtener el IIR en equilibrio que conseguiría, bajo ningún input de hidrato de carbono o bolo de insulina, mantener la glucosa en 100 mg/dL. Con los parámetros iniciales, el IIR de equilibrio es 0.868 U/h. Así, se puede entender por qué en 3.24, con una basal de 0.7 U/h, la glucosa iba aumentando poco a poco. Para cada parámetro variado se incluye el IIR en equilibrio resultante.

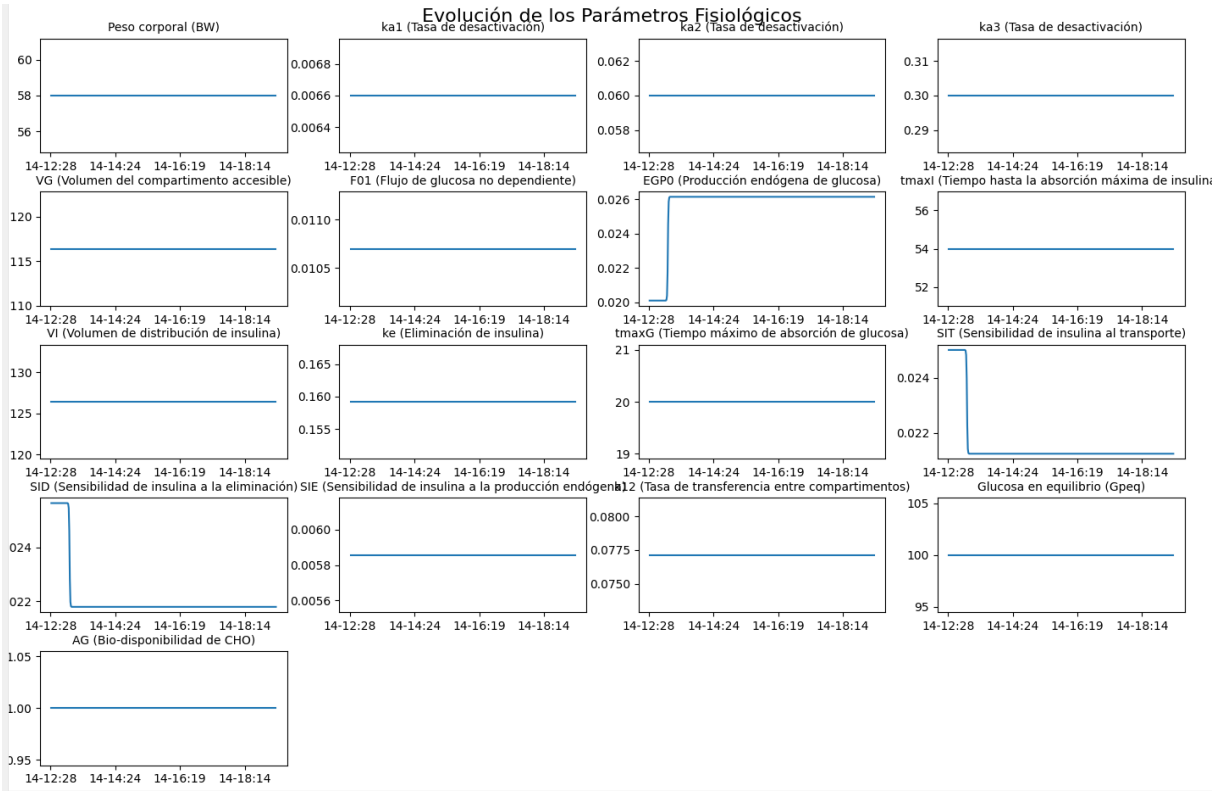


Figura 3.40: Parámetros fisiológicos tras bolo de insulina e infección.

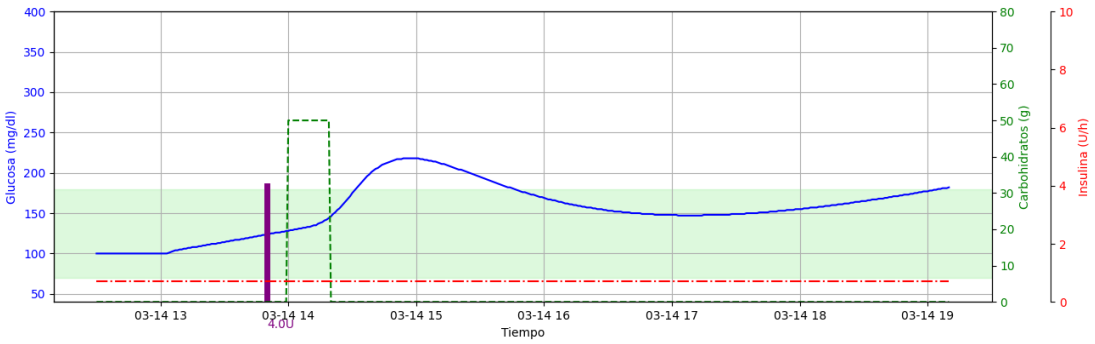


Figura 3.41: Glucosa tras bolo de insulina y fiebre.

3.5.3.1. SIT, SID, SIE

- SIT. En 3.45 se ve la evolución de la glucosa. Las líneas de glucosas altas corresponden a SIT bajos. En 3.46 se muestra como varían las variables de estado. Q1 y Q2 son mayores a menor SIT. En 3.47 se muestran los valores de SIT para la simulación.

  1. SIT = 0.0150, corresponde a un IIR en equilibrio de 1.09975
  2. SIT = 0.0217, corresponde a un IIR en equilibrio de 0.92870
  3. SIT = 0.0283, corresponde a un IIR en equilibrio de 0.81954
  4. SIT = 0.0350, corresponde a un IIR en equilibrio de 0.74200

- SID. En 3.48 se ve la evolución de la glucosa. Las líneas de glucosas altas corresponden a

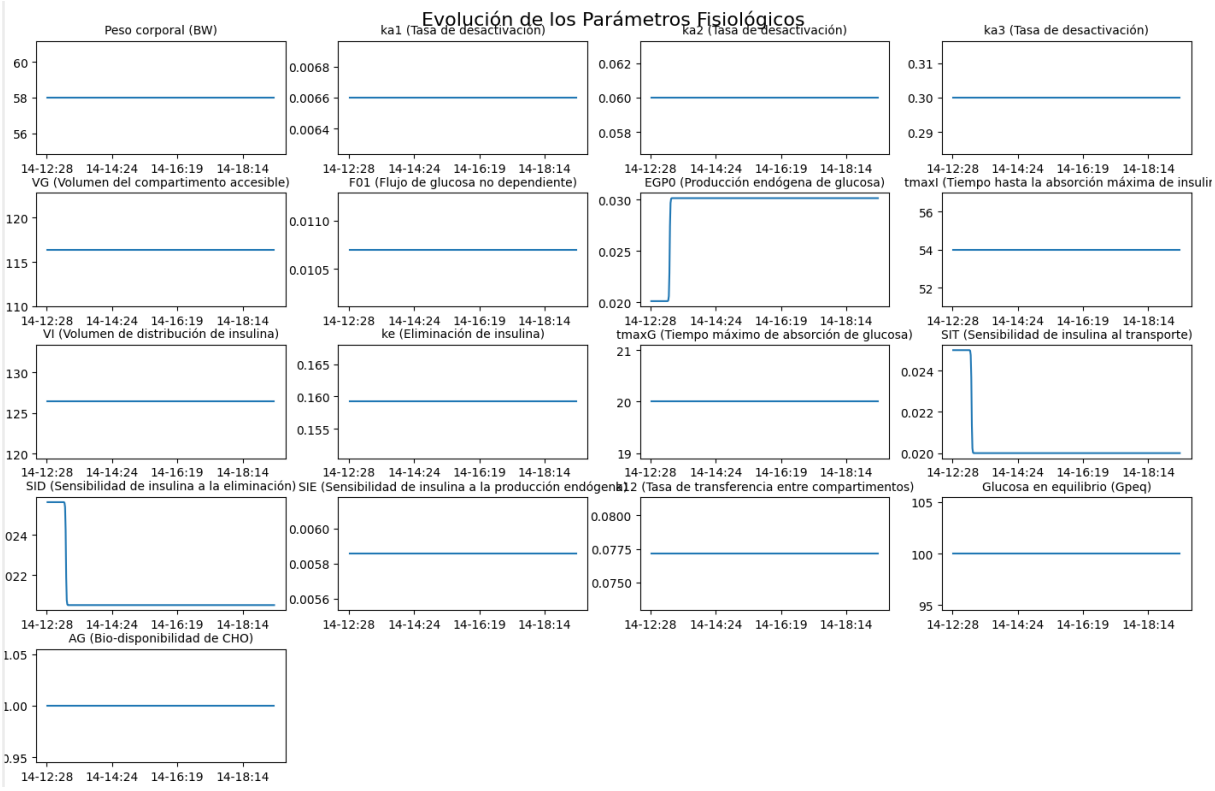


Figura 3.42: Parámetros fisiológicos tras bolo de insulina y fiebre.

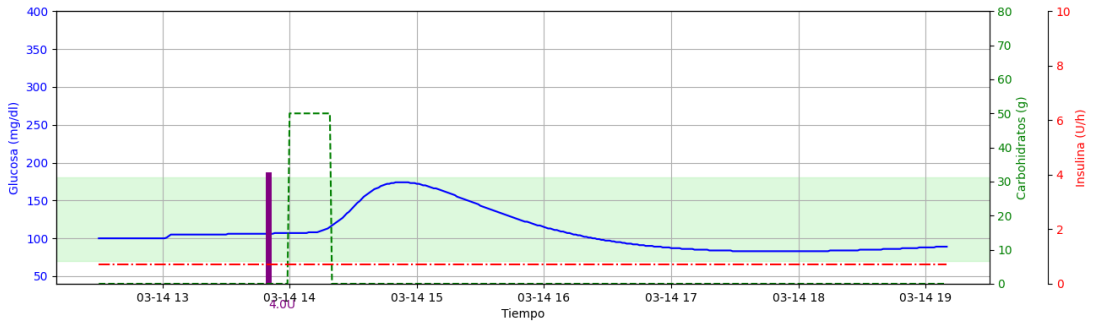


Figura 3.43: Glucosa tras bolo de insulina y deshidratación.

SID bajos.

1. SID = 0.0002000, corresponde a un IIR en equilibrio de 0.97073
2. SID = 0.0002267, corresponde a un IIR en equilibrio de 0.91816
3. SID = 0.0002534, corresponde a un IIR en equilibrio de 0.87364
4. SID = 0.0002800, corresponde a un IIR en equilibrio de 0.83530

- SIE. En 3.49 se ve la evolución de la glucosa. Las líneas de glucosas altas corresponden a SIE bajos.

1. SIE = 0.004, corresponde a un IIR en equilibrio de 0.89292
2. SIE = 0.006, corresponde a un IIR en equilibrio de 0.86710
3. SIE = 0.008, corresponde a un IIR en equilibrio de 0.84212

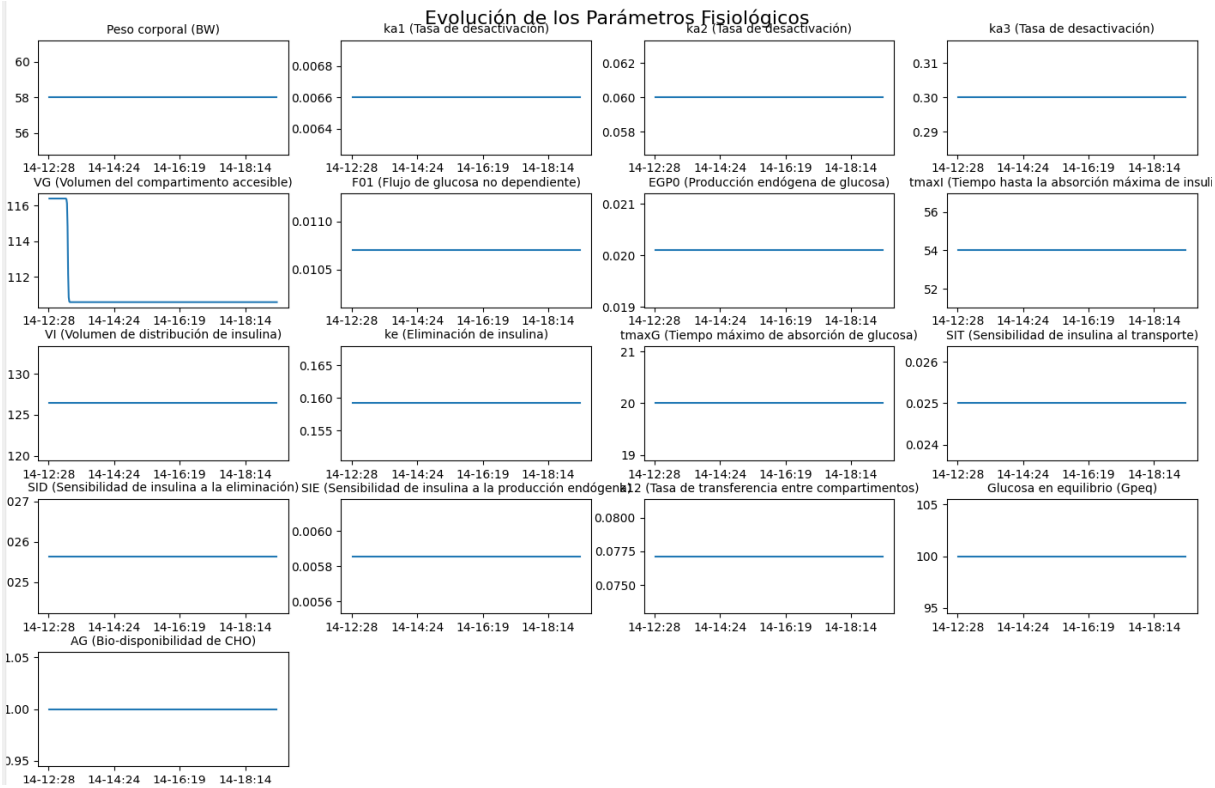


Figura 3.44: Parámetros fisiológicos tras bolo de insulina y deshidratación.

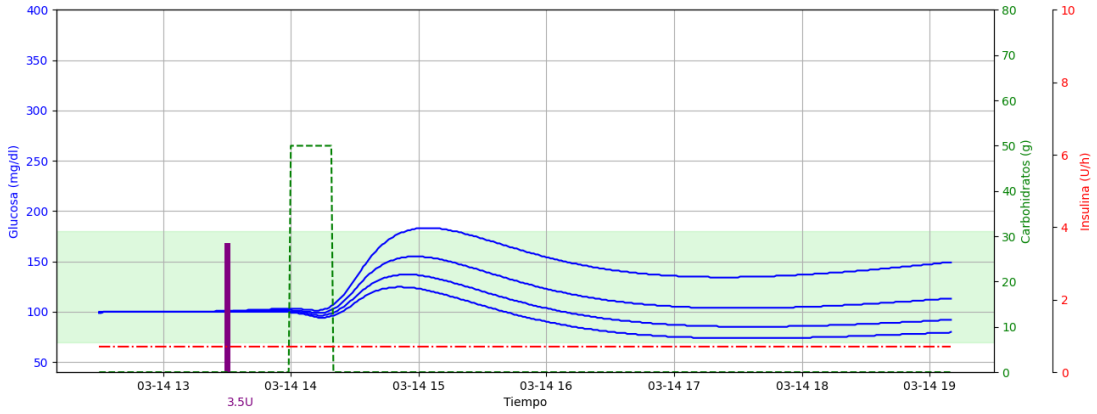


Figura 3.45: Glucosa con variación en el parámetro SIT.

4. SIE = 0.010, corresponde a un IIR en equilibrio de 0.81799

3.5.3.2. EGP0

En 3.50 se ve la evolución de la glucosa. Las líneas de glucosas altas corresponden a EGP0 altos. En 3.51 se muestra como varían las variables de estado. Q1 y Q2 son mayores a mayor EGP0. En 3.52 se muestran los valores de EGP0 para la simulación.

1. EGP0 = 0.01400, corresponde a un IIR en equilibrio de 0.50346

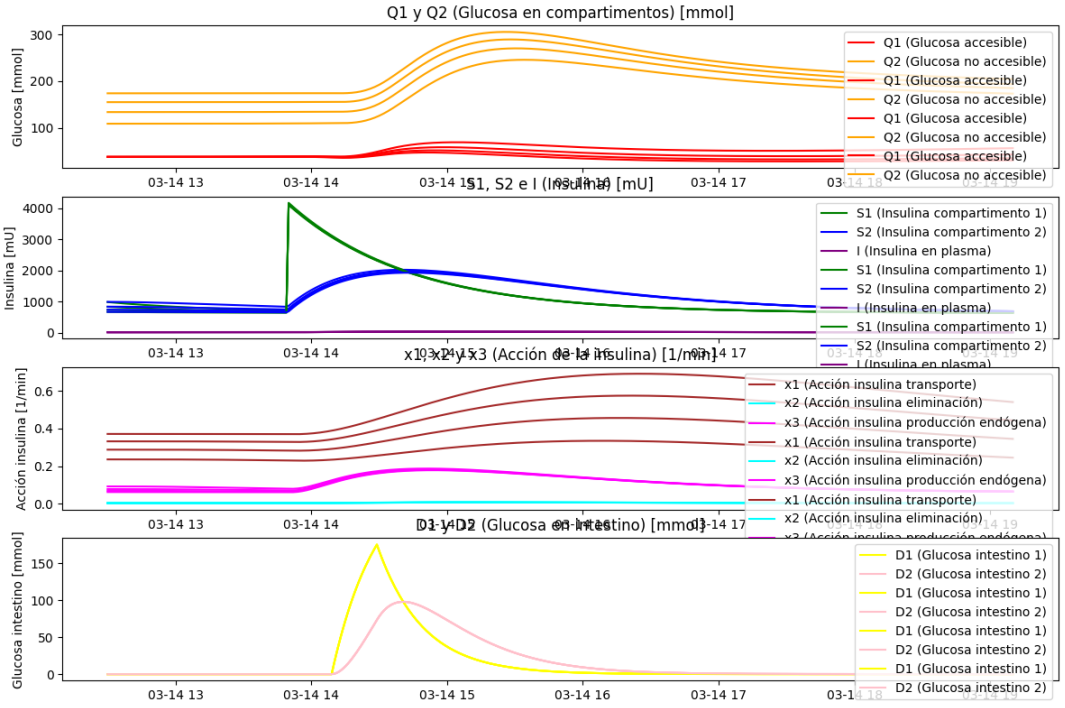


Figura 3.46: Variables de estado con variación en el parámetro SIT.

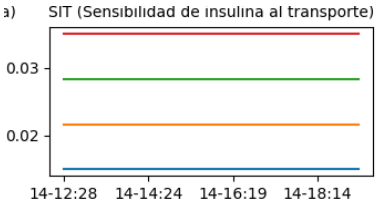


Figura 3.47: Variación en el parámetro SIT.

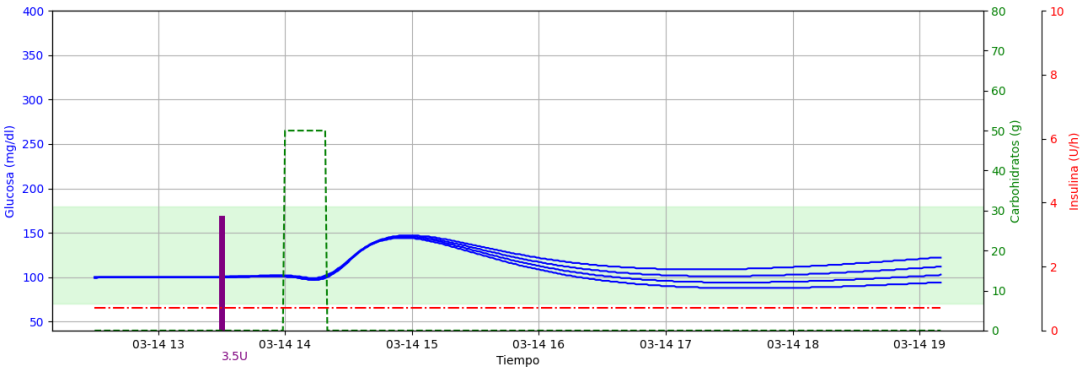


Figura 3.48: Glucosa con variación en el parámetro SID

2. EGP0 = 0.01867, corresponde a un IIR en equilibrio de 0.79816
3. EGP0 = 0.02333, corresponde a un IIR en equilibrio de 1.01018
4. EGP0 = 0.02800, corresponde a un IIR en equilibrio de 1.18381

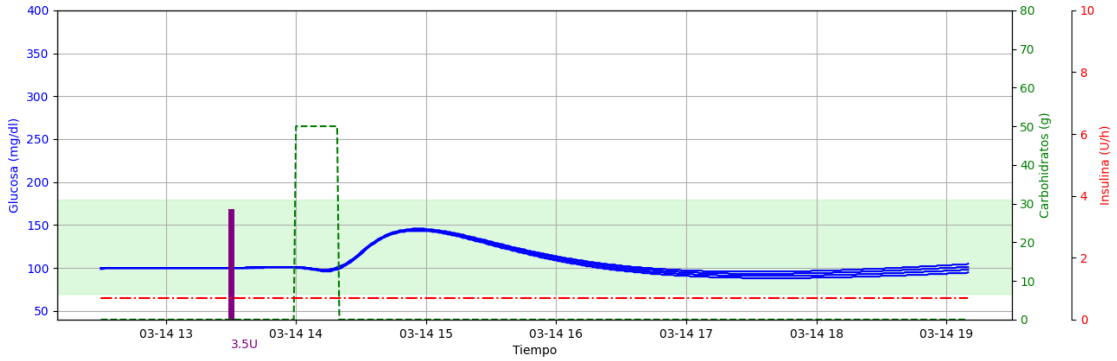


Figura 3.49: Glucosa con variación en el parámetro SIE.

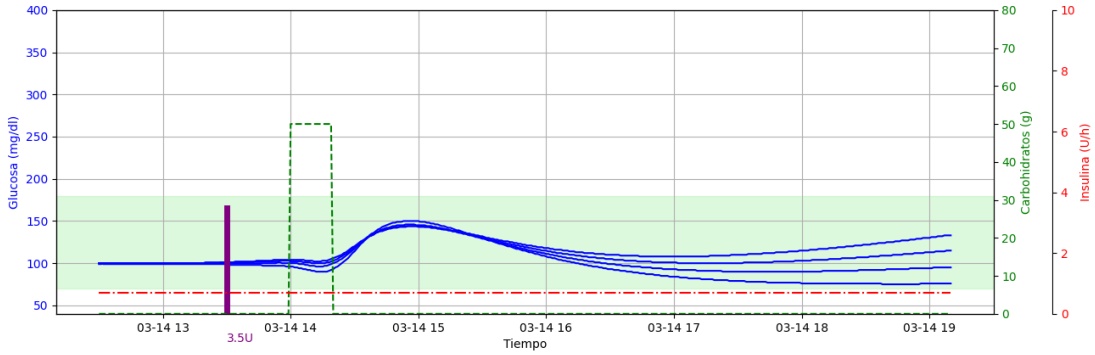


Figura 3.50: Glucosa con variación en el parámetro EGP0.

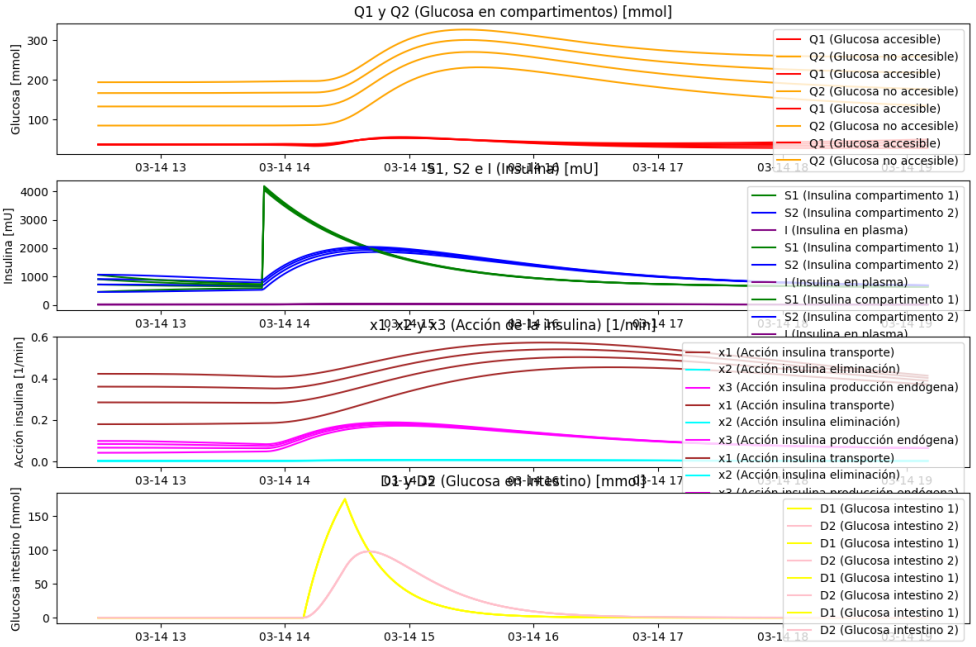


Figura 3.51: Variables de estado con variación en el parámetro EGP0.

El Capítulo 7 contiene una sección dedicada a la validación de este modelo de diabetes. Para validarlo, se compara la evolución de la glucosa con datos reales obtenidos a través de la nube

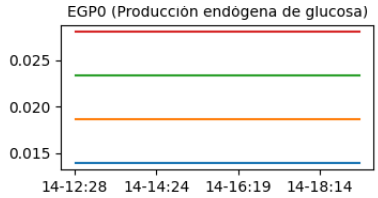


Figura 3.52: Variación en el parámetro EGP0.

de Medtronic, tal y como se detalla en el Capítulo 7.

3.6. Conclusiones

Este capítulo ha documentado el proceso completo de desarrollo de la herramienta DiabetesTool, desde el análisis y diseño hasta la implementación. La herramienta implementa un modelo matemático de diabetes tipo 1 basado en el Cambridge Model, combinado con un motor de integración numérica adaptativa RK1-RK2 y una interfaz gráfica de usuario para la simulación.

Se han definido requisitos funcionales y no funcionales que guían su diseño, y se ha estructurado la herramienta en módulos software bien diferenciados. Las pruebas y simulaciones realizadas confirman la validez del modelo en escenarios clínicamente realistas.

Esta implementación servirá de base para los algoritmos de soporte a la decisión que se abordarán en los siguientes capítulos, abriendo la posibilidad de desarrollar sistemas de control automático de insulina más personalizados y robustos.

Capítulo 4

Algoritmo de la bomba de insulina

4.1. Introducción

En el capítulo anterior se ha detallado el modelo de simulación de la diabetes tipo 1, incluyendo la representación fisiológica de la absorción de glucosa, los efectos de la insulina y el impacto de eventos externos. A partir de este modelo, en el presente capítulo se desarrolla el algoritmo que gobierna el comportamiento de una bomba de insulina inteligente, capaz de decidir automáticamente las dosis necesarias de insulina para mantener la glucosa en rangos óptimos.

Este algoritmo sigue una lógica de control híbrido, donde se combinan decisiones automáticas con entradas manuales por parte del usuario, como la indicación de comidas o eventos especiales (ejercicio, alcohol, enfermedad, etc.). Además, se implementa un controlador PID adaptativo que regula la administración de insulina basal y calcula bolos de corrección, ajustando sus parámetros en función de la estabilidad glucémica y el tiempo en rango.

La arquitectura del algoritmo ha sido diseñada para ser flexible, escalable e integrable con una interfaz gráfica de usuario desplegable en dispositivos como la Raspberry Pi. Por último, se presentan las pruebas de validación del algoritmo mediante simulaciones dinámicas, utilizando tanto escenarios con eventos como sin ellos.

La estructura de este capítulo es la siguiente:

- En la Sección 4.2, se explica la **lectura de glucosa**, el sistema de tiempo y la sincronización con la fuente de datos.
- En la Sección 4.3, se describen las **entradas de usuario**, incluyendo parámetros configurables, ingesta de alimentos y eventos que afectan al metabolismo.
- En la Sección 4.4, se detalla cómo se calcula la **insulina suministrada**, diferenciando entre insulina basal, bolos de corrección y bolos por comida, incluyendo el tratamiento de UGPs.
- En la Sección 4.5, se presenta la **arquitectura del algoritmo**, sus módulos principales y las dependencias entre componentes.
- En la Sección 4.6, se describen los **ejemplos de uso** del algoritmo mediante simulaciones dinámicas en distintos escenarios.

4.2. Lectura de glucosa

El primer paso para implementar el funcionamiento de una bomba de insulina es considerar el tiempo y las lecturas de glucosa como elementos clave. La bomba de insulina puede ser vista como un sistema cuya señal de entrada es la glucosa, que varía en función del tiempo. La generación de esta señal, es decir, los datos de glucosa, proviene de una fuente externa a la bomba. Esto implica que la bomba simplemente recibe los datos sin tener conocimiento exacto de su origen. En este trabajo, las fuentes de datos pueden provenir del modelo de diabetes o de datos reales obtenidos de Medtronic en tiempo real.

- **Modelo de diabetes**: En la Sección 3 se describe el modelo de diabetes, el cual genera los datos de glucosa en función del tiempo a partir de unos parámetros e inputs específicos. Estos datos se obtienen a través de las ecuaciones y el método numérico utilizado. Si se emplea el modelo de diabetes y se retroalimenta con los inputs de insulina suministrados por la bomba (outputs generados por el algoritmo de la bomba), el conjunto formado por el modelo de diabetes y la bomba de insulina puede ser entendido como un sistema con retroalimentación.
- **Datos reales**: En la Sección 6 se explicará cómo se obtienen los datos de tiempo y glucosa (junto con otros parámetros adicionales) en tiempo real. En este caso, los outputs generados por el algoritmo de la bomba no pueden ser administrados al paciente directamente, ya que su propia bomba de insulina, con su algoritmo específico, es la que decide la cantidad de insulina a administrar. Sin embargo, estos outputs pueden ser utilizados para comparar y validar el algoritmo implementado en este trabajo.

Cada pareja de tiempo-glucosa se recoge en un archivo en formato .json, con nombre data.json, que es obtenido por una de las dos formas mencionadas.

4.2.1. La variable tiempo

El tiempo es una variable crucial en el comportamiento de una bomba de insulina, ya que todas las decisiones sobre la administración de insulina, ya sea en forma de tasas basales o bolos de corrección, dependen de la evolución temporal de la glucosa en sangre. Es importante tener un control preciso del tiempo para que las intervenciones de insulina se ajusten de manera adecuada a las necesidades del paciente.

En cada inicio de la simulación de la bomba de insulina, el tiempo arranca con la fecha y hora actuales del sistema. Sin embargo, si se desea sincronizar el inicio de la simulación con el último valor de glucosa registrado en el archivo de datos de glucosa, esta sincronización también es posible. Esto permite que la simulación continúe desde el punto en el que se obtuvo la última lectura de glucosa, asegurando coherencia con los datos históricos. Se espera entonces que el archivo de glucosa vaya incluyendo poco a poco nuevos datos para que los vaya leyendo la bomba. Evidentemente, si no se estan recogiendo datos en tiempo real o no se está simulando la evolución de la glucosa con el modelo de diabetes, el valor de la glucosa leído por la bomba será siempre el mismo (el último record de glucosa del archivo).

Si se está simulando el modelo de diabetes, se puede incluir una velocidad de simulación que permita avanzar el tiempo más rápido que en tiempo real. Esta funcionalidad es útil para observar el comportamiento de la bomba y del paciente en intervalos largos de tiempo de manera más eficiente.

Para gestionar el tiempo en la simulación, se han implementado timers mediante callbacks que permiten actualizar tanto el reloj como las lecturas de glucosa. El reloj se actualiza cada 10 segundos en tiempo real, aunque la precisión del reloj en la simulación está limitada a 1 minuto, lo que hace que esta frecuencia de actualización sea suficiente. Las lecturas de glucosa se actualizan cada 5 minutos en tiempo real. Esto se refiere a "tiempo real"porque, dependiendo de la velocidad de simulación seleccionada, este minuto puede corresponder a un periodo mayor o menor en la simulación.

Cada vez que se obtiene una nueva lectura de glucosa, el sistema calcula la tasa basal de insulina que debe administrarse y, en caso de ser necesario, un bolo de corrección para ajustar los niveles de glucosa. Estos cálculos se basan en los algoritmos de control integrados en la bomba de insulina.

Por último, los bolos relacionados con las comidas son introducidos manualmente por el usuario. Este introduce la cantidad de HG y UGPs, convirtiendo el sistema de control en un modelo híbrido y ya no exclusivamente de lazo cerrado. Por lo tanto, estos bolos no están sujetos al sistema temporal de la simulación (aunque ciertos parámetros del paciente si dependen del tiempo, como se ha visto en la descripción de los eventos). Por ello son atemporales, a diferencia de los bolos de corrección y de la insulina basal.

4.2.2. Los datos de glucosa

Tal y como se ha comentado antes, los datos de glucosa son leídos cada 5 minutos real. Es decir

- Si se está simulando a velocidad normal, a medida que avanza 5 minutos la simulación se recoge su dato de glucosa.
- Si se simula a velocidad x10, en un minuto de tiempo real se habrá avanzado 600 segundos de la simulación, es decir, se recogerán datos de glucosa cada step de 30 segundos de simulación.

El archivo de donde se leen los datos de glucosa se va rellenando con datos de la siguiente forma, según la fuente de datos:

- Modelo de diabetes: se incluye en cada record de json, la fecha y hora, glucosa, insulina activa dentro del cuerpo, y variables de estado del modelo de diabetes (que no son utilizados por la bomba).
- Datos reales: se incluye en cada record del json, la fecha, la glucosa y la insulina activa. En el 6 se incluye otros parámetros que se descargan pero que no son almacenados dentro del fichero ya que no los necesita el algoritmo de este trabajo.

4.3. Entradas de usuario

Aunque el objetivo inicial de los sistemas de páncreas artificial ha sido la implementación de un control totalmente en lazo o bucle cerrado (CL), donde el paciente no interviene en las decisiones de administración de insulina, la realidad actual de estos sistemas ha evolucionado hacia un enfoque híbrido o semicerrado. Esto se debe a la complejidad de las dinámicas del cuerpo humano y las limitaciones tecnológicas actuales. A pesar de los avances en el desarrollo de algoritmos capaces de gestionar la glucosa de forma autónoma, aún existen situaciones en las que la intervención del usuario es necesaria para garantizar un control seguro y efectivo de los niveles de glucosa.

El algoritmo implementado en este proyecto ya no es estrictamente un sistema en lazo cerrado puro, ya que incorpora inputs manuales del usuario, como la información relacionada con la ingesta de carbohidratos o la planificación de actividades físicas. Este tipo de intervención es crucial en determinados momentos, especialmente durante las comidas o en episodios de ejercicio intenso, donde el cuerpo requiere ajustes específicos de la dosis de insulina.

La inclusión de estos inputs manuales responde a la necesidad de una mayor flexibilidad y personalización del tratamiento. El sistema híbrido permite que el usuario informe al algoritmo sobre factores que el propio sistema no puede detectar de manera autónoma, mejorando así la precisión de las dosis administradas y adaptándose mejor a las variaciones individuales en el metabolismo.

4.3.1. Parámetros

Existen una serie de parámetros necesarios de configurar en las bombas de insulina que son propios de cada paciente y que suelen ser constantes durante largos periodos de tiempo. De hecho, solamente se suelen cambiar algunos de estos parámetros bajo supervisión del endocrino en alguna de las revisiones anuales que se llevan a cabo. Estos parámetro son:

1. **Glucosa objetivo**: es el objetivo de glucosa que se quiere conseguir. Dependiendo de la hora del día, puede variar. Por ejemplo, es muy típico que durante el día esta glucosa objetivo sea de 100 mg/dL y por la noche sea algo mayor, unos 120 mg/dL, para evitar posibles hipoglucemias nocturnas debidas a varias horas seguidas sin ingerir alimentos. Este objetivo también se puede cambiar dependiendo de las necesidades del paciente, aunque lo más habitual es que estas modificaciones sean debidas a cambios en la actividad física de la persona.

2. Ratio carbohidratos / unidades de insulina (insulin-to-carb ratio IRC): representa la cantidad de hidratos de carbono que es capaz de cubrir una unidad de insulina [34]. Por ejemplo, si el IRC es 6.0 g/U, quiere decir que si se va ingerir un alimento de 12 g de HC, es necesario suministrar 2 U de insulina. Este ratio depende de la sensibilidad a la insulina del paciente, la cual también puede depender de la hora del día.

3. **Factor de sensibilidad a la insulina** medido en mg/dL por U (insulin sensitivity factor ISF) es un algoritmo de corrección de la glucemia antes de las comidas. En particular,

el ISF indica cuántos mg/dL baja la glucosa en sangre por cada unidad de insulina; se obtiene dividiendo 1800 entre el la dosis diaria total de insulina que un paciente necesita a lo largo del día (total daily dose TDD). El TDD incluyen tanto la insulina basal (la administrada de manera continua o en varias dosis para cubrir las necesidades básicas del organismo) como la insulina prandial (la que se administra antes de las comidas para controlar los aumentos de glucosa provocados por la ingesta de alimentos) [35]

4. **Efecto de la insulina dependiendo del cambio de reservorio**: es común que los pacientes con bombas de insulina noten una disminución en la eficacia de la insulina administrada a medida que transcurren los días desde el último cambio del reservorio y el equipo de infusión. Esta disminución en la eficacia se debe a varios factores:

   - Formación de cristales en los tubos: Si el equipo de infusión no se cambia regularmente, pueden formarse pequeños cristales de insulina en los tubos, lo que puede bloquear el flujo de insulina a la cánula. Esto puede provocar un aumento en los niveles de glucosa en sangre.
   - Pérdida de la capacidad de absorción de la piel: La piel puede desarrollar una menor capacidad de absorción de insulina si se utiliza repetidamente la misma zona de inserción sin rotarla. Esto puede llevar a una menor eficacia de la insulina administrada.

   Por ello, se recomienda cambiar el equipo de infusión y el reservorio cada 2 o 3 días. Este parámetro indicaría la cantidad de tiempo (normalmente 2.5 días por lo anterior) durante el cual la insulina presenta una eficacia plena. A partir de ese tiempo, se iría perdiendo poco a poco [36].

5. **Basal máxima** permitida para su administración en un determinado momento. A la hora de calcular la basal, lo normal es que sea una cantidad suficiente para suplir las necesidades del paciente, pero sin superar un valor que podría causar una hipoglucemia futura. Lo típico es un valor de 2 U/h.

Todos estos parámetros se encuentran recogidos en un archivo json de nombre config.json, donde se indica el tipo de parámetro, el nombre, las unidades, y una lista de periodos de tiempo con la hora de inicio y final de dicho periodo, y el valor del parámetro. Por ejemplo para el objetivo:

```
"target": {
    "name": "BG Target",
    "period": [
        {
            "from": "00:00",
            "to": "12:00",
            "value": "110"
        },
        {
            "from": "12:00",
            "to": "24:00",
            "value": "100"
        }
    ],
    "unit": "mg/dL"
}
```

Código 4.1: Detalle del objetivo de glucosa en el archivo config.json.

4.3.2. Comidas

Cada vez que el paciente va a ingerir un alimento que contenga hidratos de carbono, y que esta ingesta no sea consecuencia de una hipoglucemia (generalmente se deben ingerir unos 10-15 g de HC simple para contrarrestar una hipoglucemia), ha de indicárselo a la bomba de insulina mediante acción manual.

Para la diabetes, el macronutriente esencialmente importante a tener en cuenta es el hidrato de carbono. Normalmente, un usuario típico de bombas de insulina ha sido formado por algún enfermero especialista en diabetes para saber calcular hidratos de carbono en función de la cantidad y del alimento. Con la experiencia, este cálculo se va mejorando.

En la bomba de Medtronic modelo 780G el input introducido manualmente por el usuario es el peso en gramos de HC. El algoritmo de la bomba de insulina de este trabajo también utiliza este input. Gracias al parámetro de Ratio de HC, la obtención de las U de insulina necesarias es directa, sin más que dividir los HC entre el ratio. Además, se añade una variable adicional llamada Ajuste:

- Es posible que ya exista insulina activa en el cuerpo del paciente. Esta cantidad es restada a la que se había calculado, evitando así una sobredosis de insulina e hipogluicemia. También se realiza un ajuste negativo en caso de que la glucosa actual del paciente sea menor a la del objetivo.
- Por otro lado, si la glucosa actual del paciente es mayor, se calcula una cantidad adicional de insulina a administrar con el parámetro de Sensibilidad a la insulina.

Algo que no presenta la bomba de Medtronic y que ha de tenerse en consideración es la cantidad de grasa y proteína ingerida. Es conocido que estos macronutrientes también afectan a la glucosa. Se incluye por tanto otro input llamado UGP (unidad grasa-proteína). El cálculo de las UGPs se hace calculando la cantidad de calorías que aportan en conjunto las proteínas y grasa del alimento. Para este cálculo, hay que conocer cuántas calorías aportan cada uno por gramo:

- 1 gramo de grasa aporta 9 kcal.
- 1 gramo de proteína aporta 4 kcal.

Cada 150 kcal que provengan de grasa y/o proteína constituye 1 UPG.

Según [32], a partir de las 3 horas posteriores a la ingesta, el nivel de azúcar en sangre puede continuar aumentando entre 30 y 50 mg/dL por cada unidad grasa-proteína (UGP), con un pico glucémico entre las 5 y 8 horas después de la comida. Este comportamiento se debe a la digestión más lenta de las grasas y proteínas, que retrasa la absorción de glucosa y prolonga la hiperglucemia.

Para contrarrestar este efecto, el algoritmo de la bomba administra insulina de forma diferida y escalonada cuando se detectan UGPs en la comida.

- La insulina correspondiente a los carbohidratos se administra de forma inmediata, ajustada según el ratio del periodo, la eficacia de la insulina y una posible corrección por la glucosa actual.
- Si se han identificado UGPs asociadas a la comida, el bolo total se divide en tantas fracciones como UGPs haya.
- Cada fracción de insulina se programa para ser administrada de forma diferida: la primera a las 3 horas de la ingesta, y las siguientes separadas por una hora cada una.

El pseudocódigo para la decisión de la administración de insulina por UGPs se muestra en la Sección 4.4.3

4.3.3. Eventos

La situación de cada paciente en cada momento puntual puede ser muy diferente, sin necesariamente cambiar sus parámetros intrínsecos de Ratio HC y Sensibilidad a la insulina. Por ejemplo, puede que tenga una celebración y durante toda la tarde se encuentre ingiriendo alcohol, lo que provoca que el hígado del paciente deje de liberar endógenamente glucosa a la sangre, ya que se encuentra procesando el alcohol, y por lo tanto, puede que tenga más tendencia a hipoglucemia durante el día siguiente. Otro ejemplo muy típico y que contempla una casuística interesante, es el ejercicio. Dependiendo de las horas de ejercicio, estado físico inicial del paciente, e intensidad, el cuerpo puede actuar requiriendo más glucosa, o liberando más al cuerpo sanguíneo debido a la producción de cortisol (hormona del estrés), debido a la adrenalina.

En los siguientes apartados se comentan todos los posibles eventos que son considerados dentro del algoritmo de la bomba de este trabajo.

4.3.3.1. Alcohol

El hígado libera glucosa al torrente sanguíneo según sea necesario para ayudar a mantener el azúcar en la sangre a un nivel normal. Cuando se toma alcohol, el hígado necesita descomponerlo. Mientras el hígado lo procesa, deja de liberar glucosa. Como resultado, el nivel de azúcar en la sangre puede caer velozmente, lo cual pone en riesgo de sufrir una reducción de azúcar en sangre. Si se inyecta insulina, esto puede causar una reducción grave del nivel de azúcar. Beber alcohol sin comer al mismo tiempo también puede incrementar este riesgo. El riesgo de bajo nivel de azúcar en sangre permanece durante horas después de beber alcohol.

Para la bomba de insulina, la forma de indicar el alcohol ingerido es mediante periodos de tiempo, con fecha y hora de inicio y final (determinado por la duración del evento), y una cantidad de bebidas. Esta cantidad de bebidas es orientativa, aunque la idea es que 1 bebida se aproxima al alcohol contenido en una copa de destilado (sin contar la mezcla, o considerando que la mezcla es una bebida sin azúcar o zero). Es posible que al ingerir otras bebidas alcohólicas como vino o cervezas, la cantidad de alcohol no sea la misma que en una copa de destilado. Por ello, el usuario debe conocer previamente cómo le afecta cada tipo de alcohol y en qué medida.

La forma de actuar de la bomba de insulina de acuerdo a la ingesta de alcohol es como sigue:

- **Target de glucosa:**
  - Si la cantidad ingerida es pequeña (menor de 5), el objetivo de glucosa aumenta en 10 mg/dL por cada unidad de bebida durante el tiempo de la ingesta, extendiéndose 30 minutos posteriores. Por ejemplo, si la cantidad = 3, y se está bebiendo de 17:30 a 19:00, y el objetivo nominal es 100 mg/dL, entonces el objetivo se modifica a 130 mg/dL de 17:30 a 19:30. De esta forma, tanto la insulina basal, como los bolos de corrección o por comida se calculan buscando este objetivo. Al ser un objetivo mayor, se evitan posibles bajadas de azúcar e hipoglucemias.
  - Si la cantidad ingerida es grande (5 o más), el objetivo aumenta en 50 mg/dL desde 30 minutos antes del inicio de la ingesta (para evitar correcciones previas que incrementen la cantidad de insulina activa en el cuerpo) hasta 30 minutos posteriores por unidad de bebida. Por ejemplo, si la cantidad = 7, y se está bebiendo de 17:30 a 19:00, y el objetivo nominal es 100 mg/dL, entonces el objetivo se modifica a 150 mg/dL de 17:00 a 23:00. Cuando se ingieren grandes cantidades de alcohol, el riesgo de hipoglucemias se extiende durante las siguientes 24 horas. Además, los hidratos de carbono consumidos durante este periodo tardan más tiempo en subir el azúcar en sangre.
- **Sensibilidad a la insulina:**
  - Si la cantidad es mayor que 2 pero menor que 5, la sensibilidad a la insulina aumenta durante la ingesta de alcohol. El incremento de sensibilidad tiene una forma triangular: aumenta linealmente durante el tiempo que dura la ingesta y luego desciende. La sensibilidad máxima es un 10 % mayor por cada unidad de bebida. Por ejemplo, si se están consumiendo 4 bebidas de 18:00 a 19:00, la sensibilidad se incrementará gradualmente hasta un 40 % al final de la ingesta, para luego volver al valor original en los 30 minutos posteriores.
  - Si la cantidad es 5 o mayor, la sensibilidad a la insulina sigue una forma trapezoidal. Durante los primeros 30 minutos, aumenta linealmente hasta alcanzar un 50 % más de sensibilidad. Este nivel se mantiene constante durante toda la ingesta, y comienza a descender lentamente 30 minutos después del fin del consumo, volviendo al valor original.

4.3.3.2. Deporte

La actividad física tiene un impacto directo en los niveles de glucosa en sangre, especialmente en personas con diabetes tipo 1. La intensidad y la duración del ejercicio son factores clave que determinan cómo fluctúa el azúcar en sangre. El ejercicio aeróbico de baja o moderada intensidad, como caminar o trotar, tiende a disminuir los niveles de glucosa durante y después de la actividad. Sin embargo, el ejercicio de alta intensidad o anaeróbico, como levantar pesas o esprints, puede provocar un aumento temporal de los niveles de azúcar en sangre debido a la liberación de hormonas como la adrenalina.

En general, cuanto más intensa y prolongada sea la actividad, más probable es que el nivel de azúcar en la sangre disminuya después de la actividad, debido a una mayor sensibilidad a la insulina. Por lo tanto, la bomba de insulina ajusta tanto el objetivo de glucosa como la

sensibilidad a la insulina en función de la intensidad y duración del ejercicio. Estos ajustes ayudan a prevenir hipoglucemias o hiperglucemias relacionadas con la actividad física.

La forma en que la bomba de insulina responde al ejercicio es como sigue:

- **Target de glucosa:**
  - Si la intensidad del ejercicio es baja (menos de 5) y la duración es menor de 60 minutos, el objetivo de glucosa aumenta en 50 mg/dL desde 2 horas antes del inicio del ejercicio hasta 30 minutos después. Esto se debe a que el ejercicio de baja intensidad, aunque beneficioso, puede reducir la glucosa de manera constante, y un objetivo más alto ayuda a evitar hipoglucemias.
  - Si la intensidad es baja pero la duración es mayor o igual a 60 minutos, el objetivo también aumenta en 50 mg/dL, pero el ajuste se extiende desde 3 horas antes del ejercicio hasta 30 minutos después de su finalización. Esto es para cubrir el efecto prolongado del ejercicio de larga duración en la reducción de la glucosa.
  - Si la intensidad es alta (5 o más) y la duración es mayor o igual a 60 minutos, el objetivo se incrementa igualmente en 50 mg/dL desde 2 horas antes del inicio hasta 30 minutos después. En estos casos, el ejercicio intenso provoca una caída más drástica del azúcar en sangre una vez que la actividad finaliza, por lo que es necesario ajustar el objetivo para evitar hipoglucemias.
  - Si la intensidad es alta pero la duración es menor de 60 minutos, el objetivo no se modifica, ya que en este tipo de ejercicios cortos y de alta intensidad, es más probable que la glucosa se mantenga o incluso aumente momentáneamente durante el ejercicio.
- **Sensibilidad a la insulina:**
  - Si la duración del ejercicio es menor de 30 minutos, la sensibilidad a la insulina aumenta progresivamente durante el ejercicio y disminuye gradualmente después. El incremento de sensibilidad sigue una forma triangular: aumenta linealmente durante el ejercicio hasta alcanzar un valor máximo (determinado por la intensidad), y luego disminuye de vuelta a los niveles originales durante un periodo equivalente a la duración del ejercicio.
  - Si la duración del ejercicio es mayor o igual a 30 minutos, la sensibilidad a la insulina sigue una forma trapezoidal. Durante los primeros 30 minutos, la sensibilidad aumenta de manera lineal, alcanzando un valor máximo que es un 10 % mayor por cada nivel de intensidad. Este valor máximo se mantiene constante hasta el final del ejercicio, y luego disminuye gradualmente durante los 30 minutos posteriores. Este ajuste asegura que la sensibilidad no caiga abruptamente después de largos periodos de ejercicio, evitando hipoglucemias severas.

Tanto la sensibilidad a la insulina como el objetivo varían en función de las horas de deporte, y por ello es conveniente añadir con suficiente tiempo de antelación el periodo en el que se va a estar realizando deporte.

4.3.3.3. Enfermedad

Las enfermedades pueden afectar de manera significativa la regulación de la glucosa en personas con diabetes, ya que el cuerpo libera hormonas de estrés que pueden aumentar la resistencia a la insulina. Esto significa que el cuerpo necesita más insulina para mantener los niveles de glucosa en sangre dentro de un rango saludable. Las infecciones, la fiebre y la deshidratación son algunos ejemplos de enfermedades que pueden influir en la sensibilidad a la insulina.

Dependiendo del tipo de enfermedad y su duración, los efectos sobre la glucosa y la sensibilidad a la insulina pueden variar. Por ejemplo, la deshidratación puede disminuir la capacidad del cuerpo para utilizar la insulina de manera eficaz, lo que aumenta los niveles de glucosa en sangre. En cambio, durante infecciones o episodios de fiebre, el cuerpo produce hormonas que reducen la sensibilidad a la insulina, llevando a un incremento en los niveles de glucosa.

En el modelo, los ajustes durante los períodos de enfermedad son principalmente en la sensibilidad a la insulina, mientras que el valor del objetivo de glucosa no se ve afectado.

El código trata diferentes tipos de enfermedad de la siguiente manera:

- Deshidratación: Durante la deshidratación, la sensibilidad a la insulina disminuye en un 10 %. Esto se traduce en un menor efecto de la insulina en el control de la glucosa. En el código, esto se refleja mediante una reducción del valor de sensibilidad en un 10 %, que se mantiene constante durante todo el período de deshidratación.
- Infección y Fiebre: Tanto la infección como la fiebre causan una disminución mayor en la sensibilidad, del 20 %. El efecto sobre la sensibilidad se modela de manera trapezoidal. Al inicio de la enfermedad, la sensibilidad disminuye gradualmente durante los primeros 30 minutos, hasta alcanzar un mínimo. Durante el resto del período de la enfermedad, la sensibilidad se mantiene en este valor mínimo. Una vez finalizada la enfermedad, la sensibilidad regresa lentamente a su valor original en un período de 30 minutos adicionales.

Por lo tanto, los cambios en la sensibilidad a la insulina durante los episodios de enfermedad dependen tanto del tipo como de la duración del evento. Esto permite al modelo simular de manera más precisa el comportamiento del cuerpo en diferentes situaciones y ajustar el tratamiento en consecuencia.

4.3.3.4. Estado anímico

El estado anímico de una persona puede afectar significativamente el control de la glucosa en sangre. Factores como el estrés o la relajación pueden influir en la forma en que el cuerpo responde a la insulina. El estrés, por ejemplo, provoca la liberación de hormonas como el cortisol, que pueden aumentar los niveles de glucosa en sangre al reducir la sensibilidad a la insulina. En contraste, los momentos de relajación suelen mejorar la sensibilidad a la insulina, facilitando el control de la glucosa.

En este modelo, se ajustan tanto el objetivo de glucosa como la sensibilidad a la insulina durante los períodos de estrés y relajación. Estos ajustes reflejan la capacidad del cuerpo para gestionar

los niveles de glucosa en diferentes estados emocionales.

El código trata los cambios en el estado anímico de la siguiente manera:

- Estrés: Durante los episodios de estrés, el objetivo de glucosa aumenta en 20 mg/dL. Esto significa que el cuerpo permite un nivel de glucosa más alto antes de realizar correcciones, lo que puede ayudar a evitar caídas bruscas en los niveles de glucosa debido a la menor sensibilidad a la insulina. Además, si el episodio de estrés dura más de 120 minutos, la sensibilidad a la insulina se reduce en un 10 %, haciendo que la insulina sea menos efectiva para bajar la glucosa.
- Relajación: En los momentos de relajación, el cuerpo responde mejor a la insulina, por lo que la sensibilidad aumenta en un 10 %. Esto permite que la insulina sea más eficaz en la reducción de los niveles de glucosa durante estos períodos.

Los ajustes realizados en este modelo permiten simular cómo el estado emocional influye en la gestión de la diabetes, con el objetivo de ajustar las dosis de insulina y los objetivos de glucosa de manera más personalizada y efectiva.

4.3.3.5. Cambio de reservorio de insulina

El cambio de reservorio en las bombas de insulina es un aspecto crucial para mantener la eficacia de la insulina administrada. Como se mencionó anteriormente, es común que la insulina pierda eficacia cuando se aproxima el tiempo de cambiar el reservorio y el equipo de infusión. Factores como la formación de cristales en los tubos de infusión y la pérdida de la capacidad de absorción de la piel pueden influir en este proceso, lo que provoca que la insulina administrada no sea tan efectiva a medida que pasan los días.

En el archivo de configuración `config.json`, se establece un parámetro que define el tiempo a partir del cual la insulina empieza a perder efectividad, comúnmente a los 2.5 días después del último cambio de reservorio. Este tiempo marca el punto donde el efecto de la insulina comienza a disminuir. Además, en el archivo `events.json`, se especifica un evento denominado Çambio del reservorio", que indica la fecha y la hora en la que se realizó un cambio de reservorio, así como la periodicidad (normalmente 3 días entre cada cambio). Esta fecha puede corresponder al último cambio o no, ya que el último cambio se calcula con la periodicidad en días de cambio.

El modelo tiene en cuenta este evento de la siguiente manera: desde el cambio de reservorio, se considera que la insulina tiene una eficacia plena durante los primeros 2.5 días. Después de este período y hasta el próximo cambio (es decir, entre los 2.5 y 3 días), la efectividad de la insulina disminuye progresivamente. Este ajuste en la eficacia se aplica mediante un factor multiplicativo que afecta la insulina activa en el cuerpo. Si la eficacia es menor, es probable que se requiera un aumento en la dosis de insulina basal o en los bolos de corrección para compensar esta disminución.

En bombas como la Medtronic 780G, la experiencia ha demostrado que al no aplicar este factor de eficacia, los pacientes tienden a tener niveles de glucosa más elevados a partir del segundo día, ya que la insulina administrada es menos efectiva. Esto obliga a la bomba a aplicar más

bolos de corrección, bien sea tras las comidas o por el hecho de que la dosis basal no es suficiente para mantener niveles adecuados de glucosa en sangre debido a esta menor eficacia.

4.3.3.6. Pseudocódigo de funciones auxiliares de eventos

A continuación se muestran los Pseudocódigos 4.1 y 4.3 de las funciones que ajustan los parámetros de la bomba de insulina en función de los eventos. Estas funciones se encargan de modificar el objetivo de glucosa y la sensibilidad a la insulina según los eventos registrados en el archivo `events.json`.

Pseudocódigo 4.1 Ajuste de parámetros por eventos (I).

```
Función events_adjust(time, events, name, value):
    Si name ∉ {'target', 'sensitivity'} entonces:
        retornar value
    exercise ← events['exercise']['period']
    alcohol ← events['alcohol']['period']
    illness ← events['illness']['period']
    mood ← events['mood']['period']
    Para cada periodo en exercise hacer:
        Extraer start_time, duration, intensity, end_time
        Si name = 'target' entonces:
            Aplicar reglas condicionales para sumar 50 a target
        Si name = 'sensitivity' entonces:
            max_sensitivity ← value × (1 + 0.1 × intensity)
            duration_sensitivity ← duration + 30 × intensity
            Aplicar función triangular o trapezoidal según duración
    Para cada periodo en alcohol hacer:
        Extraer start_time, duration, amount, end_time
        Si name = 'target' entonces:
            Aplicar reglas para sumar al target según amount
        Si name = 'sensitivity' entonces:
            max_sensitivity ← value × (1 + 0.1 × amount)
            Aplicar función triangular o trapezoidal según amount
    Para cada periodo en illness hacer:
        Extraer start_time, duration, type, end_time
        Si name = 'sensitivity' entonces:
            Si type = 'Dehydration' entonces: value ×= 0.9
            Si type ∈ {'Infection', 'Fever'} entonces:
                min_sensitivity ← value × 0.8
                Aplicar función trapezoidal descendente
```

Pseudocódigo 4.2 Ajuste de parámetros por eventos (II).

```
    Para cada periodo en mood hacer:
        Extraer start_time, duration, type, end_time
        Si name = 'target' y type = 'Stress' entonces: value += 20
        Si name = 'sensitivity' entonces:
            Si type = 'Stress' y duration >120 entonces: value ×= 0.9
            Si type = 'Relax' entonces: value ×= 1.1
    Retornar value
```

En el Pseudocódigo 4.3 se muestra la función que decide si un evento debe aplicarse o no en un momento dado. Esta función comprueba si el tiempo actual se encuentra dentro de los periodos de ejercicio o alcohol, y si es así, determina si se debe aplicar el evento o no. Si el evento no se aplica, se retorna `False`, lo que indica que no se deben realizar ajustes en los parámetros de la bomba de insulina.

Pseudocódigo 4.3 Decisión de aplicación de eventos.

```
Función events_apply(time, events):
    exercise ← events['exercise']['period']
    alcohol ← events['alcohol']['period']
    Para cada periodo en exercise hacer:
        Extraer start_time, duration, intensity, end_time
        Si duration ≥ 60 y intensity <5 entonces:
            Si time ∈ [start - 3h, end + 2h] entonces: retornar False
        Si duration ≥ 60 y intensity ≥ 5 entonces:
            Si time ∈ [start, end + 2h] entonces: retornar False
    Para cada periodo en alcohol hacer:
        Extraer start_time, duration, amount, end_time
        Si amount >5 entonces:
            Si time ∈ [start, end + 30 × amount] entonces: retornar
False
    Retornar True
```

En el Pseudocódigo 4.4 se muestra la función que calcula el factor de eficacia de la insulina en función del tiempo transcurrido desde el último cambio de reservorio. Este factor se utiliza para ajustar la cantidad de insulina activa en el cuerpo, teniendo en cuenta la disminución de eficacia a medida que pasa el tiempo desde el último cambio.

Pseudocódigo 4.4 Cálculo del factor de eficacia.

```
Función efficacy_factor(current_time, config, events):
    duración ← config['reservoir']['value']
    start_time ← events['reservoir']['start']
    es periódico ← (events['reservoir']['periodic'] == 'Yes')
    Si es periódico entonces:
        calcular último cambio según días y ciclos
    Si no, last_change_time ← start_time
    days_since ← current_time - last_change_time
    Si days_since <duración entonces factor ← 1.0
    Si duración ≤ days_since ≤ días_entre_cambios entonces:
        factor ← 1.0 + incremento proporcional
    Si no: factor ← 1.0
    retornar factor
```

4.4. Insulina suministrada

4.4.1. Insulina basal

En este trabajo, se ha implementado un algoritmo PID para el suministro de la insulina basal, que tiene como objetivo mantener los niveles de glucosa en un rango óptimo de manera continua. El uso del controlador PID se basa en tres ganancias principales: proporcional (K_P), integral (K_I) y derivativa (K_D). Estas ganancias determinan la respuesta del sistema ante las variaciones en los niveles de glucosa, y se ajustan dinámicamente cada cierto periodo de tiempo para optimizar la administración de insulina.

4.4.1.1. Controlador P, I, D

Como ya se comentó en 2, el controlador proporcional (K_P) responde de manera directa a la diferencia entre el nivel de glucosa medido y el objetivo de glucosa (error). A mayor error, mayor será la corrección que aplica el controlador proporcional, proporcionando una corrección inmediata. Por otro lado, el controlador integral (K_I) se encarga de acumular el error a lo largo del tiempo, lo que permite corregir desajustes prolongados y mejorar el control en situaciones en las que el sistema se estabiliza lejos del objetivo. Finalmente, el controlador derivativo (K_D) calcula la tasa de cambio del error, ayudando a suavizar las correcciones y reduciendo las oscilaciones bruscas en la administración de insulina.

Desde la perspectiva de la ingeniería de control, la variable manipulada en este sistema es la insulina, mientras que la variable monitorizada es la glucosa en sangre. El controlador ajusta de forma continua la administración de insulina en función del error con respecto al valor objetivo, con el fin de mantener la glucosa dentro del rango deseado.

Aparte de los parámetros introducidos por el usuario mencionados anteriormente, existe otro fichero llamado params.json que contiene otros parámetros internos del algoritmo. Estos son:

- Ganancias KP [U/h / (mg/dL)]
- Ganancias KI [U/h / (mg/dL) / h]
- Ganancias KD [U/h / (mg/dL) * h]
- Basal por defecto [U/h]
- Frecuencia de ajuste de las ganancias del PID y de la basal por defecto [h]
- Fecha y hora de la última actualización de los parámetros anteriores
- Histórico con las fechas y horas, y las ganancias y basal por defecto.

El algoritmo sigue los siguientes pasos:

1. Lee inicialmente los parámetros de objetivo de glucosa (para calcular el error e) y de sensibilidad a la insulina, que son función de la hora del día (y de algún posible evento puntual descrito en el apartado anterior, que hagan ajustar estos parámetros)
2. Comprueba si es necesario actualizar las ganancias y la basal, en función de si se ha llegado a la hora indicada por la frecuencia de adaptación, y en ese caso, se actualizan los parámetros. En el siguiente apartado se explica esto.
3. Si no se han actualizado los parámetros, simplemente se leen tanto el parámetro de basal máxima como las ganancias actuales.
4. Se calcula la eficacia de la insulina activa en el cuerpo dependiendo de la fecha de cambio de reservorio y del parámetro de efecto de la insulina dependiendo del cambio de reservorio.
5. El error e es la diferencia entre la glucosa medida y el objetivo de glucosa. A dicho error se le da mayor peso si es negativo, ya que indica que hay riesgo de hipoglucemia, que es más peligroso en periodos cortos de tiempo.
6. Se calcula el error integral sumando a la variable el error. Este error integral está acotado para evitar hiperglucemias o hipoglucemias largas en el tiempo. Además, si la glucosa está en rango, se resetea este valor a 0.
7. El error derivativo es calculado haciendo una media de las diferencias en el error en los 5 anteriores pasos de tiempo, para suavizar posibles cambios o ruido que no sería representativo para un paso de tiempo puntual. Si esta velocidad del error es negativa y muy alta, indica que es una caída en la glucosa muy rápida, y no se suministra nada de basal.

8. Finalmente, la cantidad de basal es

$$\text{basal} = (Basal_{Defecto} + K_P \cdot \text{e} + K_I \cdot e_{\text{Integral}} + K_D \cdot e_{\text{Derivativo}}) \tag{4.1}$$

4.4.1.2. Ajuste dinámico de K_P, K_I, K_D

El algoritmo implementado no utiliza valores estáticos para las ganancias K_P, K_I y K_D. En su lugar, estas ganancias se ajustan dinámicamente a lo largo del tiempo en función la estabilidad de la glucosa del paciente y el tiempo en el rango de normoglucemia.

- Estabilidad: se calcula analizando las oscilaciones en los registros de glucosa. Para ello, se almacenan las diferencias entre los valores y se calcula la desviación estándar. Finalmente, se obtiene la estabilidad de acuerdo a:

$$\text{Stability} = \max(0, 1 - \frac{\sigma}{LIMIT_STD_DIFFS}) \tag{4.2}$$

  Donde LIMIT_STD_DIFFS es un parámetro que indica el límite máximo a partir del cual esas diferencias son suficientemente considerables y grandes como para asignar una estabilidad nula a la glucosa.

- Tiempo en rango: es el tiempo en el cual la glucosa permanece dentro de los límites establecidos para esta variable, que son 10 mg por encima y por debajo del objetivo.

Los ajustes dependen de cada ganancia:

- **Ajuste de** K_P: El parámetro K_P ajusta la rapidez de la respuesta del sistema en función del error. Un valor alto de K_P hace que el sistema responda de manera más agresiva a los errores grandes, mientras que un valor bajo reduce el riesgo de sobrecorrección.

  - Lógica de adaptación:
    - Incrementar K_P cuando el sistema responde lentamente a grandes errores (por ejemplo, hiperglucemias prolongadas) o cuando hay una falta de corrección en hiperglucemias persistentes.
    - Disminuir K_P cuando ocurren oscilaciones grandes, indicando sobrecorrección (por ejemplo, hipoglucemias repetidas) o cuando se observa inestabilidad, con la glucosa cruzando constantemente el rango objetivo.
  - Ajuste: Si la estabilidad es mayor al 80 % y el tiempo en rango es menor al 70 %, K_P se incrementa en un 10 %. Si la estabilidad es menor al 70 % y el tiempo en rango es mayor al 70 %, K_P se reduce en un 10 %.

- **Ajuste de** K_I: El parámetro K_I acumula el error a lo largo del tiempo, lo que ayuda a corregir desviaciones persistentes de bajo nivel (errores de estado estacionario). Un valor alto de K_I permite que el sistema corrija los desplazamientos a largo plazo, pero puede provocar acumulación excesiva si no se controla adecuadamente.

  - Lógica de adaptación:

- Incrementar K_I cuando se observan errores residuales persistentes o cuando la tasa basal actual es insuficiente para mantener la glucosa dentro del rango.
- Disminuir K_I cuando hay acumulación excesiva del término integral, lo que puede llevar a sobrecorrecciones (como hipoglucemia después de una hiperglucemia) o oscilaciones alrededor del objetivo debido a una sobrecompensación.

- Ajuste: Si el tiempo en rango es menor al 70 %, K_I se incrementa en un 5 %. Si el tiempo en rango es mayor al 90 %, K_I se reduce en un 5 %.

- **Ajuste de K_D**: El parámetro K_D responde a la tasa de cambio del error, actuando como un amortiguador para reducir las oscilaciones y el sobrepaso. Un valor alto de K_D ayuda a mitigar las variaciones rápidas, mientras que un valor bajo reduce la capacidad de freno.

  - Lógica de adaptación:
    - Incrementar K_D cuando ocurren oscilaciones rápidas debido a respuestas proporcionales demasiado agresivas o cuando el sistema no puede amortiguar eficazmente los cambios rápidos en los niveles de glucosa.
    - Disminuir K_D cuando el sistema se vuelve demasiado amortiguado, lo que retrasa la estabilización, o cuando el término derivativo no contribuye de manera significativa debido a variaciones lentas de glucosa.
  - Ajuste: Si la estabilidad es menor al 70 %, K_D se incrementa en un 10 %. Si la estabilidad es mayor al 90 %, K_D se reduce en un 10 %.

4.4.1.3. Pseudocódigo de funciones auxiliares de adaptación PID

A continuación se muestran los Pseudocódigos 4.5 y 4.6 de las funciones que calculan la insulina basal mediante un PID adaptativo. Estas funciones se encargan de ajustar los parámetros del PID y calcular la tasa de insulina basal en función de la glucosa medida, el error histórico y el tiempo transcurrido.

Pseudocódigo 4.5 Cálculo de la insulina basal mediante PID adaptativo (I).

```
Función basalDelivery(time, timeOld, glucose, error_history,
eIntegral):
/* Leer parámetros, configuración y eventos */
    params ← leer('data/params.json')
    config ← leer('data/config.json')
    events ← leer('data/events.json')
/* Obtener último ajuste y frecuencia de adaptación */
    last_update ← datetime de params['last_update']['value']
    adapt_freq ← horas de params['freq']['value']
/* Obtener valores nominales y ajustarlos con eventos */
    nom_targetBG ← get_period_value(config, 'target', time)
    nom_coFactor ← get_period_value(config, 'sensitivity', time)
    targetBG ← events_adjust(time, events, 'target', nom_targetBG)
/* Adaptar parámetros PID si corresponde */
    Si should_adapt_params(time, last_update, adapt_freq) entonces:
        adapt_pid_params(time, params, adapt_freq, targetBG)
/* Leer parámetros PID y tasa basal nominal */
    kP ← float(params['kP']['value'])
    kI ← float(params['kI']['value'])
    kD ← float(params['kD']['value'])
    bR ← float(params['bR']['value'])
    basalMax ← float(config['basal']['value'])
/* Calcular error actual */
    actualError ← glucose - targetBG
    error ← actualError × HIPO_WEIGHT si actualError <0, si no
actualError
Si timeOld es None entonces:
    añadir actualError a error_history
    retornar (0, 0, 0, 0)
/* Calcular intervalo de tiempo en horas */
    dt ← (time - timeOld) / 3600
/* Término integral con saturación */
    eIntegral += error × dt
    eIntegral ← limitado entre MIN_INTEGRAL_LIMIT y MAX_INTEGRAL_LIMIT
    Si glucose ∈ [targetBG - 10, targetBG + 30] entonces:
        eIntegral ← 0
```

Pseudocódigo 4.6 Cálculo de la insulina basal mediante PID adaptativo (II).

```
    /* Término derivativo */
        errorRate ← calcular_tasa_error(actualError, error_history, dt)
    Si error <WEIGHTED_ERROR_LIMIT entonces:
        retornar (0, 0, eIntegral, errorRate)
    Si errorRate / 60 <RAPID_DROP_THRESHOLD entonces:
        retornar (0, 0, eIntegral, errorRate)
    /* Controlador PID */
        basalRate ← bR + kP × error + kI × eIntegral + kD × errorRate
    /* Aplicar factor de eficacia del reservorio */
        efficacy ← efficacy_factor(time, config, events)
        basalRate ×= efficacy
        basalRate ← limitado entre 0 y basalMax
    Si basalRate >0 entonces:
        guardar basalRate con timestamp y duración dt
    /* Calcular unidades de insulina administradas */
        basal ← basalRate × dt
    Retornar (basal, basalRate, eIntegral, errorRate)
```

El Pseudocódigo 4.7 muestra la función que calcula la tasa de error. Esta función se encarga de almacenar el error en un historial, calcular la tasa de error promedio. También se incluye la función que determina si es necesario realizar ajustes en los parámetros del PID (que depende de la frecuencia de ajuste que haya configurada en config.json).

Pseudocódigo 4.7 Cálculo de la derivada del error y decisión de adaptación.

```
    Función calculate_error_rate(error, error_history, dt):
        añadir error a error_history
        Si longitud(error_history) >5 entonces:
            eliminar el primer elemento
        errorAvg ← media de (error[i] - error[i-1]) para i en 1..N
        errorRate ← errorAvg / dt
        retornar errorRate

    Función should_adapt_params(time, last_update, adapt_freq):
        retornar (time - last_update) ≥ adapt_freq
```

El Pseudocódigo 4.8 muestra la función que ajusta los parámetros del PID y la tasa basal. Esta función se encarga de leer los registros de glucosa y de insulina basal, filtrar los registros relevantes en función del tiempo actual y la frecuencia de ajuste, analizar la estabilidad de la glucosa y el tiempo en rango, y finalmente ajustar los parámetros del PID y la tasa basal según las condiciones actuales.

Pseudocódigo 4.8 Adaptación de parámetros PID.

```
Función adapt_pid_params(current_time, params, freq, target):
    glucose_records ← leer DATA_PATH
    basal_records ← leer INPUTS_PATH["basal"]
    start_time ← current_time - freq
    relevant_basal ← filtrar(basal_records, start_time, current_time)
    relevant_glucose ← filtrar(glucose_records, start_time,
current_time)
    stability ← analizar_estabilidad(relevant_glucose)
    tir ← analizar_tiempo_en_rango(relevant_glucose, target)
    Si tir <0 entonces retornar
    kP ← ajustar_kP(params["kP"]["value"], stability, tir)
    kI ← ajustar_kI(params["kI"]["value"], stability, tir)
    kD ← ajustar_kD(params["kD"]["value"], stability, tir)
    bR ← calcular_basal_rate(relevant_basal, params["bR"]["value"],
start_time, current_time)
    añadir a params["hist"]["period"]:
        start ← current_time
        kP, kI, kD, bR ← nuevos valores
    actualizar valores actuales en params
    params["last_update"]["value"] ← current_time
    escribir JSON en "data/params.json"
```

El Pseudocódigo 4.9 muestra la función que analiza el tiempo en rango de glucosa. Esta función calcula el porcentaje de tiempo que los registros de glucosa se encuentran dentro de un rango específico alrededor del objetivo, y retorna este porcentaje o -1 si no hay registros.

Pseudocódigo 4.9 Tiempo en rango.

```
Función analyze_time_in_range(glucose_records, target):
    range_lower ← target - 10
    range_upper ← target + 10
    contar registros con glucosa ∈ [range_lower, range_upper]
    Si total >0 entonces:
        tir ← 100 × (in_range / total)
    Si no: tir ← -1
    retornar tir
```

El Pseudocódigo 4.10 muestra la función que analiza la estabilidad de los registros de glucosa. Esta función calcula las diferencias sucesivas entre los registros de glucosa, calcula la desviación estándar de estas diferencias y retorna un valor de estabilidad basado en un límite predefinido.

Pseudocódigo 4.10 Análisis de estabilidad.

```
Función analyze_stability(glucose_records):
    Si número de registros <2 retornar 1
    differences ← diferencias sucesivas de glucosa
    std ← desviación estándar de differences
    stability ← max(0, 1 - std / LIMIT_STD_DIFFS)
    retornar stability
```

Los Pseudocódigos 4.11, 4.12 y 4.13 muestran las funciones que ajustan las ganancias K_P, K_I y K_D respectivamente. Estas funciones toman en cuenta la estabilidad de la glucosa y el tiempo en rango para determinar si se deben incrementar o reducir las ganancias.

Pseudocódigo 4.11 Ajuste de K_P.

```
Función adjust_kP(kP, stability, tir):
    Si stability >0.8 y tir <70 entonces retornar kP × 1.1
    Si stability <0.7 y tir >70 entonces retornar kP × 0.9
    retornar kP
```

Pseudocódigo 4.12 Ajuste de K_I.

```
Función adjust_kI(kI, stability, tir):
    Si tir <70 entonces retornar kI × 1.05
    Si tir >90 entonces retornar kI × 0.95
    retornar kI
```

Pseudocódigo 4.13 Ajuste de K_D.

```
Función adjust_kD(kD, stability, tir):
    Si stability <0.7 entonces retornar kD × 1.1
    Si stability >0.9 entonces retornar kD × 0.9
    retornar kD
```

El Pseudocódigo 4.14 muestra la función que calcula la tasa basal promedio. Esta función toma los registros de insulina basal, la tasa basal actual, y el tiempo de inicio y fin para calcular una tasa basal promedio ajustada a los límites establecidos.

Pseudocódigo 4.14 Cálculo de la tasa basal promedio.

```
Función calculate_basal_rate(basal_records, bR, start_time, end_time):
    Si basal_records está vacío entonces:
        mostrar advertencia
        retornar bR

/* Calcular insulina basal total (U) */
    total_basal ← suma de (iir × duración/60) para cada registro
/* Calcular tiempo cubierto en horas */
    time_covered ← (end_time - start_time) en horas
/* Calcular tasa basal media */
    average_basal_rate ← total_basal / time_covered

/* Limitar el cambio máximo de tasa basal */
    max_change ← LIMIT_BASAL_CHANGE × bR
    lower_limit ← bR - max_change
    upper_limit ← bR + max_change

/* Restringir dentro de los límites permitidos */
    average_basal_rate ← max(lower_limit, min(average_basal_rate,
upper_limit))
    retornar average_basal_rate
```

4.4.2. Bolo de corrección

En el proceso de cálculo de la insulina basal, se realiza una predicción de los niveles de glucosa futuros para determinar si es necesario administrar un bolo de corrección. Esta predicción evalúa si la glucosa en los próximos 30 minutos descenderá por debajo de 70 mg/dL. Si este

es el caso, el algoritmo detecta un riesgo de hipoglucemia y no administra un posible bolo de corrección (el algoritmo no evalúa si lo necesita), ya que se espera que la glucosa disminuya de manera natural.

Por ejemplo, si la glucosa es 150 mg/dL y el objetivo es 100 mg/dL, estaría bien administrar un bolo de corrección para disminuir la glucosa. Sin embargo, si se predice (gracias al error rate) que en media hora la glucosa va a ser menor de 70 mg/dL, no se va a suministrar ningún bolo de corrección, ya que la lógica de decisión se salta este paso.

Nótese que si se predice una hipoglucemia, es seguramente porque el ratio de cambio (error derivativo, siendo el error el objetivo menos la glucosa actual) es muy positivo, y ya se explicó que bajo esta condición tampoco se libera basal, ya que la tendencia de la glucosa ya es por sí sola a disminuir.

Si no se predice una hipoglucemia, el algoritmo evalúa si es necesario administrar un bolo de corrección siguiendo los siguientes pasos:

1. Se calcula la eficacia de la insulina en función de la fecha del último cambio de reservorio. Este cálculo tiene en cuenta si el reservorio ha sido cambiado recientemente o si ha pasado un tiempo suficiente (2.5 días) para que la eficacia de la insulina se vea reducida.

2. Se calcula el objetivo (objetivo) y la sensibilidad a la insulina en función de la hora del día. Estos parámetros pueden variar dependiendo del momento en que se efectúe el cálculo.

3. Se ajustan estos valores de acuerdo con los eventos que estén ocurriendo, como consumo de alcohol, práctica de ejercicio físico u otros factores que puedan afectar la sensibilidad a la insulina o los niveles de glucosa.

4. Se calcula el error de glucosa, es decir, la diferencia entre la glucosa actual y el valor objetivo (objetivo).

5. Este error, expresado en mg/dL, se divide entre el factor de sensibilidad a la insulina, que indica cuántos mg/dL de glucosa se reducen por cada unidad de insulina administrada. Este factor puede estar modificado por algún evento en curso, si corresponde.

6. A la cantidad de insulina calculada en el paso anterior se le resta la insulina activa en el cuerpo (teniendo en cuenta su eficacia, obtenida en el paso 1). Esto se debe a que la insulina ya presente aún puede estar haciendo efecto y, por tanto, es posible que no se necesite administrar el bolo completo.

7. Se ajusta la cantidad del bolo de corrección en función nuevamente de la eficacia de la insulina calculada previamente. Si la eficacia de la insulina es menor, será necesario aumentar la cantidad del bolo.

8. Finalmente, se verifica si hay algún evento que impida la administración del bolo de corrección. Esto incluye eventos como el ejercicio físico o el consumo de alcohol.

    - En el caso del ejercicio, si la duración es mayor a 60 minutos y la intensidad es baja, el bolo se suprime desde 3 horas antes de que comience el ejercicio hasta 2 horas después de que termine. Si la intensidad del ejercicio es alta, el bolo se suprime desde el inicio del ejercicio hasta 2 horas después de finalizar.

- En el caso del consumo de alcohol, si la cantidad es superior a 5 unidades, el bolo se suprime desde el inicio del consumo hasta 30 minutos multiplicados por la cantidad de alcohol ingerido.

4.4.2.1. Pseudocódigo para los bolos de corrección

El Pseudocódigo 4.15 muestra la función que calcula el bolo corrector. Esta función toma como entrada el tiempo actual, la glucosa actual, la insulina activa y la insulina basal, y realiza los cálculos necesarios para determinar si se debe administrar un bolo de corrección y, en caso afirmativo, cuánto debe ser.

Pseudocódigo 4.15 Cálculo del bolo corrector.

```
Función correctionBolus(currentTime, actualBG, activeInsulin, basal):
/* Cargar parámetros y calcular eficacia */
    config ← leer('data/config.json')
    events ← leer('data/events.json')
    efficacy ← efficacy_factor(currentTime, config, events)
    effective_activeInsulin ← activeInsulin / efficacy
    effective_basalInsulin ← basal / efficacy
/* Obtener valores nominales */
    nom_targetBG ← get_period_value(config, 'target', currentTime)
    nom_coFactor ← get_period_value(config, 'sensitivity',
currentTime)
    Si nom_targetBG = None o nom_coFactor = None entonces retornar
    targetBG ← events_adjust(currentTime, events, 'target',
nom_targetBG)
    coFactor ← events_adjust(currentTime, events, 'sensitivity',
nom_coFactor)
/* Calcular el error de glucosa */
    BGerror ← actualBG - targetBG
/* Calcular bolo corrector */
    correctionBolus ← BGerror / coFactor - (effective_activeInsulin +
effective_basalInsulin)
/* Aplicar eficacia */
    effective_correctionBolus ← redondear(correctionBolus × efficacy,
1)
/* Verificar si se puede aplicar la corrección */
    apply_correction ← events_apply(currentTime, events)
/* Guardar bolo si se permite y es significativo */
    Si apply_correction ∧ effective_correctionBolus ≥ 0.1 entonces:
        guardar_bolo(currentTime, effective_correctionBolus)
```

4.4.3. Bolos por comida

Cuando el paciente indica que va a comer, introduce el número de gramos de hidratos de carbono (HC) y, opcionalmente, el número de UGPs (Unidades Grasa-Proteína). Si el paciente no introduce UGPs, se asume que son 0, ya que es posible que aún no esté familiarizado con este concepto, ya que las bombas de insulina actuales no lo contemplan. Tanto los hidratos de carbono como los UGPs siguen una lógica que determina la cantidad de insulina que se debe administrar en ese momento, y si es necesario, en el futuro. Además, se realiza un ajuste basado en la glucosa actual del paciente y si tiene insulina activa en el cuerpo.

Los pasos a seguir para calcular el bolo de comida son los siguientes:

1. Se calcula la eficacia de la insulina en función del tiempo transcurrido desde el último cambio de reservorio. Si han pasado más de 2.5 días desde el cambio, la insulina puede haber perdido parte de su efectividad.
2. Se determina si se necesita un ajuste y la cantidad de insulina correspondiente, siguiendo los pasos explicados en la sección anterior de bolo de corrección.
3. Se calcula el ratio de hidratos de carbono (HC) en función de la hora del día. Este ratio varía y es más alto o más bajo dependiendo del momento en que el paciente ingiera los alimentos.
4. Se calcula el bolo de la comida dividiendo la cantidad de gramos de HC entre el ratio de hidratos de carbono calculado en el paso anterior. Además, se tiene en cuenta la eficacia de la insulina para aumentar ligeramente la cantidad de insulina si es necesario, en caso de que haya perdido efectividad.
5. Se ajusta la cantidad calculada en el paso 4 sumando o restando el bolo de corrección, de acuerdo con los niveles de glucosa actuales y la insulina activa en el cuerpo.
6. Se calculan los bolos debidos a las UGPs, explicados a continuación.

4.4.3.1. Bolos por UGPs

Tras la administración del bolo inicial, se calculan bolos futuros debidos a los UGPs. Estos bolos se administran de forma escalonada debido a la naturaleza de la absorción de grasas y proteínas, que es más lenta y prolongada en comparación con los hidratos de carbono. A continuación, se explica la lógica detrás de estos bolos:

Los UGPs se absorben de manera gradual, distribuyendo la digestión en varias horas. La duración total de la absorción se incrementa en una hora por cada UGP adicional. El bolo correspondiente a los UGPs se divide proporcionalmente entre las unidades indicadas por el paciente. Cada UGP genera una pequeña cantidad de insulina que se administra en momentos específicos en el futuro, distribuyendo la dosis de insulina de manera adecuada a lo largo del tiempo. El tiempo de administración de cada bolo de UGP se calcula añadiendo un incremento de horas a partir del momento de la comida. Para 1 UGP, la duración de absorción es de 3 horas, y para cada UGP adicional, se incrementa una hora. La razón por la que estos bolos se administran

en el futuro es porque los alimentos ricos en grasas y proteínas tienen un impacto más lento en los niveles de glucosa, lo que podría llevar a hiperglucemias tardías si no se administra la insulina de manera adecuada y escalonada.

4.4.3.2. Pseudocódigo para los bolos por comidas

El Pseudocódigo 4.16 muestra la función que calcula el bolo por ingesta de comida. Esta función toma como entrada los gramos de hidratos de carbono, el número de UGPs, el tiempo actual, la glucosa actual, la insulina activa y un indicador de si se debe aplicar la insulina activa.

Pseudocódigo 4.16 Cálculo del bolo por ingesta de comida.

```
Función mealBolus(ch_value, ugp_value, currentTime, actualBG,
activeInsulin, apply_activeInsulin):
/* Cargar configuración y calcular eficacia */
    config ← leer('data/config.json')
    events ← leer('data/events.json')
    efficacy ← efficacy_factor(currentTime, config, events)
    adjustment ← adjustMeal(currentTime, actualBG, activeInsulin,
apply_activeInsulin)
    ratio ← get_period_value(config, 'ratio', currentTime)
/* Calcular bolo inicial */
    result ← redondear(ch_value / ratio × efficacy, 1)
/* Limitar ajuste para no superar el bolo total */
    Si |adjustment| >result entonces: adjustment ← -result
/* Calcular bolo total final */
    bolus ← redondear(result + adjustment, 1)
/* Bolo retardado para UGPs */
    ugps ← lista vacía
    duración ← 3 horas
    Para ugp_num ∈ [1, ugp_value] hacer:
        ugp_bolus ← redondear(result / ugp_value, 1)
        ugp_time ← currentTime + duración
        añadir {bolus, time, num} a ugps
        duración += 1
    Retornar result, adjustment, bolus, ugps
```

El Pseudocódigo 4.17 muestra la función que ajusta el bolo por corrección de glucosa precomida. Esta función toma como entrada el tiempo actual, la glucosa actual, la insulina activa y un indicador de si se debe aplicar la insulina activa. Realiza los cálculos necesarios para determinar el ajuste del bolo basado en la glucosa actual y otros parámetros.

Pseudocódigo 4.17 Ajuste del bolo por corrección de glucosa precomida.

```
Función adjustMeal(currentTime, actualBG, activeInsulin,
apply_activeInsulin):
/* Cargar configuración y calcular eficacia */
    config ← leer('data/config.json')
    events ← leer('data/events.json')
    efficacy ← efficacy_factor(currentTime, config, events)
/* Determinar insulina activa a aplicar */
    Si apply_activeInsulin entonces:
        effective ← activeInsulin / efficacy
    Si no: effective ← 0
    target ← get_period_value(config, 'target', currentTime)
    sensitivity ← get_period_value(config, 'sensitivity',
currentTime)
    Si target = None o sensitivity = None entonces retornar
/* Ajustar con eventos */
    target ← events_adjust(currentTime, events, 'target', target)
    sensitivity ← events_adjust(currentTime, events, 'sensitivity',
sensitivity)
/* Calcular bolo corrector */
    BGerror ← actualBG - target
    correction ← BGerror / sensitivity - effective
    adjustment ← redondear(correction × efficacy, 1)
    Retornar adjustment
```

4.5. Arquitectura del algoritmo

En esta sección se comenta la arquitectura del algoritmo de la bomba, así como dependencias entre módulos y librerías externas. En la Figura 5.5 del Capítulo 5, se presenta un diagrama completo de los diferentes módulos implicados ya integrando el algoritmo de la bomba de insulina con la interfaz gráfica de usuario.

La mayor parte del algoritmo de la bomba de insulina se encuentra en el grupo (carpeta) `pump`. El detalle de la arquitectura de pump, sacado de la Figura 5.5 se puede ver en la Figura 4.1.

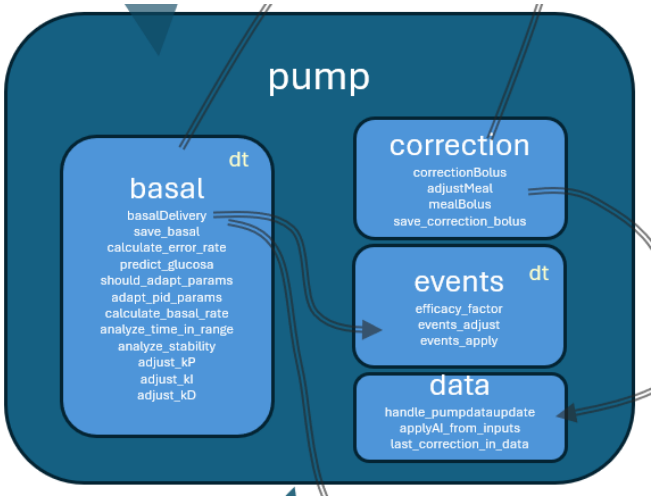


Figura 4.1: Detalle de pump.

- El módulo o script `correction` contiene los métodos siguientes. Se puede ver un esquema de éste en la Figura 4.2:

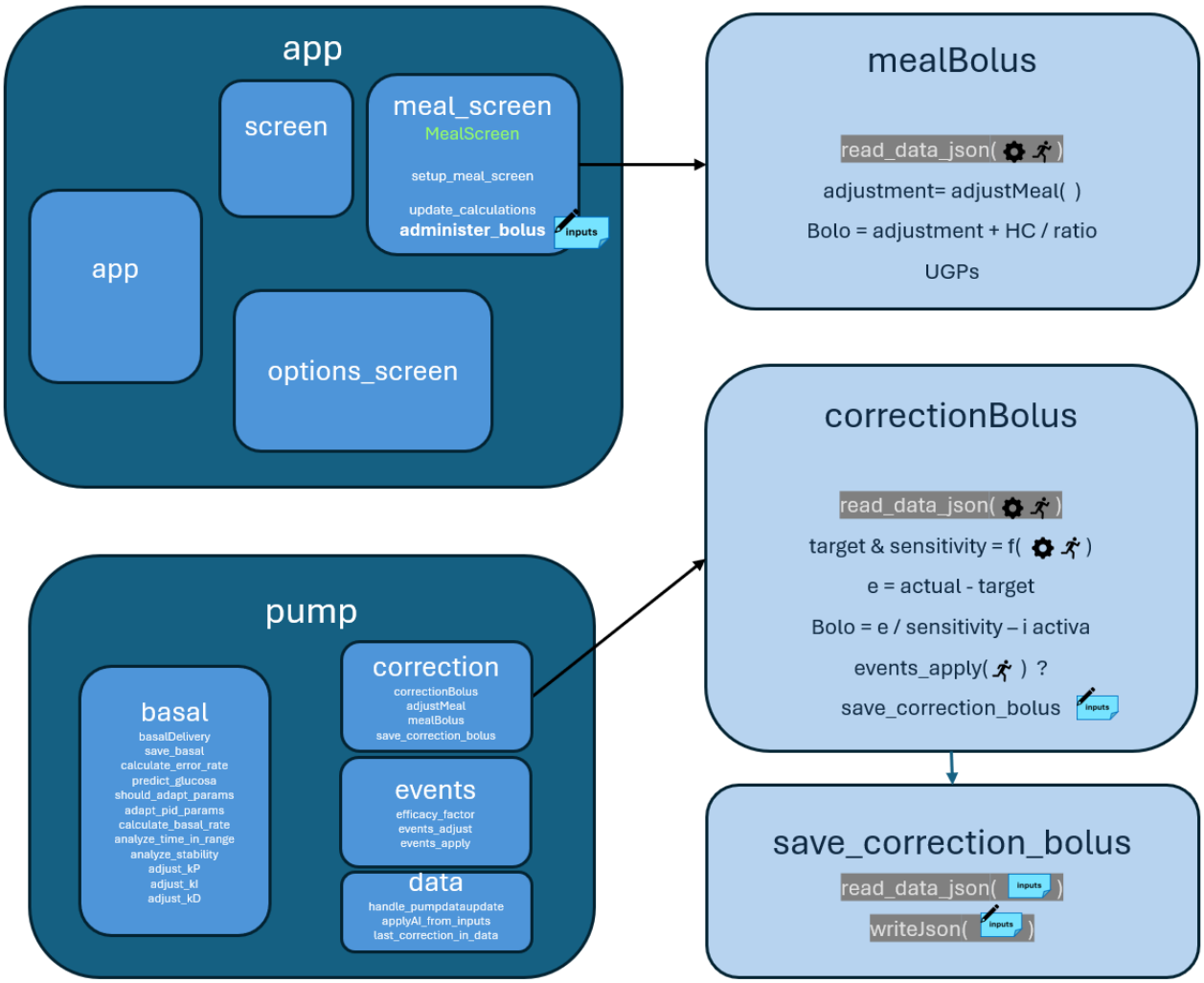


Figura 4.2: Detalle de correction.

1. `correctionBolus`: que calcula y escribe en `inputs.json`, si procede, el bolo de corrección. Dicho bolo depende de los eventos (deporte, alcohol...) y de variables de configuración (periodicidad de cambio de reservorio para calcular la eficacia de la insulina, targte, sensibilidad a la insulina...). Estos son leídos con `read_data_json`, ubicado en el módulo util. Se calculan el objetivo y la sensibilidad en función de estos datos. Después, se calcula el bolo, y si la configuración de los eventos indica que se puede aplicar el evento (con events_apply), se escribe el bolo con `save_correction_bolus`.
2. `save_correction_bolus`: lee inputs.json, genera una entrada json con el bolo a añadir, y escribe el json.
3. `mealBolus`: igual que en `correctionBolus`, se leen los eventos y la configuración. Se llama a `correctionBolus` para calcular el ajuste si fuese necesario, y se calcula el bolo con los HCs y el ratio de HC. También se generan los diccionarios de UGP que servirán para añadir bolos en el futuro. Este módulo, al contrario que en los casos anteriores, no escribe directamente en nputs.json. Quien se encarga de esto es el método administrer_bolus de la instancia que se haya generado de MealScreen. El módulo de `meal_screen` se explica en el Capítulo 5, ya que forma parte de la generación de la GUI para la Raspberry Pi.

- El módulo `basal` está representado en la Figura 4.3. Los métodos más importantes son:

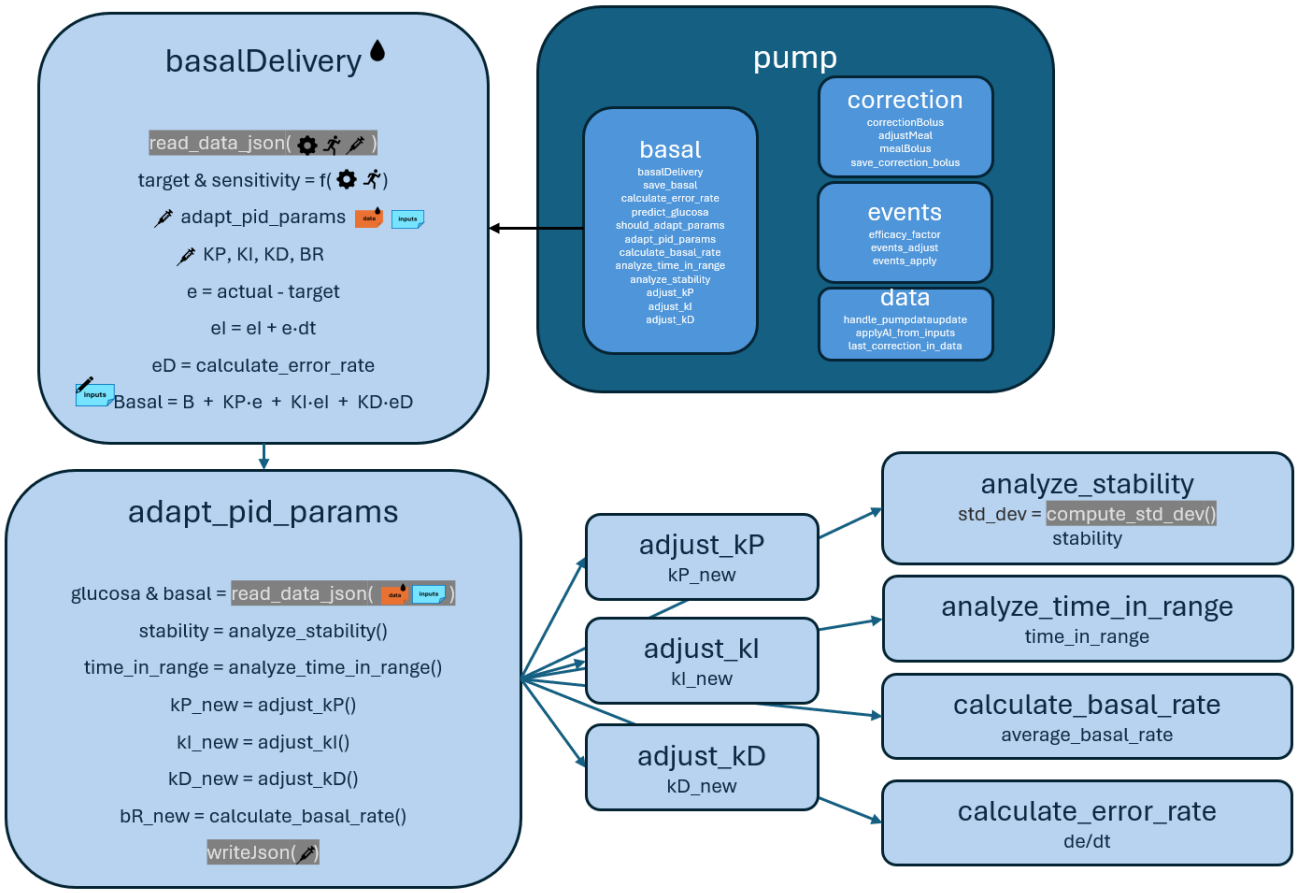


Figura 4.3: Detalle de basal.

1. `basalDelivery`: calcula la basal y la escribe en el archivo `inputs.json`. Para ello, lee los eventos, configuración y parámetros con `read_data_json`. Calcula el objetivo y la sensibilidad. Calcula los nuevos parámetros del PID si fuera necesario, así como el nuevo basal rate (BR). Calcula el error, el error derivativo y el error integral, y aplica la fórmula de controlador PID.
2. `adapt_pid_params`: calcula los nuevos parámetros del PID y BR. Para ello, lee los últimos datos de basales suministradas de inputs.json, y los últimos datos de glucosa de `data.json`. Con ellos calcula la estabilidad (`analyze_stability`) y el tiempo en rango `time_in_range`, para obtener los parámetros PID (`adjust_KP/I/D`). Con las basales de `inputs.json` se calcula la nueva basal rate (`calculate_basal_rate`).

- El módulo `events` contiene los métodos necesarios para calcular la efectividad de la insulina, ajustar el objetivo y la sensibilidad, y evaluar si se puede aplicar el bolo de corrección.
- El módulo `data` lee los datos de `data.json` para obtener la última glucosa disponible.

4.6. Ejemplos de uso y validación

Se va a probar el algoritmo mediante simulaciones en un entorno de desarrollo en el ordenador, donde se modela el comportamiento de la glucosa en función de diferentes ingestas de carbohidratos y dosis de insulina administradas. Durante esta fase, se comparará la cantidad de insulina calculada por el algoritmo con la cantidad que debería ser la correcta, según modelos estándar de gestión de la diabetes. Esta comparación ayudará a identificar y corregir posibles errores en el algoritmo, asegurando que las decisiones de administración de insulina sean razonablemente precisas antes de pasar a la validación con datos reales en la fase 5.

El módulo de DT de Dynamic simulator envía datos continuamente a la velocidad deseada, los cuales son recibidos por el algoritmo de la bomba de insulina para la generación de datos de insulina, tanto bolos de corrección y de comida, como insulina basal.

Se van a utilizar ejemplos parecidos a los de la Sección 3.5, pero ahora los inputs de insulina que se muestren en las figuras son generados por el algoritmo de la bomba. Además, los ejemplos utilizados para esta primera fase de validación representan casos de uso de la interfaz gráfica de ususario de la boba de insulina en la Raspberry Pi, por lo que muchos serán comentados nuevamente en la Sección 5.5 del próximo capítulo.

4.6.1. Simulación sin eventos

En esta sección se presentan varios casos de simulaciones de paciente diabético bajo ciertas circunstancias (comida, cambio de parámetros, eventos...) y la respuesta del algoritmo de la bomba de insulina. El modelo proporcionará los datos de glucosa e insulina activa en el cuerpo, y el algoritmo de la bomba generará bolos de insulina y basales que automáticamente son inyectadas al modelo.

Las únicas entradas manuales para el modelo de diabetes son las ingestas de comidas, ya que las entradas de insulina son introducidos por la bomba. Las únicas entradas manuales para la bomba de insulina son los HC que el paciente indica que va a comer. Nótese que no tienen por qué coincidir ni la cantidad de HC ni la hora de la ingesta de alimentos en el modelo y en la bomba.

La evolución de la glucosa y variables se mostrará con plots obtenidos con DiabetesTool. Se van a realizar simulaciones dinámicas aceleradas (para poder realizar varias simulaciones sin demorar demasiado tiempo en ellas) a la vez que el algoritmo de la bomba de insulina va a estar trabajando de manera sincronizada con la simulación. Por añadir claridad a estas simulaciones y a las respuestas que proporciona el algoritmo de la bomba de insulina, se van a incluir imágenes de la GUI de la bomba de insulina que utiliza este algoritmo, aunque la integración del algoritmo con una interfaz gráfica es objeto del Capítulo 5.

4.6.1.1. Comidas

Sin UGPs

En esta sección se analiza la respuesta del sistema en la administración de insulina sin la presencia de Unidades Grasa-Proteína (UGPs). La Figura 4.4 muestra la evolución de la glucosa, la insulina y los carbohidratos en un primer caso sin UGPs, al principio de la simulación.

En este caso, se indica a la bomba que se van a ingerir 30 g de HC justo antes de que realmente se ingiera (a las 15:00 h). Como el ratio de HC son 8.5 g/U, entonces 30 g / 8.5 g/U = 3.5 U.

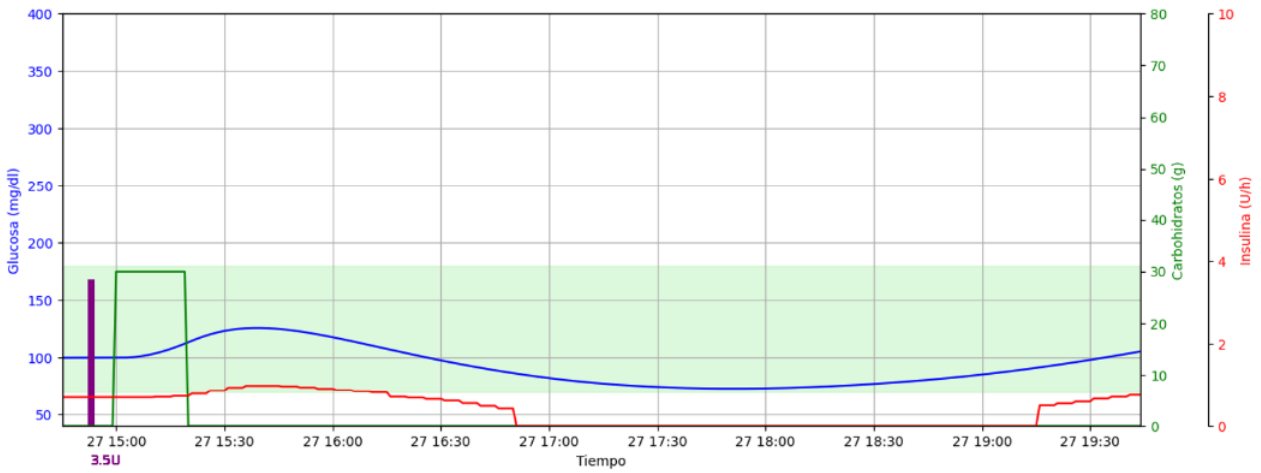


Figura 4.4: Glucosa, insulina y carbohidratos para el caso de comida sin UGPs I.

A continuación, la Figura 4.5 detalla lo que se vería en la bomba de insulina en un momento específico de la simulación, que son los 105 mg/dL a las 19:47 h, con 0 U de insulina activa (ya que ya ha pasado más de 2 horas desde el bolo de la comida) y una basal de 0.78 U/h.

La Figura 4.6 muestra el mismo ejemplo de simulación pero pasadas unas horas. No hay bolos adicionales por comida y como hay una zona de valle donde no hay basal, al final la glucosa comienza a aumentar y el algoritmo libera varios bolos de corrección de 0.1 U.

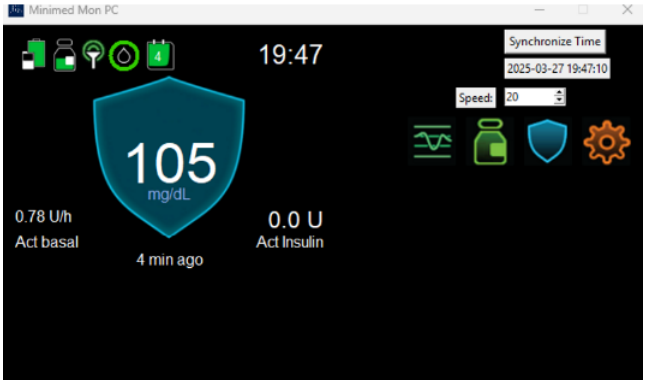


Figura 4.5: Detalle de la bomba de insulina en el momento de la Figura 4.4.

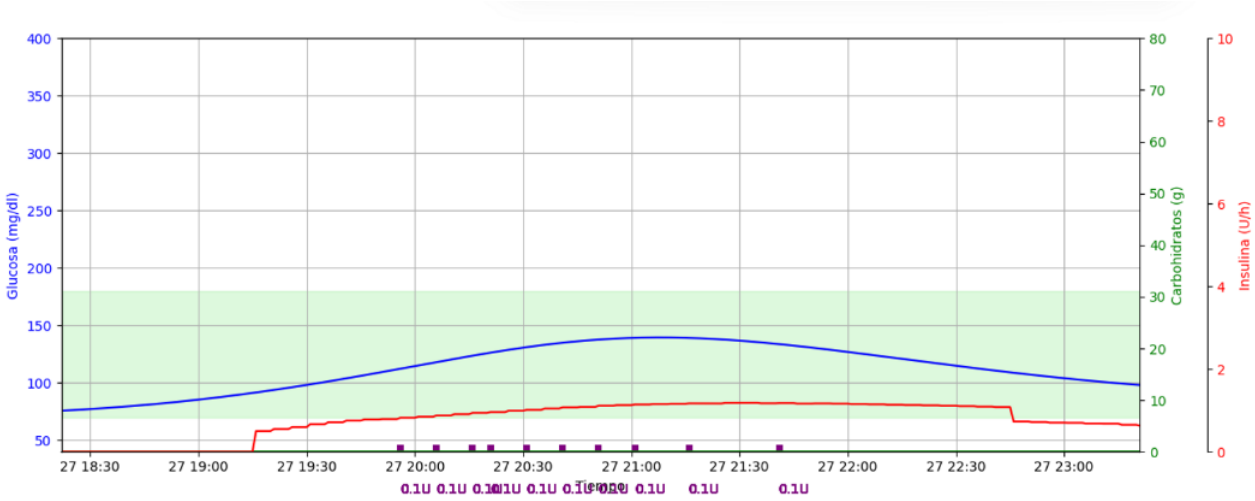


Figura 4.6: Glucosa, insulina y carbohidratos para el caso de comida sin UGPs II.

El detalle del comportamiento de la bomba en este caso se puede ver en la Figura 4.7.

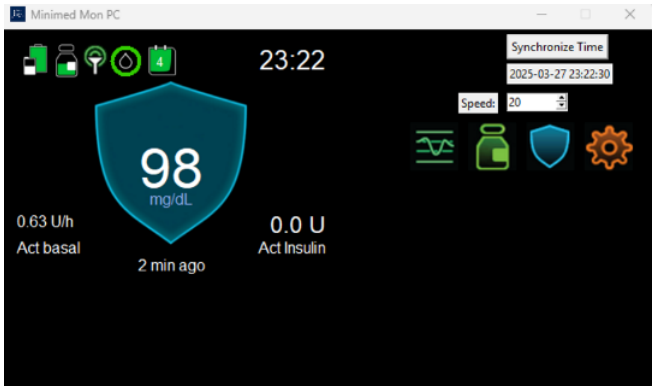


Figura 4.7: Detalle de la bomba de insulina en el momento de la Figura 4.6.

Para obtener una visión más completa, la Figura 4.8 muestra la evolución completa de la glucosa, la insulina y los carbohidratos a lo largo del tiempo. Finalmente, en la Figura 4.9, se presentan las variables de estado para todo el periodo de simulación, lo que permite un análisis más detallado del comportamiento del sistema.

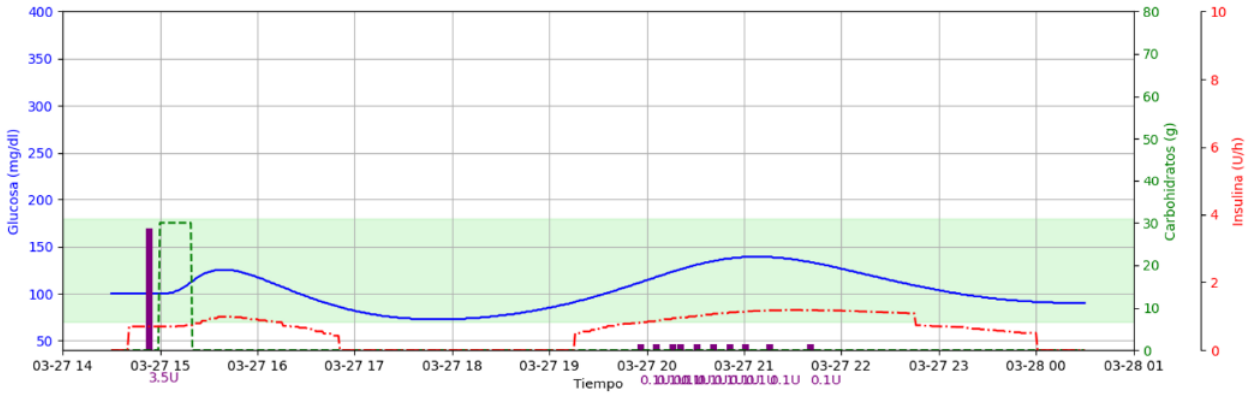


Figura 4.8: Evolución completa de glucosa, insulina y carbohidratos para el caso de comida sin UGPs.

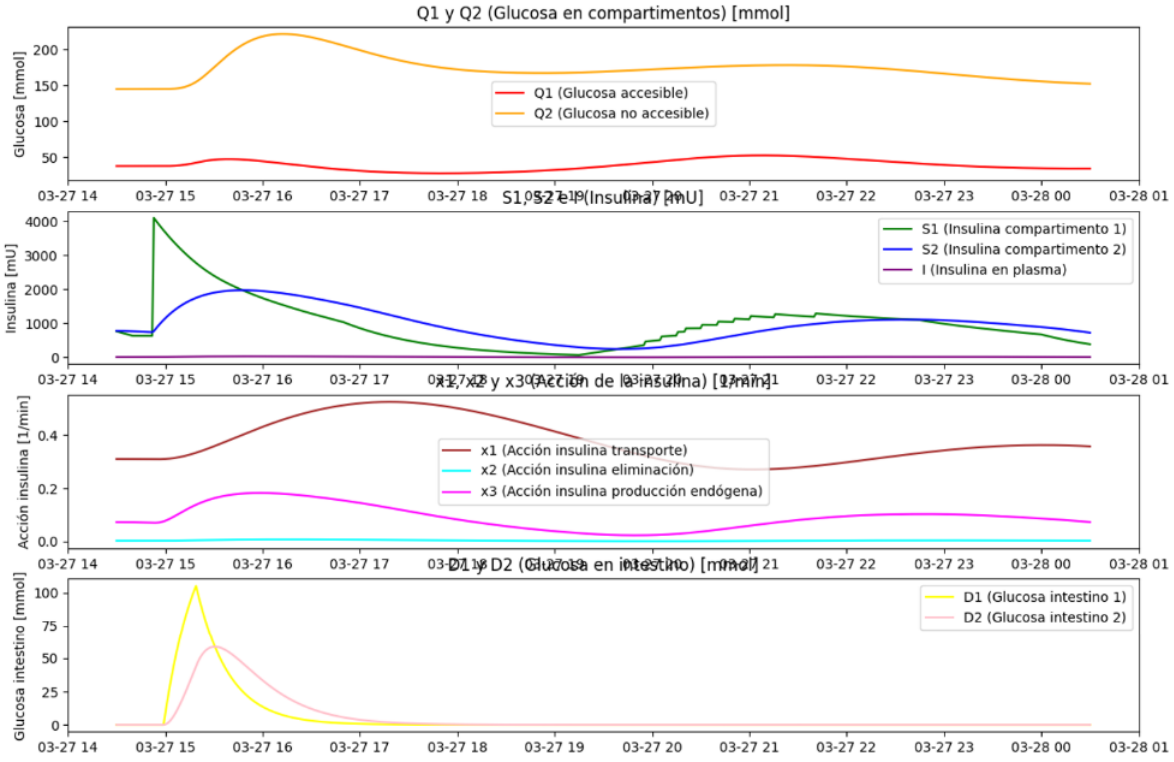


Figura 4.9: Evolución completa de variables de estado para el caso de comida sin UGPs.

Con UGPs

Cuando se incluyen UGPs en la simulación, el comportamiento del sistema cambia significativamente pasadas unas horas tras la ingesta de comida. La Figura 4.10 muestra la pantalla de configuración de las comidas con UGPs. Bajo el mismo ratio que antes, se ingieren 30 g de HC con 4 UGPs. Adicionalmente a las 3.5 U de insulina que se liberan en el momento, se calculan otros 4 bolos de 0.9 U ($0.9 \cdot 4 = 3.6$ U de UGPs) que serán liberados a partir de 3 horas progresivamente.

A partir de esta configuración, se presentan los resultados en la Figura 4.11, donde se observa

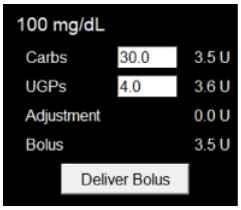


Figura 4.10: Pantalla de comida para carbohidratos con UGPs.

la evolución de la glucosa, la insulina y los carbohidratos al inicio de la simulación.

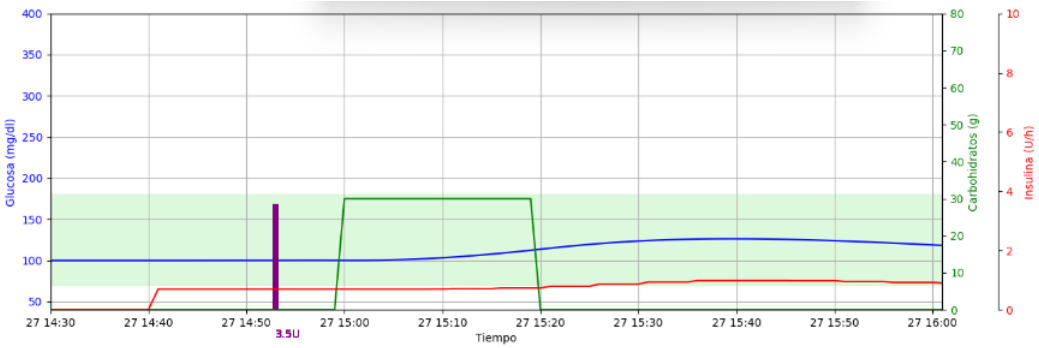


Figura 4.11: Glucosa, insulina y carbohidratos para el caso de comida con UGPs I.

El detalle de la bomba de insulina en el momento de la Figura 4.11 se ilustra en la Figura 4.12. De las 3.5 U de insulina administradas ya solo quedan activas en el cuerpo 1.5U. La basal es de 0.87 U/h y la glucosa 118 mg/dL.

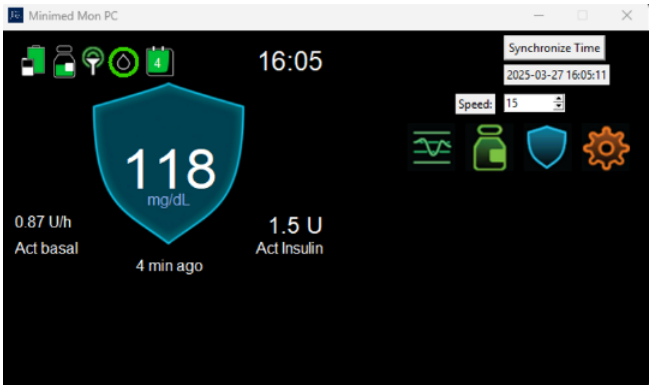


Figura 4.12: Detalle de la bomba de insulina en el momento de la Figura 4.11.

La evolución continúa en las Figuras 4.13 y 4.14, donde ya no queda insulina activa y no se esta liberando basal por estar por debajo del objetivo de insulina de 100 mg/dL.

La simulación continúa con dos nuevas gráficas, la Figura 4.15 y la Figura 4.16, que muestran el inicio de los bolos debidos a las UGPs.

En la Figura 4.17, se detalla la pantalla de la bomba en el instante mostrado en la Figura 4.17.

Finalmente, para obtener una visión global del comportamiento del sistema con UGPs, las Figuras 4.18 y 4.19 presentan la evolución completa de la glucosa, la insulina, los carbohidratos y las variables de estado a lo largo del tiempo.

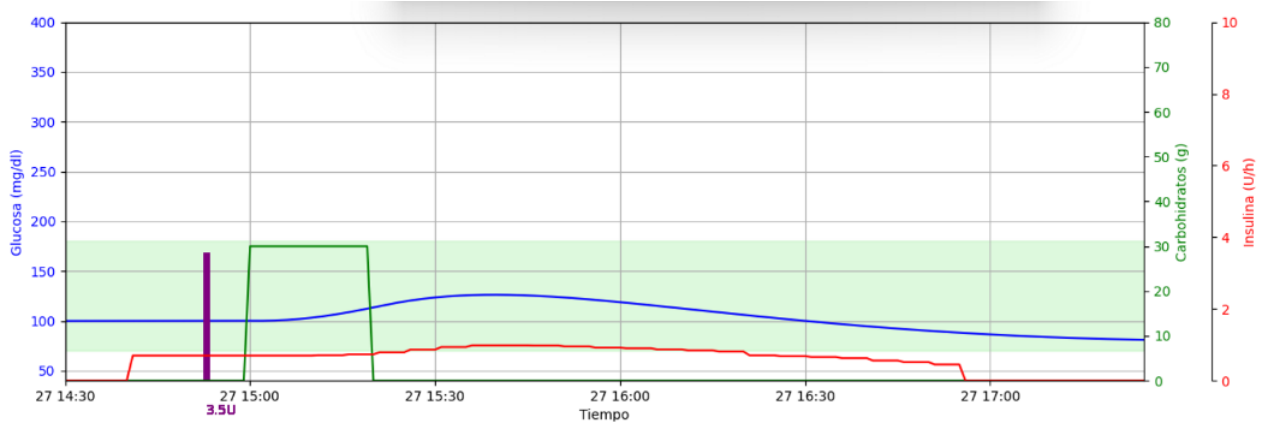


Figura 4.13: Glucosa, insulina y carbohidratos para el caso de comida con UGPs II.

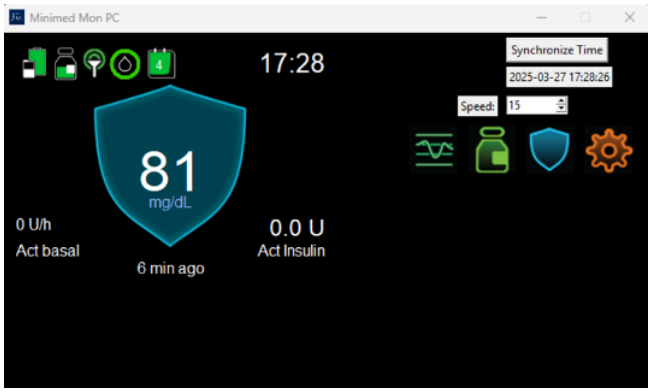


Figura 4.14: Detalle de la bomba de insulina en el momento de la Figura 4.13.

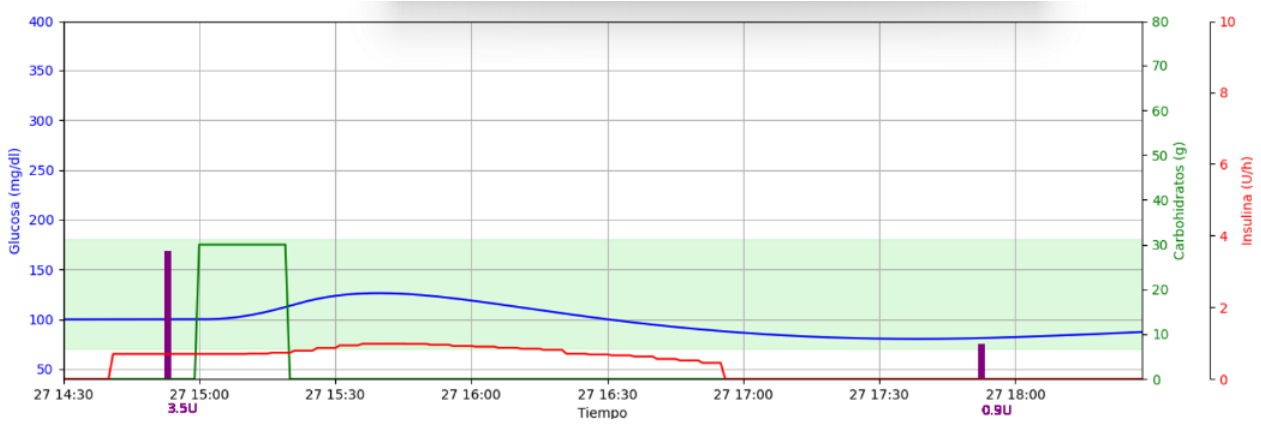


Figura 4.15: Glucosa, insulina y carbohidratos para el caso de comida con UGPs III.

4.6.1.2. Correcciones automáticas

En este apartado, se analiza el comportamiento del sistema cuando se aplican correcciones automáticas para mantener los niveles de glucosa cerca del objetivo. Se inicia la simulación con una ingesta de 15 g de HC sin inyectar insulina para que se calcule un bolo de corrección al momento de empezar a ejecutarse el algoritmo de insulina, que comienza unos 30 minutos después de haber comido.

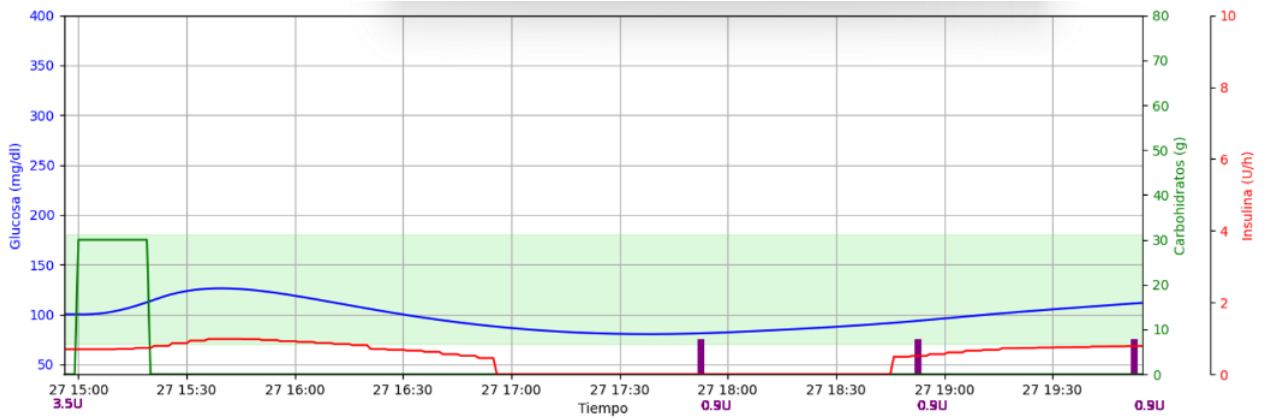


Figura 4.16: Glucosa, insulina y carbohidratos para el caso de comida con UGPs IV.

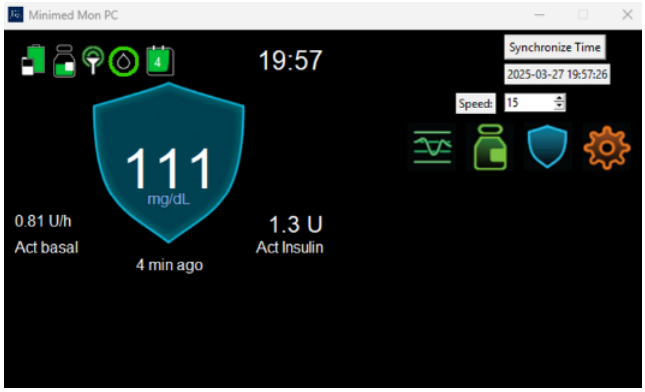


Figura 4.17: Detalle de la bomba de insulina en el momento de la Figura 4.16.

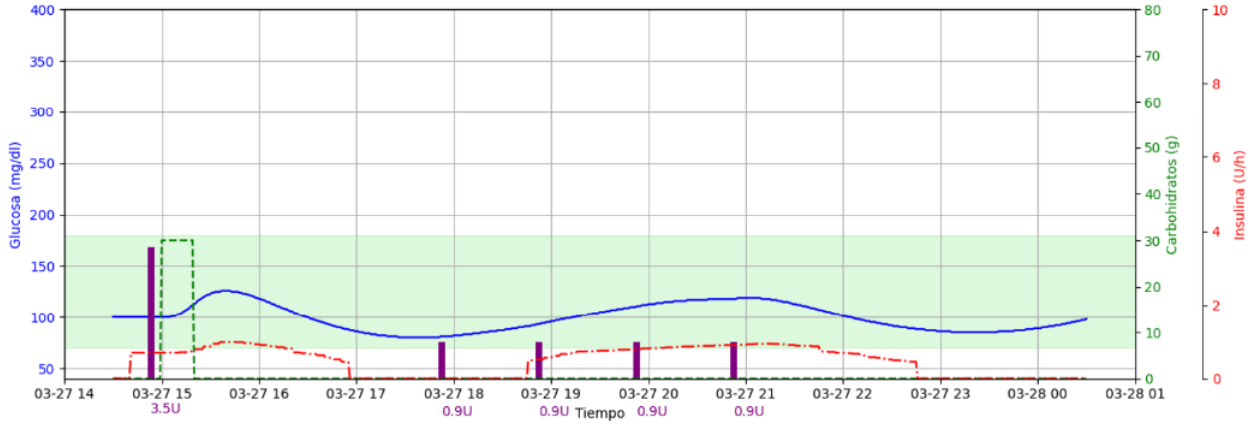


Figura 4.18: Evolución completa de glucosa, insulina y carbohidratos para el caso de comida con UGPs.

La Figura 4.20 muestra la evolución de la glucosa, la insulina y los carbohidratos en el primer caso de correcciones automáticas. Sobre las 12:45 comienza el algoritmo de la bomba de insulina (porque se ha decidido arrancarlo después, para dejar que la glucosa evolucionase aumentando) y calcula una corrección de 0.4 U de insulina.

La corrección se ha calculado así (datos obtenidos de las trazas del algoritmo), teniendo en cuenta que el objetivo es 100 mg/dL y la sensibilidad a la insulina es 60.0 mg/dL / U:

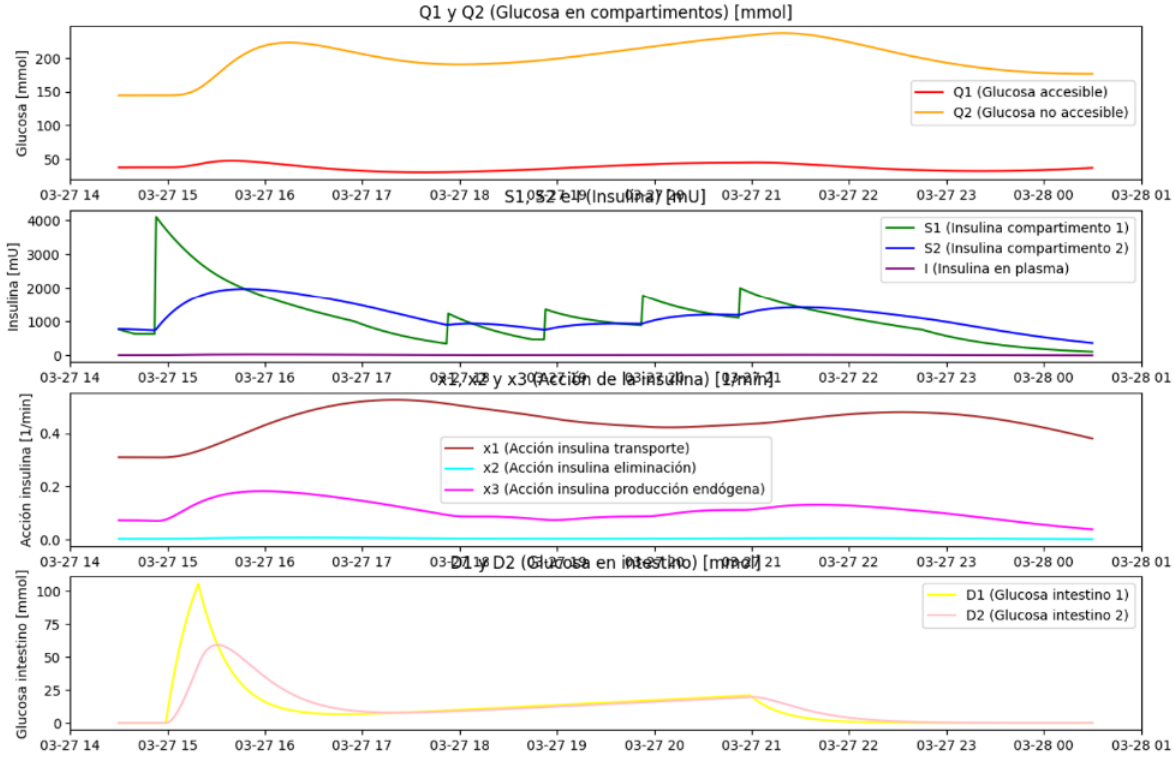


Figura 4.19: Evolución completa de variables de estado para el caso de comida con UGPs.

```
1  ****************Correction bolus calculation*************:
2  -> Current BG = 128 mg/dL
3  -> Nominal Target BG = 100 mg/dL
4  -> Target BG = 100 mg/dL
5  -> Active Insulin = 0.0000 U
6  -> Effective Active Insulin = 0.0000 U
7  -> Basal Insulin = 0.08 U
8  -> Effective Basal Insulin = 0.0800 U
9  -> Correction bolus = 28 mg/dL / (60.0 mg/dL/U) - (0.0000 U) - (0.08 U
       ) = 0.3867 U
10 -> Apply correction: True
11 -> Effective correction bolus: 0.4000
12 Correction bolus saved: {'start': '2025-03-27T12:44:11', 'i': 0.4, '
       type': 'Correction'}
```

Código 4.2: Correcciones automáticas (I).

También comienza a liberar basal y sigue con correcciones de 0.1 U. Por ejemplo, para el bolo de corrección de las 13:09 h:

```
1  ****************Correction bolus calculation*************:
2  -> Current BG = 141 mg/dL
3  -> Nominal Target BG = 100 mg/dL
4  -> Target BG = 100 mg/dL
5  -> Active Insulin = 0.5076 U
6  -> Effective Active Insulin = 0.5076 U
7  -> Basal Insulin = 0.09 U
8  -> Effective Basal Insulin = 0.0900 U
```

```
-> Correction bolus = 41 mg/dL / (60.0 mg/dL/U) - (0.5076 U) - (0.09 U
   ) = 0.0858 U
-> Apply correction: True
-> Effective correction bolus: 0.1000
Correction bolus saved: {'start': '2025-03-27T13:09:11', 'i': 0.1, '
   type': 'Correction'}
```

Código 4.3: Correcciones automáticas (II).

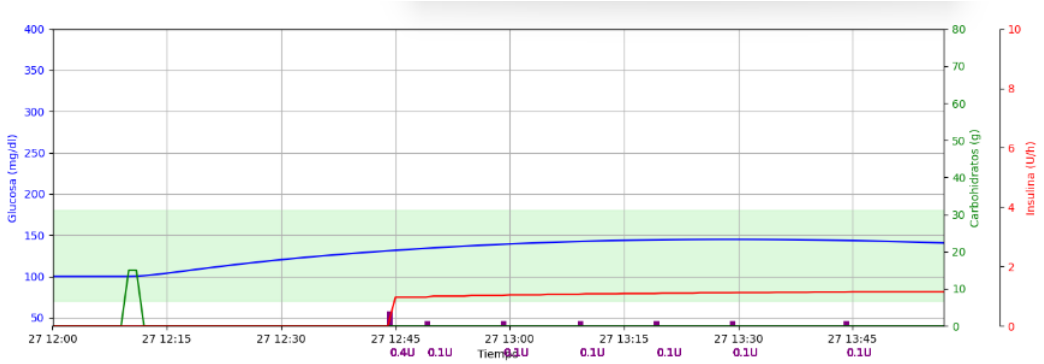


Figura 4.20: Glucosa, insulina y carbohidratos para el caso de correcciones automáticas I.

La Figura 4.21 detalla la pantalla de la bomba.

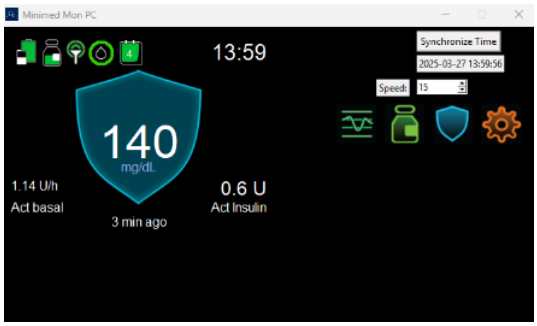


Figura 4.21: Detalle de la bomba de insulina en el momento de la Figura 4.20.

Continuando con la simulación, sobre las 19:10 h se ingieren de nuevo 15 g de HC sin bolo de insulina asociado, para ver cómo lo corregiría el algoritmo de la bomba de insulina. Se muestra en la Figura 4.22, y su detalle en la bomba de insulina se observa en la Figura 4.23.

Finalmente, para visualizar el comportamiento completo del sistema bajo este esquema de correcciones, la Figura 4.24 presenta la evolución general de los parámetros, mientras que la Figura 4.25 ilustra la evolución de las variables de estado.

4.6.1.3. Basal y ajuste de los parámetros PID

El control basal mediante un sistema de regulación PID es crucial para la estabilidad de la glucosa. En esta simulación no se han ingerido hidratos de carbono, ya que simplemente se quería observar el comportamiento del controlador PID del algoritmo de la bomba de insulina.

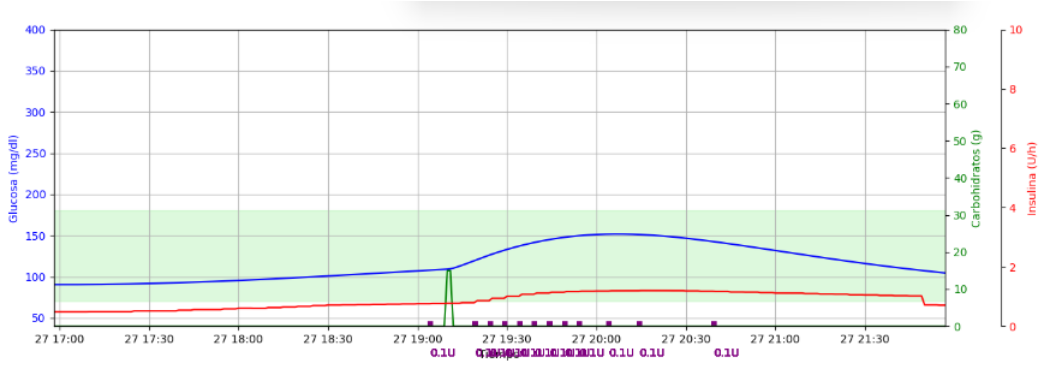


Figura 4.22: Glucosa, insulina y carbohidratos para el caso de correcciones automáticas II.

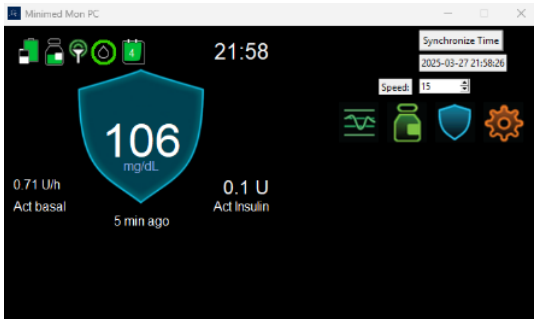


Figura 4.23: Detalle de la bomba de insulina en el momento de la Figura 4.22.

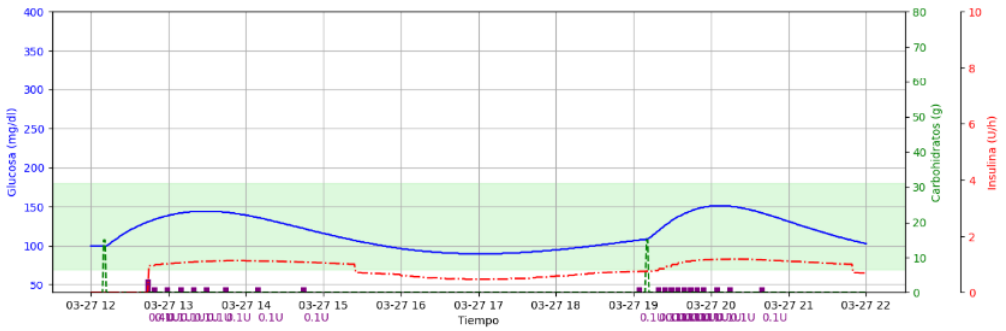


Figura 4.24: Evolución completa de glucosa, insulina y carbohidratos para el caso de correcciones automáticas.

Para ello, se inicia una simulación con una basal por defecto de 0.5 U/h, que es algo menor de la basal de equilibrio. Se ha elegido una basal menor para ver cómo el controlador va aumentando la cantidad de basal en función de sus ganancias. Además, se ha indicado que la actualización tanto de las ganancias como de la basal por defecto se realice cada 2 horas.

En la Figura 4.26, se muestra la evolución completa de la glucosa e insulina bajo este esquema de control.

Las ganancias PID, errores, basales y cálculos para determinados momentos de la simulación se listan a continuación. Se distinguen 4 tramos donde los parámetros son diferentes, ya que la simulación es de 8 horas:

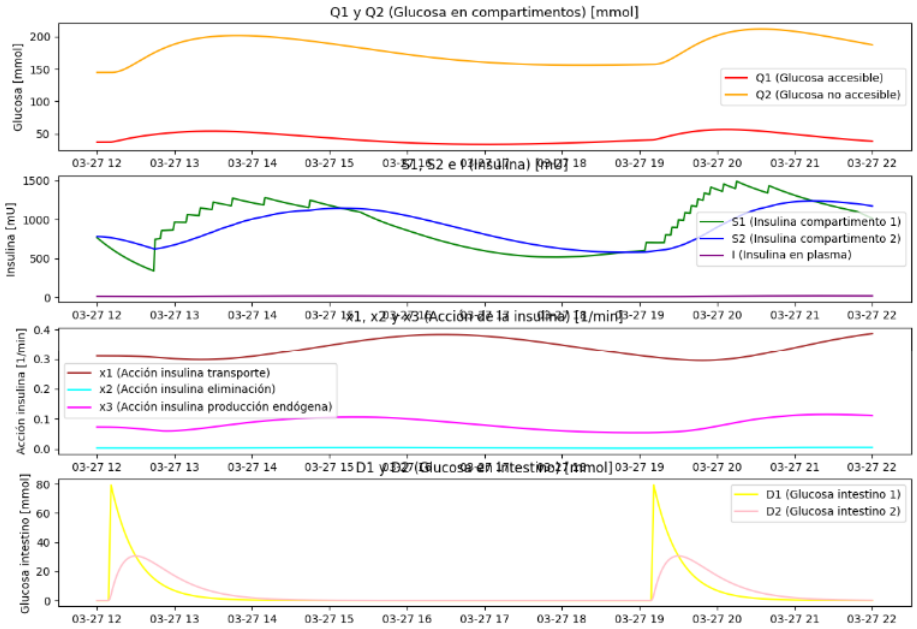


Figura 4.25: Evolución completa de variables de estado para el caso de correcciones automáticas.

- Parámetros tramo 1 (de 12h a 14h):
  - Parámetros iniciales:

```
            "kP": 0.007
            "kI": 0.003
            "kD": 0.00135
            "bR": 0.5

```

Código 4.4: Parámetros iniciales.

  - Cálculo de basal:

```
==============Basal delivery calculation=================
-> Time     = 2025-03-27 13:06:56.019662
-> TimeOld  = 2025-03-27 13:01:56.019662
-> Glucose  = 102 mg/dL
-> Error    = 2.0000 mg/dL
-> P: error = 2.0 mg/dL,          rate = 0.01 U/h
-> I: error = 0 mg/dL * h,   rate = 0.0 U/h
-> D: error = 6.0 mg/dL / h,   rate = 0.01 U/h
-> Basal Rate = 0.5221 U/h   (Nominal was 0.5 U/h)
-> Final Basal Rate = 0.5221 U/h
Basal saved: {'start': '2025-03-27T13:06:56', 'duration': 5.0
    , 'iir': 0.5221, 'type': 'Basal'}
-> Basal = 0.04 U

```

Código 4.5: Basal y ajuste de los parámetros PID (I).

- Parámetros tramo 2 (de 14h a 16h):

- Actualización de parámetros:

```
!!!!!!Adapting PID parameters at 2025-03-27 14:06:56.019662 !!!!!!!
---> Stability = 0.99,  time_in_range = 93.96551724137932 %
---> kP = 0.007
---> kI = 0.003
---> kD = 0.00135
------> kP_new = 0.007
------> kI_new = 0.00285
------> kD_new = 0.00122
Basal calculation: Total basal = 1.0167 U, Time covered = 2.0 h, Average basal rate = 0.508 U/h
Limited average basal rate to 0.508 U/h (10% range of 0.5)
------> bR_new = 0.508
PID parameters adapted at 2025-03-27 14:06:56.019662

```

Código 4.6: Basal y ajuste de los parámetros PID (II).

- Cálculo de basal:

```
==============Basal delivery calculation================
-> Time    = 2025-03-27 15:06:56.019662
-> TimeOld = 2025-03-27 15:01:56.019662
-> Glucose = 119 mg/dL
-> Error = 19.0000 mg/dL
-> P: error = 19.0 mg/dL,           rate = 0.13 U/h
-> I: error = 18.83 mg/dL * h,  rate = 0.05 U/h
-> D: error = 6.0 mg/dL / h,  rate = 0.01 U/h
-> Basal Rate = 0.702 U/h   (Nominal was 0.508 U/h)
-> Final Basal Rate = 0.702 U/h
Basal saved: {'start': '2025-03-27T15:06:56', 'duration': 5.0, 'iir': 0.702, 'type': 'Basal'}
-> Basal = 0.06 U

```

Código 4.7: Basal y ajuste de los parámetros PID (III).

- Parámetros tramo 3 (de 16h a 18h):

- Actualización de parámetros:

```
!!!!!!Adapting PID parameters at 2025-03-27 16:06:56.019662 !!!!!!!
---> Stability = 0.99,  time_in_range = 0.0 %
---> kP = 0.007
---> kI = 0.00285
---> kD = 0.00122
------> kP_new = 0.0077
------> kI_new = 0.00299
------> kD_new = 0.0011
Basal calculation: Total basal = 1.3910 U, Time covered = 2.0 h, Average basal rate = 0.695 U/h
Limited average basal rate to 0.5842 U/h (10% range of 0.508)
------> bR_new = 0.5842
```

```
PID parameters adapted at 2025-03-27 16:06:56.019662

```

Código 4.8: Basal y ajuste de los parámetros PID (IV).

- Cálculo de basal:

```
==============Basal delivery calculation=================
-> Time    = 2025-03-27 17:01:56.019662
-> TimeOld = 2025-03-27 16:56:56.019662
-> Glucose = 115 mg/dL
-> Error = 15.0000 mg/dL
-> P: error = 15.0 mg/dL,           rate = 0.12 U/h
-> I: error = 56.25 mg/dL * h,  rate = 0.17 U/h
-> D: error = -9.0 mg/dL / h,  rate = -0.01 U/h
-> Basal Rate = 0.858 U/h   (Nominal was 0.5842 U/h)
-> Final Basal Rate = 0.858 U/h
Basal saved: {'start': '2025-03-27T17:01:56', 'duration': 5.0
   , 'iir': 0.858, 'type': 'Basal'}
-> Basal = 0.07 U

```

Código 4.9: Basal y ajuste de los parámetros PID (V).

- Parámetros tramo 4 (de 18h a 20h):
  - Actualización de parámetros:

```
!!!!!!Adapting PID parameters at 2025-03-27 18:06:56.019662
  !!!!!!!
---> Stability = 0.99,  time_in_range = 21.367521367521366 %
---> kP = 0.0077
---> kI = 0.00299
---> kD = 0.0011
------> kP_new = 0.00847
------> kI_new = 0.00314
------> kD_new = 0.00099
Basal calculation: Total basal = 1.6854 U, Time covered = 2.
  0 h, Average basal rate = 0.843 U/h
Limited average basal rate to 0.67183 U/h (10% range of 0.58
  42)
------> bR_new = 0.67183
PID parameters adapted at 2025-03-27 18:06:56.019662

```

Código 4.10: Basal y ajuste de los parámetros PID (VI).

- Cálculo de basal:

```
==============Basal delivery calculation=================
-> Time    = 2025-03-27 19:06:56.019662
-> TimeOld = 2025-03-27 19:01:56.019662
-> Glucose = 103 mg/dL
-> Error = 3.0000 mg/dL
-> P: error = 3.0 mg/dL,           rate = 0.03 U/h
-> I: error = 0 mg/dL * h,  rate = 0.0 U/h
```

```
-> D: error = -3.0 mg/dL / h,  rate = -0.0 U/h
-> Basal Rate = 0.6943 U/h   (Nominal was 0.67183 U/h)
-> Final Basal Rate = 0.6943 U/h
Basal saved: {'start': '2025-03-27T19:06:56', 'duration': 5.
  0, 'iir': 0.6943, 'type': 'Basal'
-> Basal = 0.06 U
```

Código 4.11: Basal y ajuste de los parámetros PID (VII).

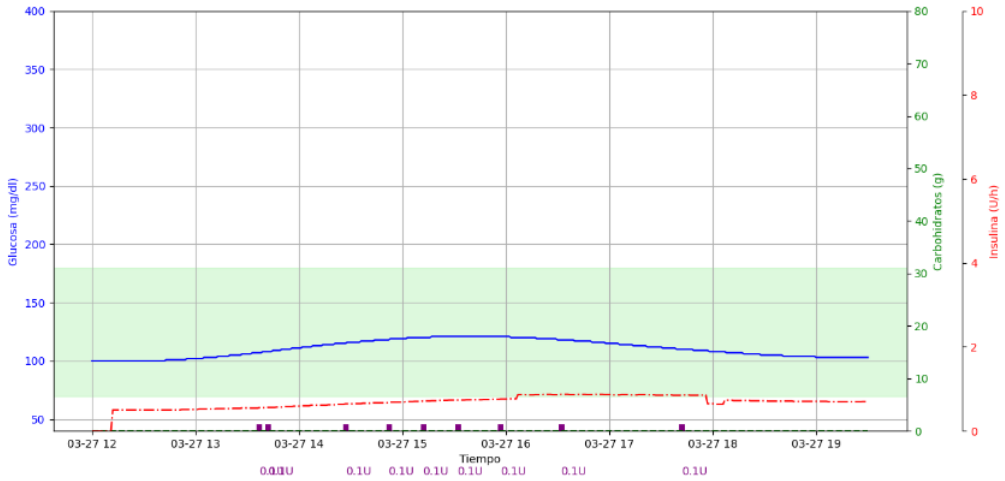


Figura 4.26: Evolución completa de glucosa e insulina para el caso de basal por control PID.

La Figura 4.27 complementa este análisis mostrando la evolución de las variables de estado a lo largo del tiempo.

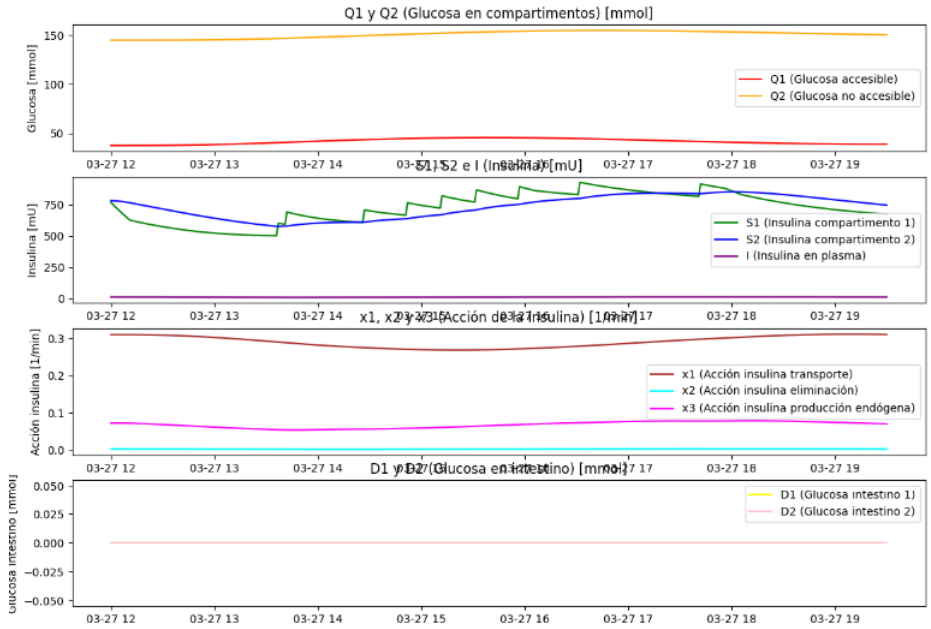


Figura 4.27: Evolución completa de variables de estado para el caso de basal por control PID.

4.6.1.4. Diferentes objetivo, ratio y sensibilidad

El ajuste de los parámetros del sistema, como el objetivo de glucosa, el ratio de insulina y la sensibilidad, influye en el control de la glucosa en sangre. Además, es común que el endocrino del paciente indique diferentes parámetros en función del momento del día. Por ejemplo, para evitar hipoglucemias nocturnas, el objetivo de la noche suele ser ligeramente superior al del resto del día. También hay momentos en los que le ratio y la sensibilidad a la insulina pueden variar (por ejemplo, también es común ser menos sensibles a la insulina durante las primeras horas de la mañana). La Figura 4.28 muestra la pantalla de configuración del objetivo de glucosa, la Figura 4.29 ilustra la pantalla de configuración de los ratios de insulina, y la Figura 4.30 muestra la sensibilidad a la insulina para la próxima simulación.

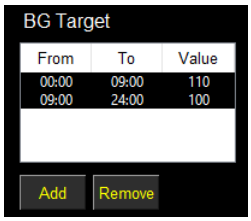


Figura 4.28: Pantalla targets de glucosa.

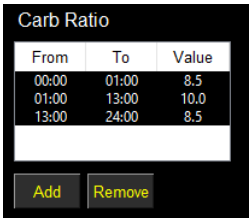


Figura 4.29: Pantalla ratios.

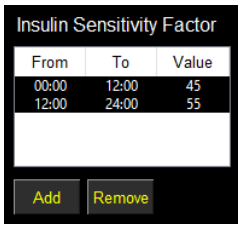


Figura 4.30: Pantalla sensibilidad a la insulina.

Para evaluar el impacto de estos cambios, la Figura 4.31 presenta un detalle de la bomba de insulina antes de administrar un bolo en el caso de una comida sin UGPs. Se ingieren 30 g de HC, si el ratio es 8.5 g/U, se administran 3.5 U de insulina.

La Figura 4.32 muestra la evolución de la glucosa inicial y la Figura 4.33 detalla la actuación de la bomba en ese momento.

Antes de que llegue la noche, el objetivo es 100 mg/dL, el raio de HC es 8.5 g/U y la sensibilidad es 55 mg/dL / U. Ya se ha mostrado como afecta el ratio a la administración de insulina por un bolo de comida. Para la sensibilidad y objetivo, se puede comparar en un cálculo de insulina basal y correcciones para este periodo de tiempo. Por ejemplo, para las 21:30 h:

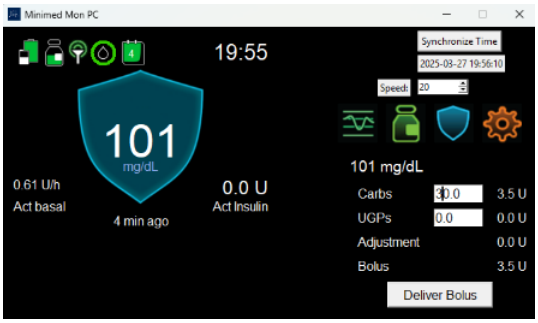


Figura 4.31: Detalle de la bomba de insulina en el momento previo a administrar un bolo por comida sin UGPs.

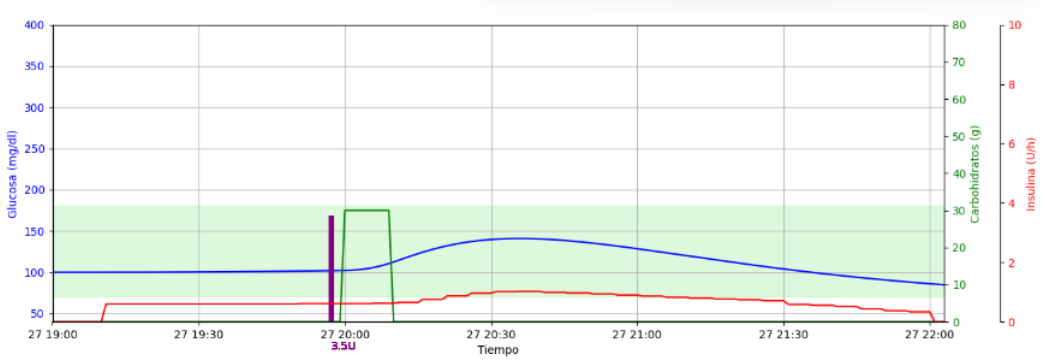


Figura 4.32: Glucosa, insulina y carbohidratos para el caso de diferentes parámetros I.

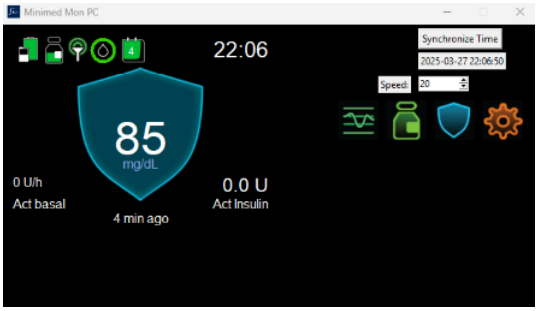


Figura 4.33: Detalle de la bomba de insulina en el momento de la Figura 4.32.

```
==============Basal delivery calculation================
-> Time     = 2025-03-27 21:30:50.688837
-> TimeOld = 2025-03-27 21:25:50.688837
-> Glucose = 107 mg/dL
-> Error = 7.0000 mg/dL
-> P: error = 7.0 mg/dL,          rate = 0.05 U/h
-> I: error = 0 mg/dL * h,  rate = 0.0 U/h
-> D: error = -48.0 mg/dL / h,  rate = -0.06 U/h
-> Basal Rate = 0.5842 U/h   (Nominal was 0.6 U/h)
-> Final Basal Rate = 0.5842 U/h
Basal saved: {'start': '2025-03-27T21:30:50', 'duration': 5.0, 'iir':
    0.5842, 'type': 'Basal'}
-> Basal = 0.05 U
***************Correction bolus calculation************:
-> Current BG = 107 mg/dL
-> Nominal Target BG = 100 mg/dL
```

```
16 ->  Target BG = 100 mg/dL
17 -> Active Insulin = 0.9090 U
18 -> Effective Active Insulin = 0.9090 U
19 -> Basal Insulin = 0.05 U
20 -> Effective Basal Insulin = 0.0500 U
21 -> Correction bolus = 7 mg/dL / (55.0 mg/dL/U) - (0.9090 U) - (0.05 U
     ) = -0.8318 U
22 -> Apply correction: True
23 -> Effective correction bolus: -0.8000
24
```

Código 4.12: Diferentes objetivo, ratio y sensibilidad (I).

Las Figuras 4.34 y 4.35 ilustran dos momentos clave antes y después de modificar el ratio de HC. Se observa que en el primer caso para 30 g de HC (ratio de 8.5 g/U) se administraría 3.5 U, mientras que en el segundo (ratio de 10 g/U) se administra 3.0 U. En la simulación, se administra justo cuando ya ha cambiado el ratio a 10 g/U.

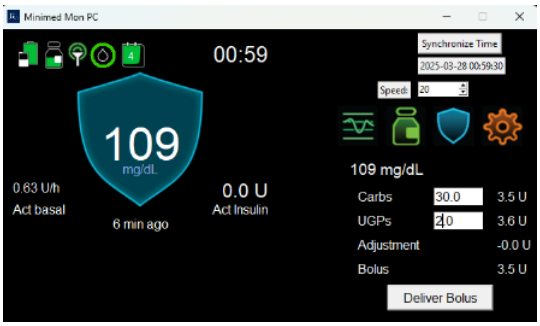


Figura 4.34: Detalle de la bomba de insulina para administrar un bolo por comida antes de cambiar su ratio. El ratio es 8.5 g/U. No se administra este bolo.

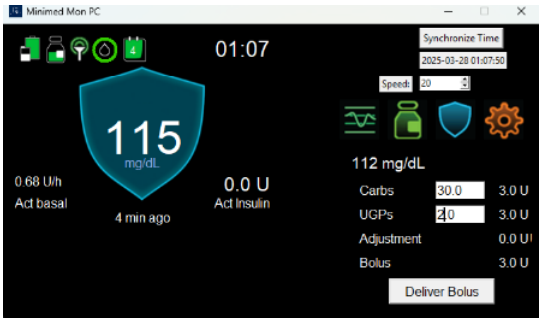


Figura 4.35: Detalle de la bomba de insulina para administrar un bolo por comida después de cambiar su ratio. El ratio es 10 g/U. Se administra este bolo.

La Figura 4.36 presenta la evolución de la glucosa tras esta comida de 30 g de HC con 2 UGPs, con su respectivo detalle en la bomba de insulina en la Figura 4.37.

De manera similar, las Figuras 4.38 y 4.39 muestran la continuación.

Ahora el objetivo es 110 mg/dL, el raio de HC es 10.0 g/U y la sensibilidad es 45 mg/dL / U. Ya se ha mostrado como afecta el ratio a la administración de insulina por un bolo de comida. Para la sensibilidad y objetivo, se puede comparar en un cálculo de insulina basal y correcciones para este periodo de tiempo. Por ejemplo, para las 02:30 h:

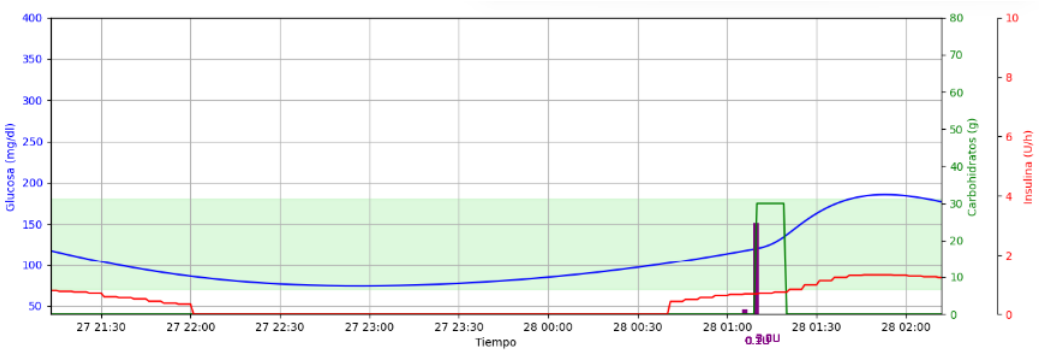


Figura 4.36: Glucosa, insulina y carbohidratos para el caso de diferentes parámetros II.

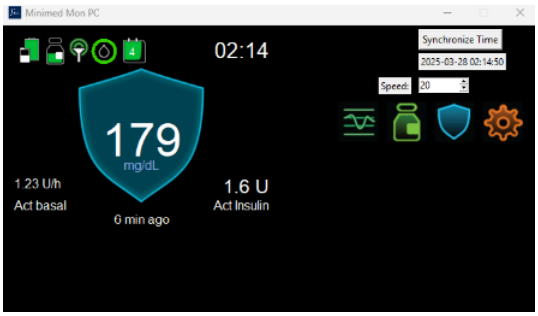


Figura 4.37: Detalle de la bomba de insulina en el momento de la Figura 4.36.

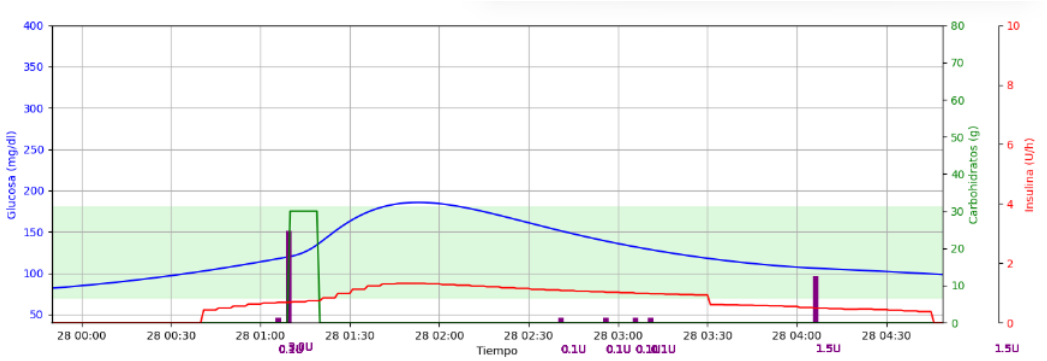


Figura 4.38: Glucosa, insulina y carbohidratos para el caso de diferentes parámetros III.

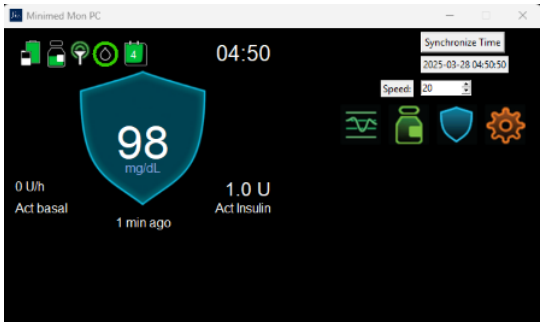


Figura 4.39: Detalle de la bomba de insulina en el momento de la Figura 4.38.

```
1 ==============Basal delivery calculation=================
2 -> Time     = 2025-03-28 02:30:50.688837
```

```
-> TimeOld = 2025-03-28 02:25:50.688837
-> Glucose = 162 mg/dL
-> Error = 52.0000 mg/dL
-> P: error = 52.0 mg/dL,           rate = 0.36 U/h
-> I: error = 75.92 mg/dL * h,  rate = 0.23 U/h
-> D: error = -51.0 mg/dL / h,  rate = -0.07 U/h
-> Basal Rate = 1.1229 U/h   (Nominal was 0.6 U/h)
-> Final Basal Rate = 1.1229 U/h
Basal saved: {'start': '2025-03-28T02:30:50', 'duration': 5.0, 'iir':
   1.1229, 'type': 'Basal'}
-> Basal = 0.09 U
**********Correction bolus calculation**********:
-> Current BG = 162 mg/dL
-> Nominal Target BG = 110 mg/dL
->  Target BG = 110 mg/dL
-> Active Insulin = 1.0690 U
-> Effective Active Insulin = 1.0690 U
-> Basal Insulin = 0.09 U
-> Effective Basal Insulin = 0.0900 U
-> Correction bolus = 52 mg/dL / (45.0 mg/dL/U) - (1.0690 U) - (0.09
  U) = -0.0035 U
-> Apply correction: True
-> Effective correction bolus: -0.0000

```

Código 4.13: Diferentes objetivo, ratio y sensibilidad (II).

Finalmente, las Figuras 4.40 y 4.41 ofrecen una visión completa de la evolución de la glucosa, la insulina y los carbohidratos, así como de las variables de estado, cuando se modifican el objetivo de glucosa, la sensibilidad a la insulina y el ratio de administración.

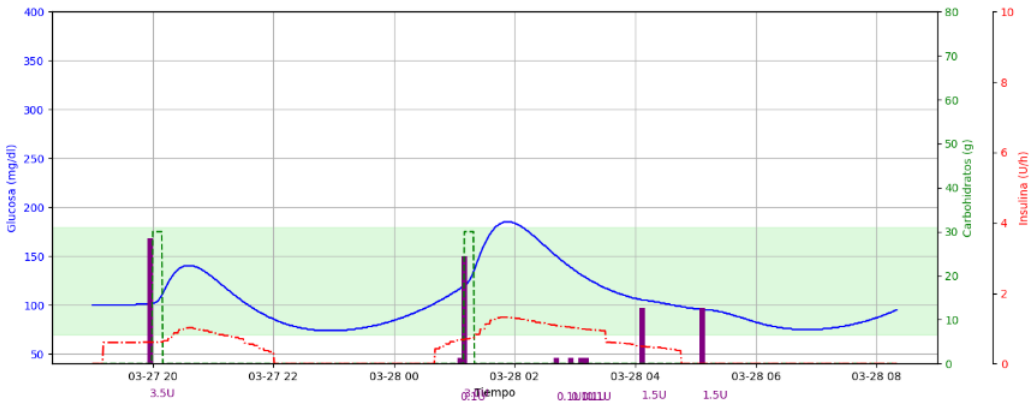


Figura 4.40: Evolución completa de glucosa, insulina y carbohidratos para el caso de diferentes objetivo, sensibilidad y ratio.

Se incluye ahora una parte inicial de la simulación anterior pero inyectando la insulina 40 minutos antes. De esta forma, la glucosa empieza a disminuir antes y el error que utiliza el controlador PID es negativo. De esta forma, se puede comparar cómo difiere el perfil de insulina basal suministrado. Esto se puede ver en la Figura 4.42.

El detalla se muestra en la Figura 4.43.

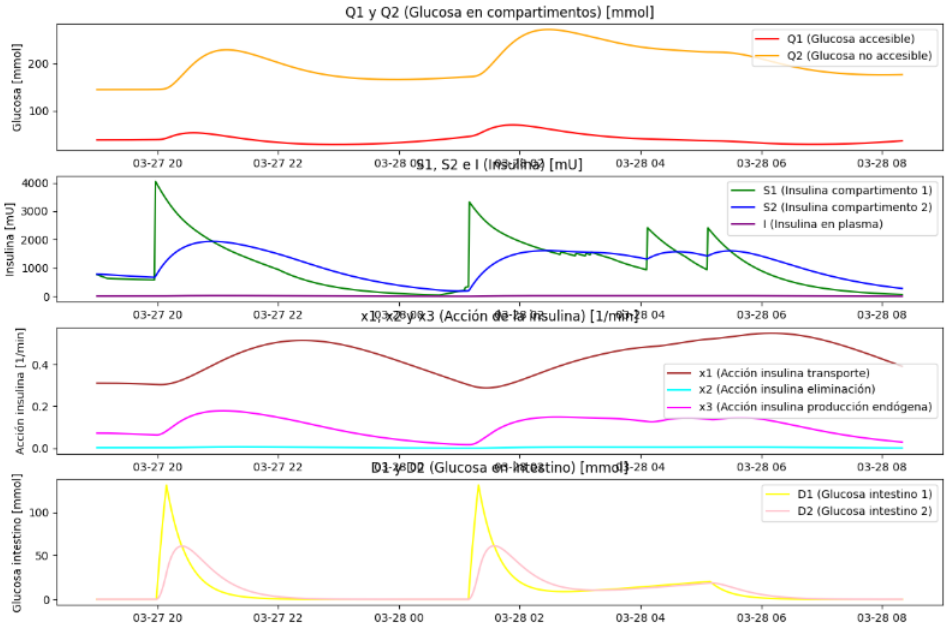


Figura 4.41: Evolución completa de variables de estado para el caso de diferentes objetivo, sensibilidad y ratio.

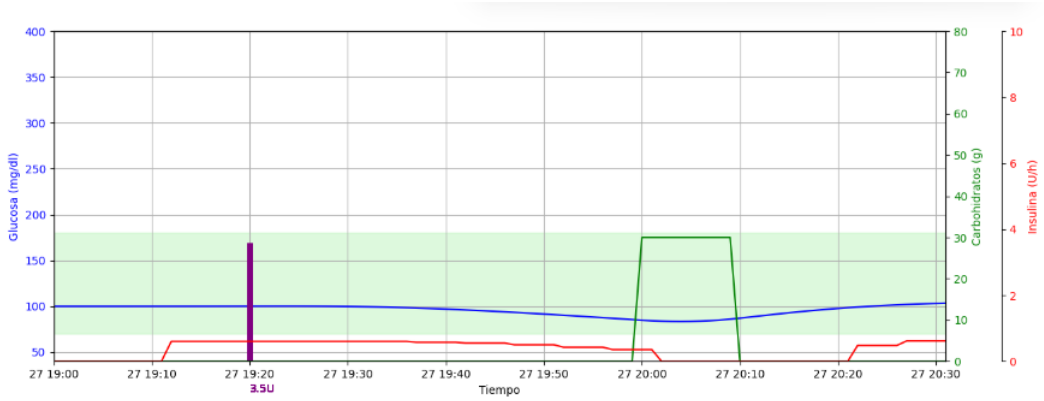


Figura 4.42: Glucosa, insulina y carbohidratos para el caso de antes pero inyectando la insulina 40 min antes I.

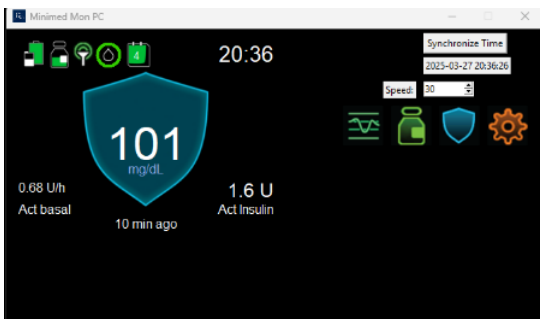


Figura 4.43: Detalle de la bomba de insulina en el momento de la Figura 4.42.

Y a continuación más tiempo de evolución en 4.44.

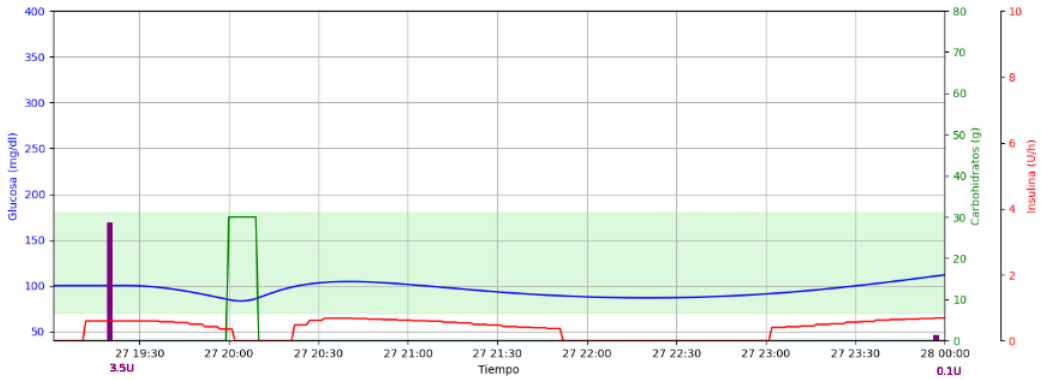


Figura 4.44: Glucosa, insulina y carbohidratos para el caso de antes pero inyectando la insulina 40 min antes II.

4.6.2. Simulación con eventos

Como referencia, se muestra en la Figura 4.45 una simulación sin eventos, que va a servir como referencia para comparar casos de simulaciones con eventos.

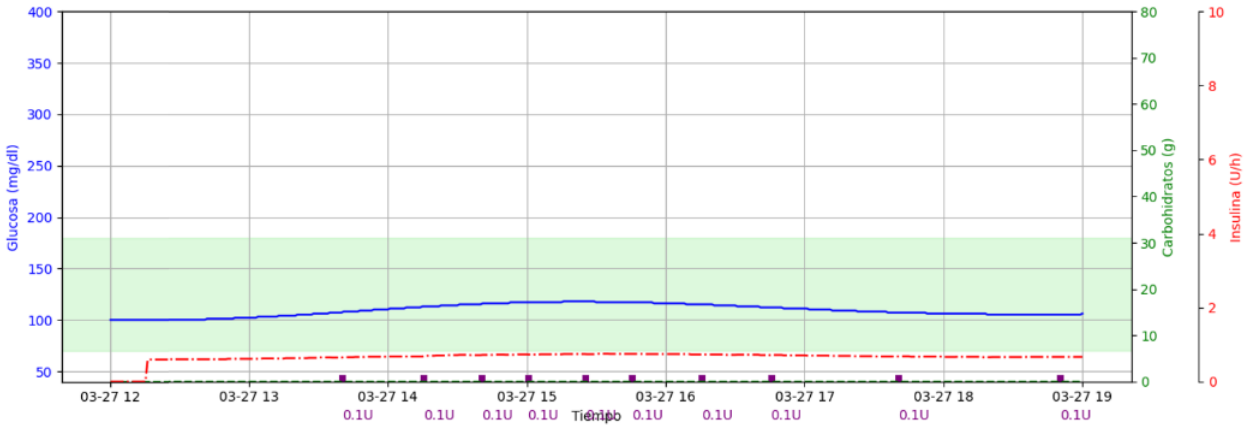


Figura 4.45: Glucosa e insulina para un caso sin eventos.

4.6.2.1. Alcohol

En esta subsección se presentan los resultados de la simulación para el caso de la ingestión de alcohol en dos escenarios diferentes: el consumo de 3 unidades y 6 unidades de alcohol, durante 3h ambas (desde las 12:15 h hasta 15:15 h). Las gráficas muestran la interacción entre los niveles de glucosa, la administración de insulina y los carbohidratos presentes en el cuerpo en cada uno de estos escenarios.

3 unidades

En la Figura 4.46 se observa la pantalla de la bomba de ingestión de alcohol, donde se introducen 3 unidades.

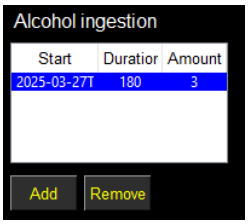


Figura 4.46: Pantalla de ingestión de alcohol.

A partir de este punto, en la Figura 4.47, se muestra el bolo de insulina administrado para los primeros 15 g de carbohidratos que se van a ingerir a las 13:15 h. Para un ratio de 8.5 g/U, se obtienen 15 g / 8.5 g/U = 1.8 U. Sin embargo, se realiza un ajuste de -0.5 U. Esto se debe a que el objetivo a cambiado debido al evento del alcohol. El algoritmo calcula un objetivo de 130 mg/dL (targte original + cantidad*10mg/dl). Por lo tanto, existe un error de -32 mg/dL. La sensibilidad también ha cambiado. A pasado de 55.0 mg/dL / U a unos 64.3 mg/dL / U. Por lo tanto, -32 mg/dL / 64.3 mg/dL / U = -0.5 U. Es por eso que se liberan finalmente 1.3 U de insulina.

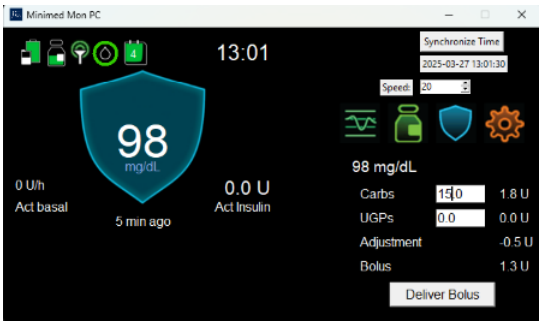


Figura 4.47: Bolo de insulina por 15 g iniciales para 3 unidades de alcohol.

Posteriormente, en la Figura 4.48, se presenta la evolución de la glucosa, insulina y carbohidratos en el cuerpo durante la fase inicial del metabolismo del alcohol, junto con los 15 g de HC. La Figura 4.49 ofrece un detalle de la bomba de insulina en este momento.

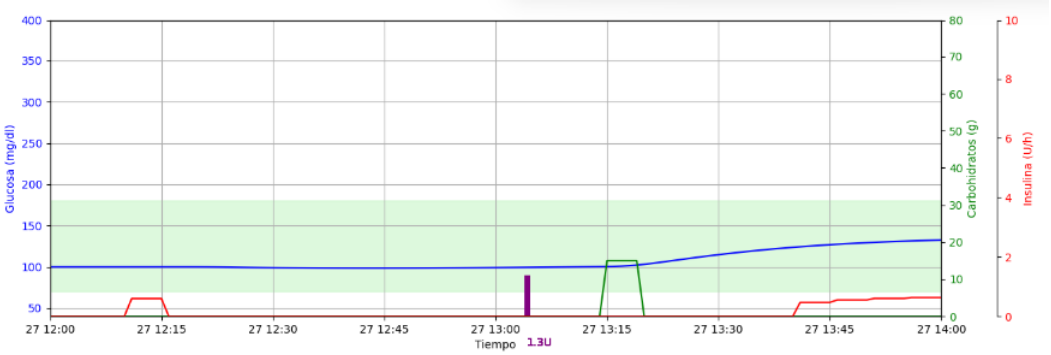


Figura 4.48: Glucosa, insulina y carbohidratos para el caso de 3 unidades de alcohol I.

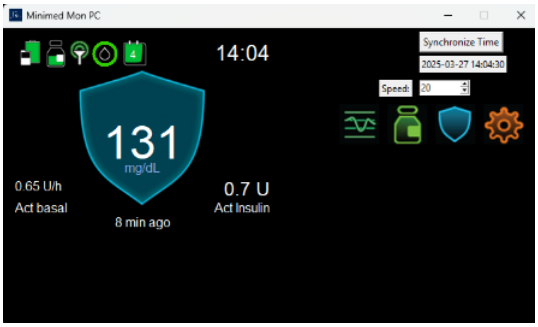


Figura 4.49: Detalle de la bomba de insulina en el momento de la Figura 4.48.

Continuando con la simulación, la Figura 4.50 ilustra la evolución de los mismos parámetros (glucosa, insulina y carbohidratos) en una etapa posterior.

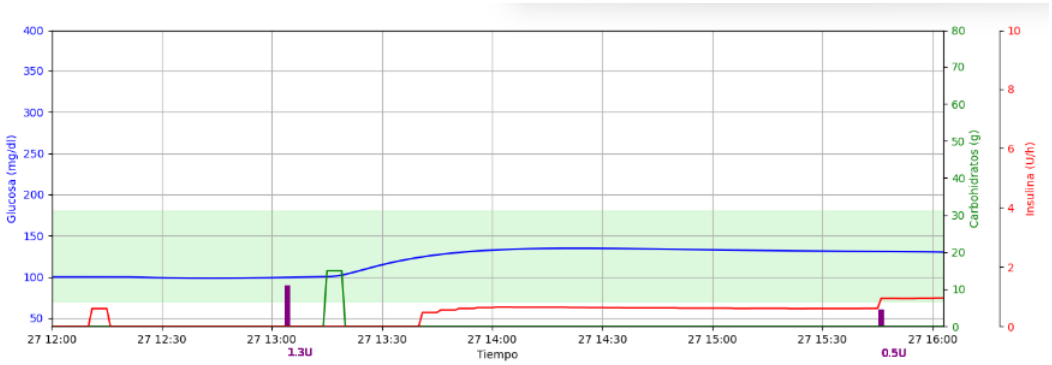


Figura 4.50: Glucosa, insulina y carbohidratos para el caso de 3 unidades de alcohol II.

La Figura 4.51 muestra el bolo de insulina administrado para los últimos 15 g de carbohidratos. En este momento, el objetivo ya vuelve a ser 100 mg/dL, por lo tanto el error ahora es de + 30 mg/dL. La sensibilidad también vuelve a 55.0 mg/dL / U, por lo que el ajuste es una corrección de 0.1 U.

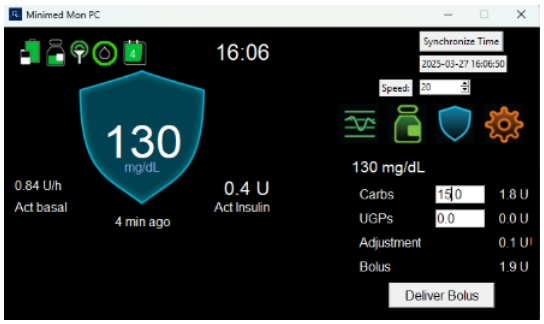


Figura 4.51: Bolo de insulina por 15 g finales para 3 unidades de alcohol.

En la Figura 4.52 se representa la evolución siguiente de la glucosa. Con el detalle de la bomba de insulina en la Figura 4.53.

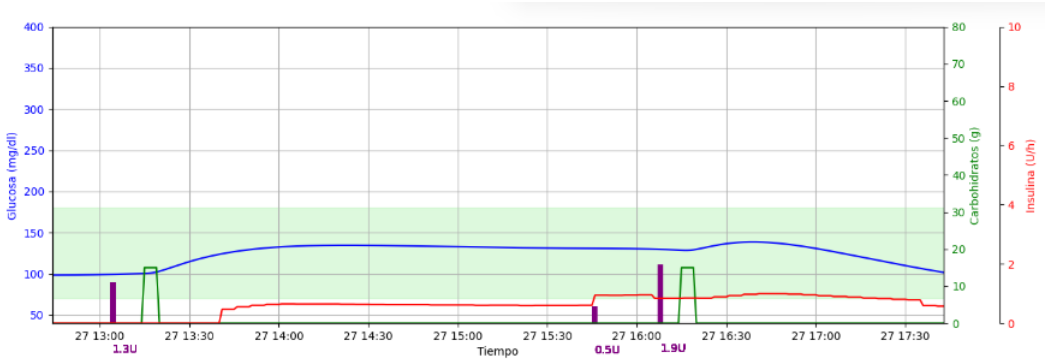


Figura 4.52: Glucosa, insulina y carbohidratos para el caso de 3 unidades de alcohol III.

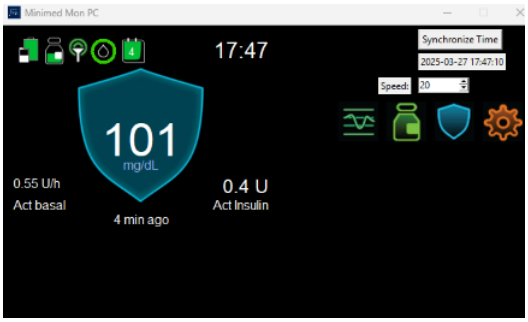


Figura 4.53: Detalle de la bomba de insulina en el momento de la Figura 4.52.

Finalmente, en la Figura 4.54, se presenta la evolución completa de los niveles de glucosa, insulina y carbohidratos a lo largo de toda la simulación para las 3 unidades de alcohol.

Desde que se termina la ingesta de alcohol hasta 30 minutos posteriores está inhibida la liberación de bolos de corrección. Es por ello que la primera corrección se realiza a las 15:45 h.

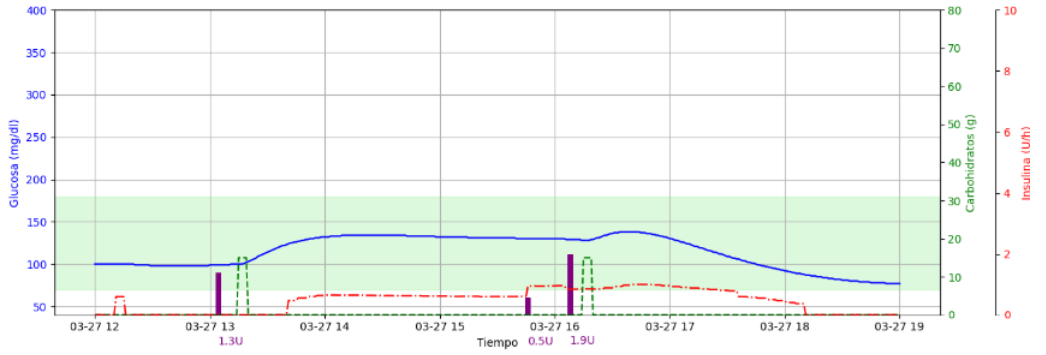


Figura 4.54: Evolución completa de glucosa, insulina y carbohidratos para el caso de 3 unidades de alcohol.

La Figura 4.55 expone la evolución completa de las variables de estado involucradas en este escenario, mientras que la Figura 4.56 muestra la evolución de los parámetros fisiológicos, donde se aprecia el descenso en EGP0.

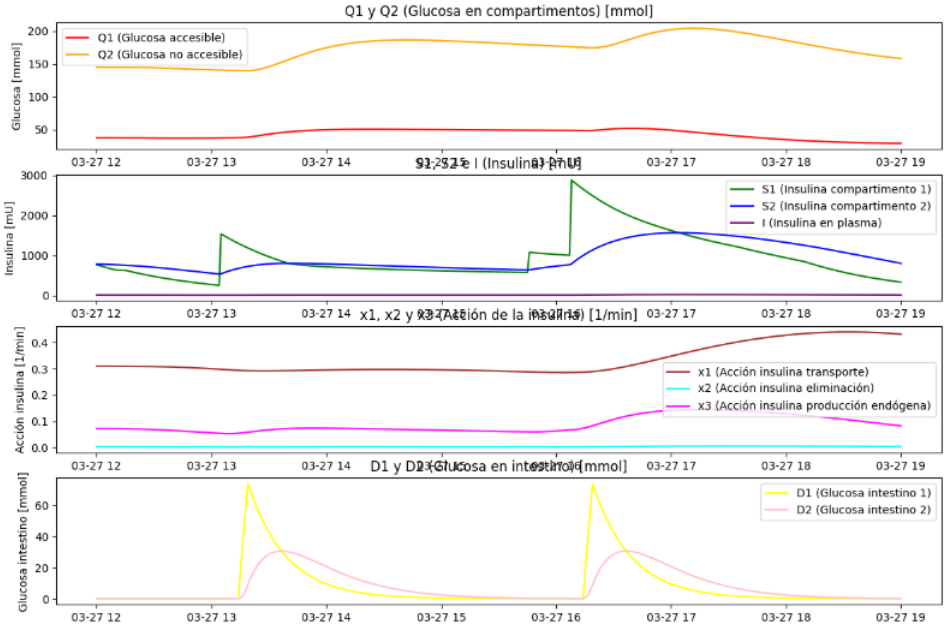


Figura 4.55: Evolución completa de variables de estado para el caso de 3 unidades de alcohol.

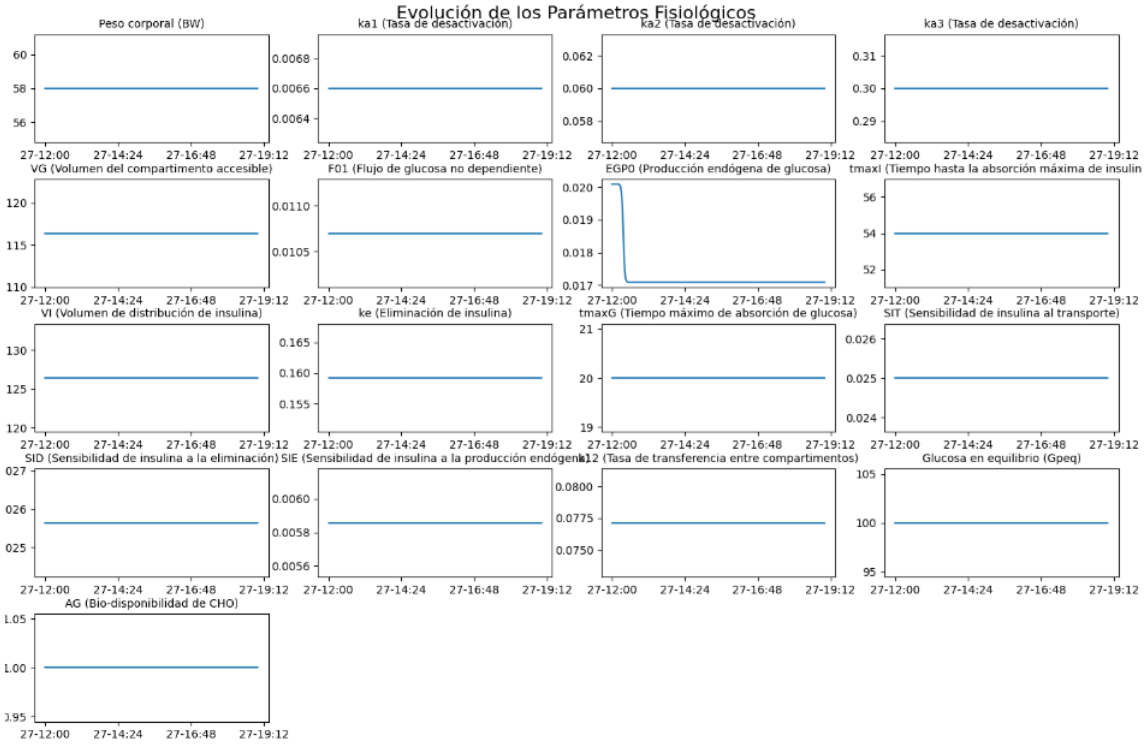


Figura 4.56: Evolución completa de parámetros fisiológicos para el caso de 3 unidades de alcohol.

6 unidades

Para el caso de 6 unidades de alcohol, la Figura 4.57 muestra el bolo de insulina administrado para los primeros 15 g de carbohidratos. Para un ratio de $8{,}5g/U$, se obtienen $15g/8{,}5g/U = 1{,}8U$. Sin embargo, se realiza un ajuste de -0.6 U. Esto se debe a que el objetivo a cambiado debido al evento del alcohol. El algoritmo calcula un objetivo de 150 mg/dL. Por lo tanto,

existe un error de -55 mg/dL. La sensibilidad también ha cambiado. A pasado de 55.0 mg/dL / U a unos 88.0 mg/dL / U. Por lo tanto, $-55mg/dL/88mg/dL/U = -0{,}6U$. Es por eso que se liberan finalmente 1.2 U de insulina.

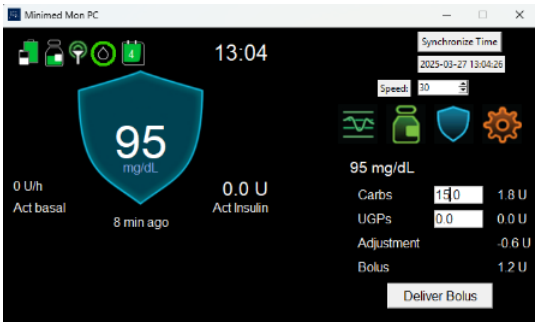


Figura 4.57: Bolo de insulina por 15 g iniciales para 6 unidades de alcohol.

De manera similar, la Figura 4.58 muestra el bolo administrado para los últimos 15 g de carbohidratos.

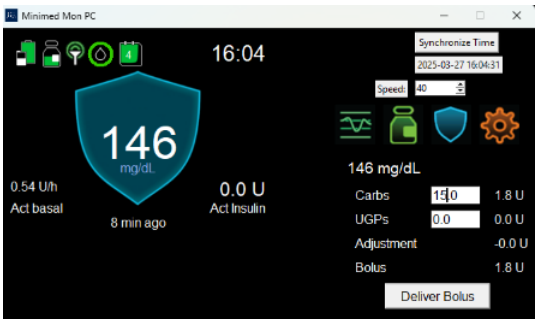


Figura 4.58: Bolo de insulina por 15 g finales para 6 unidades de alcohol.

En la Figura 4.59, se observa la evolución completa de glucosa, insulina y carbohidratos a lo largo de toda la simulación para el caso de 6 unidades de alcohol.

Desde que se termina la ingesta de alcohol hasta 180 minutos posteriores (cantidad*30 min) está inhibida la liberación de bolos de corrección. Es por ello que la primera corrección se realiza a las 18:15 h.

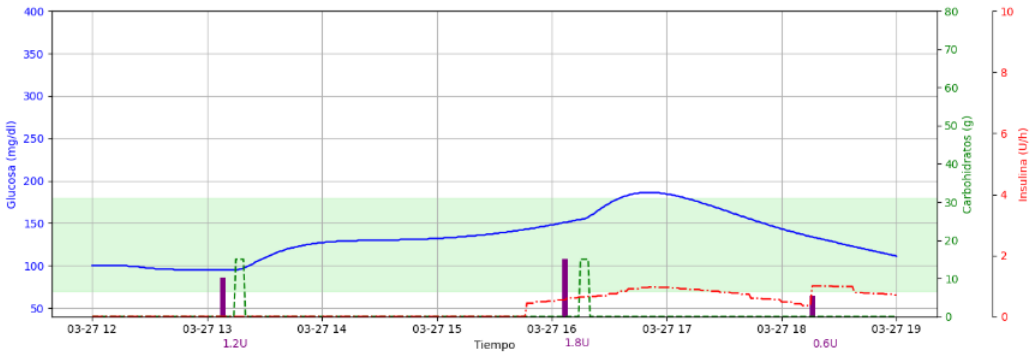


Figura 4.59: Evolución completa de glucosa, insulina y carbohidratos para el caso de 6 unidades de alcohol.

Las siguientes figuras, 4.60 y 4.61, detallan la evolución completa de las variables de estado y el valor de EGP0, respectivamente, para este mismo escenario, proporcionando una visión exhaustiva de la respuesta del cuerpo ante la ingestión de 6 unidades de alcohol.

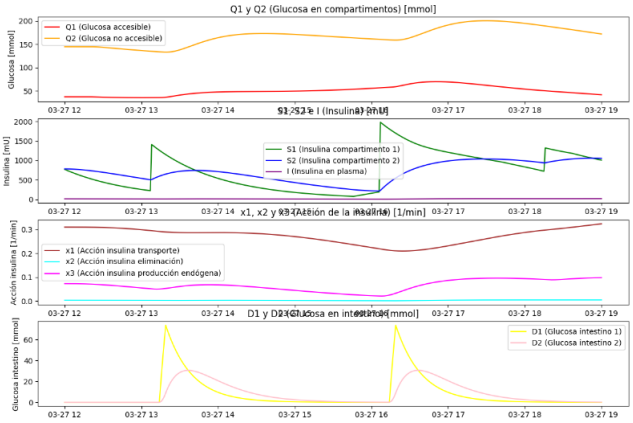


Figura 4.60: Evolución completa de variables de estado para el caso de 6 unidades de alcohol.

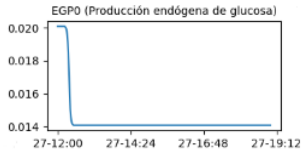


Figura 4.61: Evolución completa de EGP0 para el caso de 6 unidades de alcohol.

4.6.2.2. Ejercicio físico

Para la simulación del evento de ejercicio físico, se configura inicialmente la pantalla de ejercicio en la bomba en la Figura 4.62. En esta pantalla se pueden configurar el inicio, la duración y la intensidad.

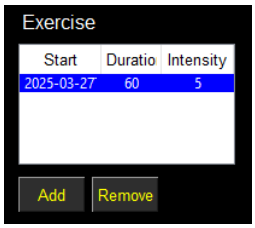


Figura 4.62: Pantalla de ejercicio físico.

La Figura 4.63 muestra la evolución de los niveles de glucosa e insulina durante un ejercicio de intensidad 4, que dura 30 minutos, comenzando a las 15h. El algoritmo solo inhibe las correcciones automáticas si la duración del ejercicio es mayor de 60 minutos, por lo que sí se calculan y liberan bolos de corrección. Además, el objetivo ahora es 150 mg/dL desde 2 horas antes hasta 30 minutos después de la práctica del ejercicio, y la sensibilidad a la insulina se multiplica por 1.4 $(1 + 0{,}1 * intensidad)$.

Se puede ver como de 13h a 15h, como el error en glucosa es negativo (comenzando a las 13h en -50 mg/dL), no se libera insulina basal. Se muestran trazas para las 13:50 h

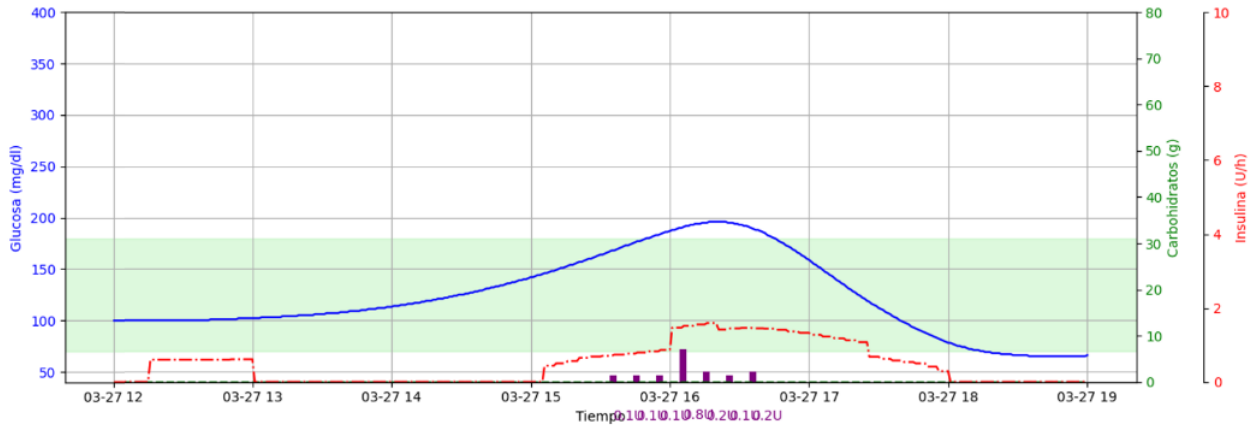


Figura 4.63: Evolución completa de glucosa e insulina para el caso de ejercicio físico.

```
==============Basal delivery calculation================
-> Time     = 2025-03-27 13:50:23.487835
-> TimeOld = 2025-03-27 13:45:23.487835
-> Glucose = 109 mg/dL
-> Error = -41.0000 mg/dL
-> Error -41.0000 mg/dL < -10 mg/dL (Error limit)
-> Basal Rate = 0 U/h
***************Correction bolus calculation************:
-> Current BG = 109 mg/dL
-> Nominal Target BG = 100 mg/dL
->  Target BG = 150.0 mg/dL
-> Active Insulin = 0.0000 U
-> Effective Active Insulin = 0.0000 U
-> Basal Insulin = 0 U
-> Effective Basal Insulin = 0.0000 U
-> Correction bolus = -41.0 mg/dL / (55.0 mg/dL/U) - (0.0000 U) - (0.0
    U) = -0.7455 U
-> Apply correction: True
-> Effective correction bolus: -0.7000

```

Código 4.14: Ejercicio físico (I).

A partir de las 15h ya se va liberando insulina basal, incluso correcciones.

```

==============Basal delivery calculation================
-> Time     = 2025-03-27 15:35:23.487835
-> TimeOld = 2025-03-27 15:30:23.487835
-> Glucose = 161 mg/dL
-> Error = 11.0000 mg/dL
-> P: error = 11.0 mg/dL,           rate = 0.08 U/h
-> I: error = 0 mg/dL * h,  rate = 0.0 U/h
-> D: error = 45.0 mg/dL / h,  rate = 0.06 U/h
-> Basal Rate = 0.7377 U/h   (Nominal was 0.6 U/h)
-> Final Basal Rate = 0.7377 U/h
Basal saved: {'start': '2025-03-27T15:35:23', 'duration': 5.0, 'iir':
    0.7377, 'type': 'Basal'}
-> Basal = 0.06 U
```

```
14 ***************Correction bolus calculation************:
15 -> Current BG = 161 mg/dL
16 -> Nominal Target BG = 100 mg/dL
17 ->  Target BG = 150.0 mg/dL
18 -> Active Insulin = 0.0000 U
19 -> Effective Active Insulin = 0.0000 U
20 -> Basal Insulin = 0.06 U
21 -> Effective Basal Insulin = 0.0600 U
22 -> Correction bolus = 11.0 mg/dL / (76.20925195888888 mg/dL/U) - (0.00
      00 U) - (0.06 U) = 0.0843 U
23 -> Apply correction: True
24 -> Effective correction bolus: 0.1000
25 Correction bolus saved: {'start': '2025-03-27T15:35:23', 'i': 0.1, '
      type': 'Correction'}
26
```

Código 4.15: Ejercicio físico (II).

A partir de las 16 h, se restablece el objetivo, y la sensibilidad a la insulina disminuye un poco, por lo que aparece una corrección mayor y la basal aumenta:

```
1
2 ==============Basal delivery calculation================
3 -> Time    = 2025-03-27 16:05:23.487835
4 -> TimeOld = 2025-03-27 16:00:23.487835
5 -> Glucose = 186 mg/dL
6 -> Error = 86.0000 mg/dL
7 -> P: error = 86.0 mg/dL,           rate = 0.6 U/h
8 -> I: error = 14.0 mg/dL * h,  rate = 0.04 U/h
9 -> D: error = 201.0 mg/dL / h,  rate = 0.27 U/h
10 -> Basal Rate = 1.5153 U/h   (Nominal was 0.6 U/h)
11 -> Final Basal Rate = 1.5153 U/h
12 Basal saved: {'start': '2025-03-27T16:05:23', 'duration': 5.0, 'iir':
      1.5153, 'type': 'Basal'}
13 -> Basal = 0.13 U
14 ***************Correction bolus calculation************:
15 -> Current BG = 186 mg/dL
16 -> Nominal Target BG = 100 mg/dL
17 ->  Target BG = 100.0 mg/dL
18 -> Active Insulin = 0.2685 U
19 -> Effective Active Insulin = 0.2685 U
20 -> Basal Insulin = 0.13 U
21 -> Effective Basal Insulin = 0.1300 U
22 -> Correction bolus = 86.0 mg/dL / (71.8092519588889 mg/dL/U) - (0.268
      5 U) - (0.13 U) = 0.7992 U
23 -> Apply correction: True
24 -> Effective correction bolus: 0.8000
25 Correction bolus saved: {'start': '2025-03-27T16:05:23', 'i': 0.8, '
      type': 'Correction'}
26
```

Código 4.16: Ejercicio físico (III).

En la Figura 4.64, se puede observar el cambio en las variables de estado, mientras que la Figura 4.65 presenta una visión completa de la evolución de los parámetros fisiológicos a lo largo de la

actividad física. Se ve como momentos después del ejercicio físico las sensibilidades SIT y SID aumentan considerablemente.

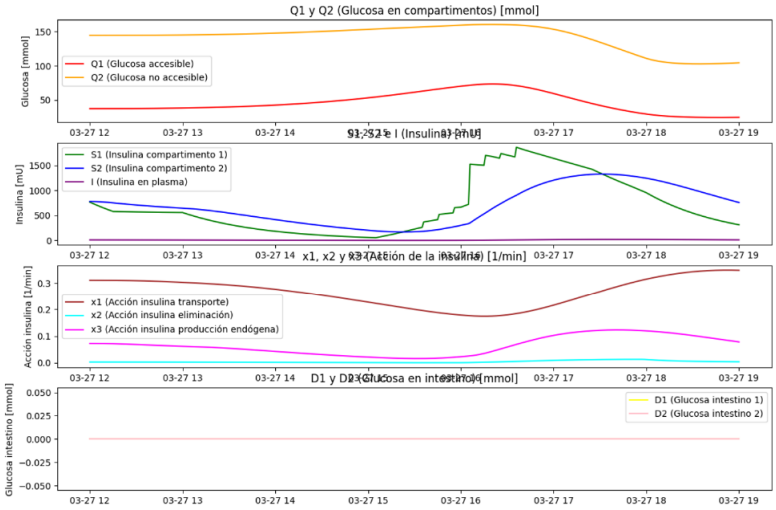


Figura 4.64: Evolución completa de variables de estado para el caso de ejercicio físico.

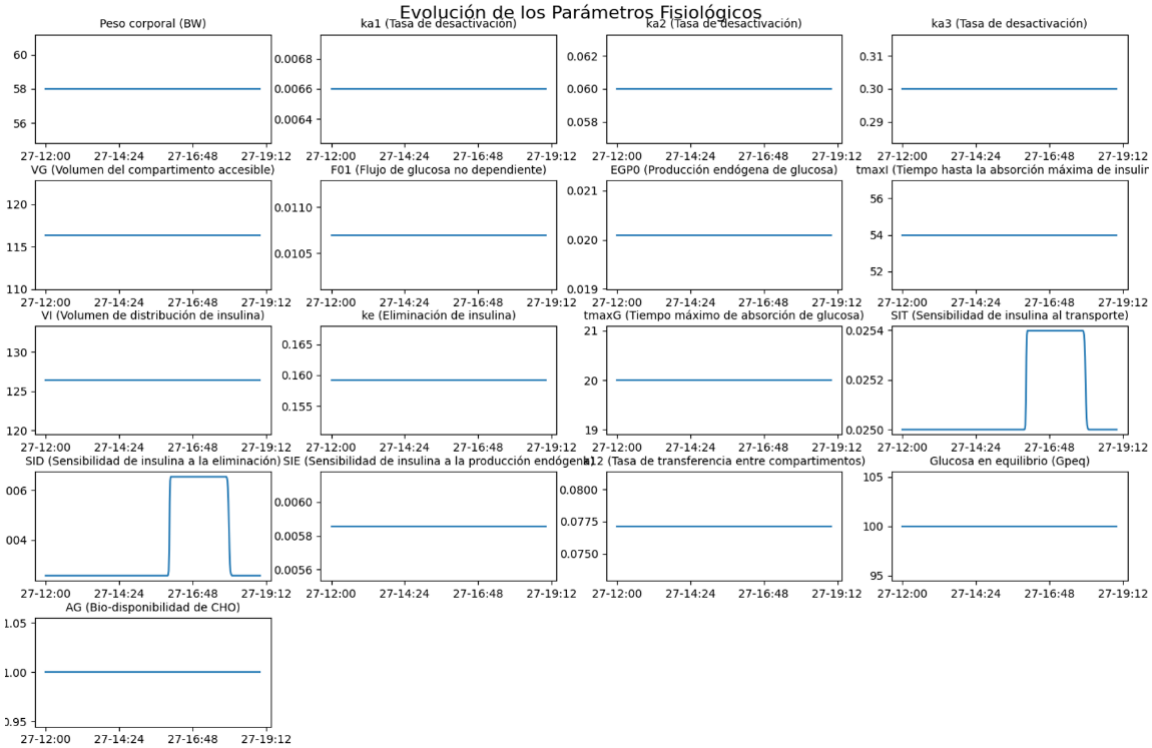


Figura 4.65: Evolución completa de parámetros fisiológicos para el caso de ejercicio físico.

4.6.2.3. Enfermedad

Para el caso de enfermedad, la Figura 4.66 ilustra la pantalla de manejo de este evento.

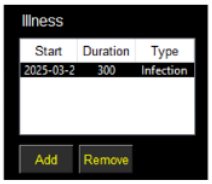


Figura 4.66: Pantalla de enfermedad.

En la Figura 4.67, se visualiza cómo la glucosa e insulina evolucionan durante una infección, que se considera que empieza a las 13h y dura 5 h. Durante este tiempo, la bomba de insulina considera que la sensibilidad es el 80 % de la nominal. Si la nominal era de 55.0 mg/dL / U, ahora es 44.0 mg/dL/U.

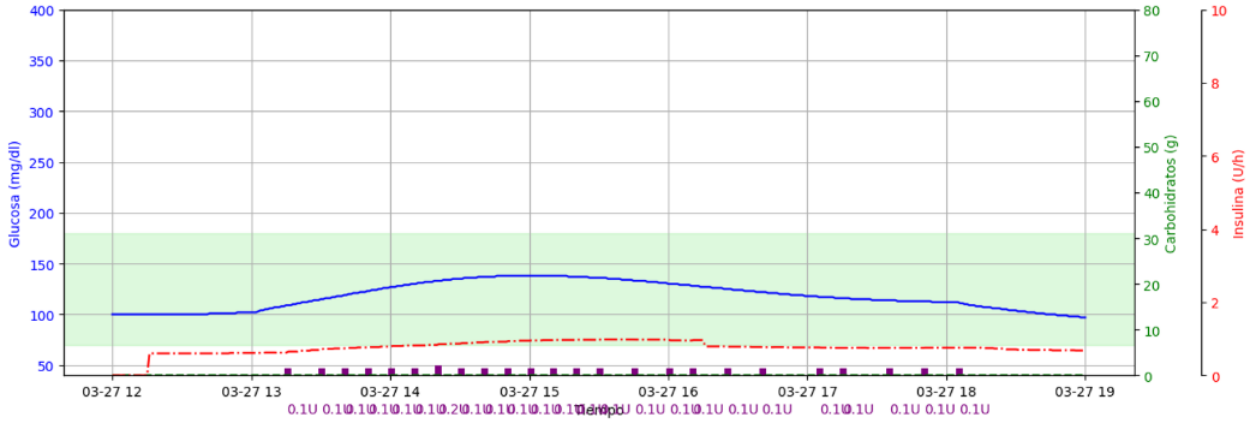


Figura 4.67: Evolución completa de glucosa e insulina para el caso de infección.

Se ve como necesita mas correcciones aunque el azucar no llega a dispararse en ningún momento. Una traza para una corrección se muestra aquí:

```
**********Correction bolus calculation**********:
-> Current BG = 122 mg/dL
-> Nominal Target BG = 100 mg/dL
->  Target BG = 100.0 mg/dL
-> Active Insulin = 0.3617 U
-> Effective Active Insulin = 0.3617 U
-> Basal Insulin = 0.06 U
-> Effective Basal Insulin = 0.0600 U
-> Correction bolus = 22.0 mg/dL / (44.0 mg/dL/U) - (0.3617 U) - (0.06
    U) = 0.0783 U
-> Apply correction: True
-> Effective correction bolus: 0.1000
Correction bolus saved: {'start': '2025-03-27T17:50:23', 'i': 0.1, '
   type': 'Correction'}

```

Código 4.17: Enfermedad.

La evolución de las variables de estado se muestra en la Figura 4.68, mientras que en la Figura 4.69 se presenta la evolución de los parámetros fisiológicos específicos para este caso. Se observa cómo disminuyen las sensibilidades SIT y SID, y además aumenta un poco el EGP0.

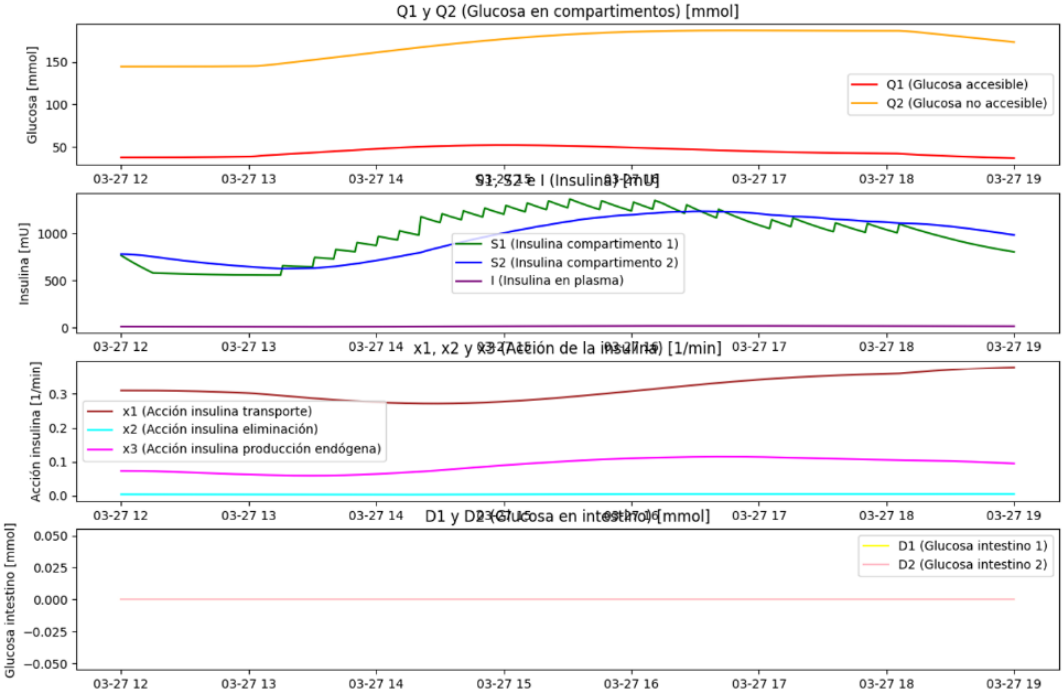


Figura 4.68: Evolución completa de variables de estado para el caso de infección.

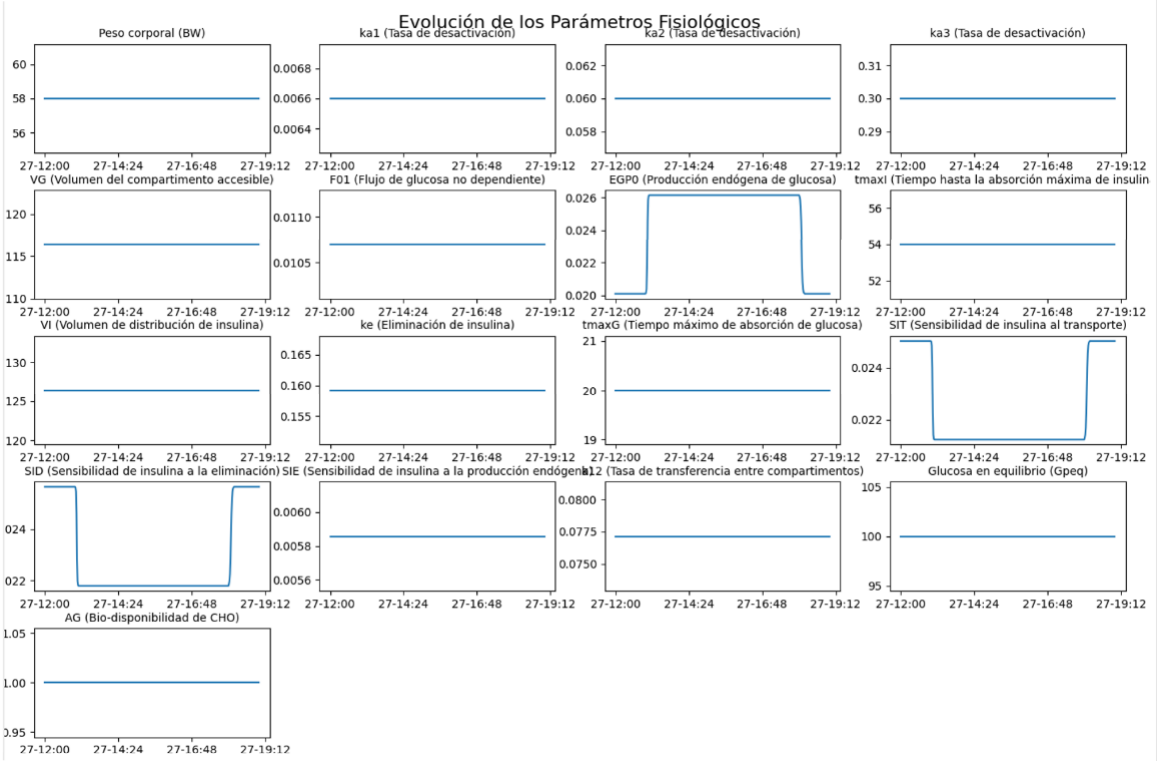


Figura 4.69: Evolución completa de parámetros fisiológicos para el caso de infección.

4.6.2.4. Estado anímico

El evento relacionado con el estado anímico contempla dos situaciones: estrés y relajación. En la Figura 4.70, se presenta la pantalla del sistema cuando el usuario está bajo estrés durante 2.5 h. Se muestra la evolución de la glucosa e insulina en la Figura 4.71.

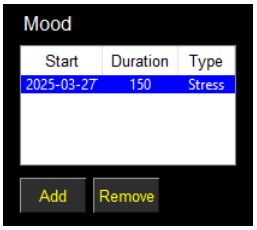


Figura 4.70: Pantalla de estado anímico (estrés).

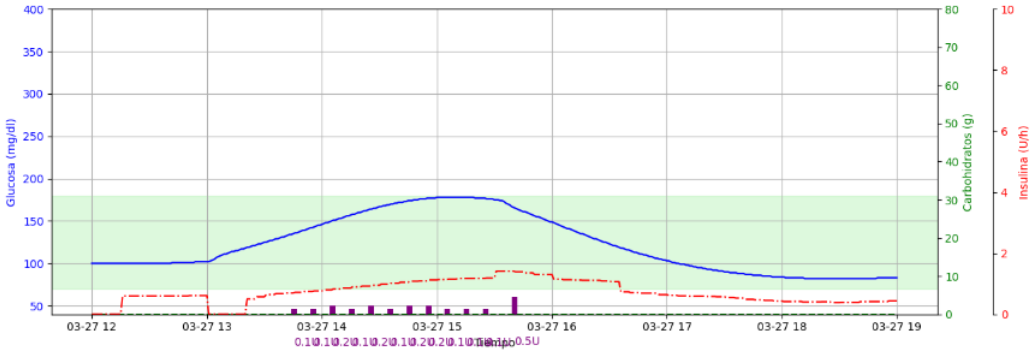


Figura 4.71: Evolución completa de glucosa e insulina para el caso de estrés.

Para las situaciones de estrés, el algoritmo incrementa en 20 el objetivo de glucosa. De esta forma, si se calculan correcciones, estas no son tan agresivas, ya que una corrección grande debía a estrés (y no ha comida) puede conllevar a una hipoglucemia posterior para cuando el cuerpo recupera su sensibilidad normal. Además, durante los primeros 30 minutos del estrés, el algoritmo de la bomba de insulina considera que la sensibilidad es menor, un 90 % de la sensibilidad nominal.

Una traza de un momento durante el estrés:

```
==============Basal delivery calculation=================
-> Time     = 2025-03-27 13:25:23.487835
-> TimeOld = 2025-03-27 13:20:23.487835
-> Glucose = 116 mg/dL
-> Error = -4.0000 mg/dL
-> P: error = -12.0 mg/dL,           rate = -0.08 U/h
-> I: error = 0 mg/dL * h,  rate = 0.0 U/h
-> D: error = 42.0 mg/dL / h,  rate = 0.06 U/h
-> Basal Rate = 0.5727 U/h   (Nominal was 0.6 U/h)
-> Final Basal Rate = 0.5727 U/h
Basal saved: {'start': '2025-03-27T13:25:23', 'duration': 5.0, 'iir':
    0.5727, 'type': 'Basal'}
-> Basal = 0.05 U
***************Correction bolus calculation*************:
-> Current BG = 116 mg/dL
-> Nominal Target BG = 100 mg/dL
```

```
16 ->  Target BG = 120.0 mg/dL
17 -> Active Insulin = 0.0000 U
18 -> Effective Active Insulin = 0.0000 U
19 -> Basal Insulin = 0.05 U
20 -> Effective Basal Insulin = 0.0500 U
21 -> Correction bolus = -4.0 mg/dL / (49.5 mg/dL/U) - (0.0000 U) - (0.05
     U) = -0.1308 U
22 -> Apply correction: True
23 -> Effective correction bolus: -0.1000
24
```

Código 4.18: Estado anímico (I).

La Figura 4.72 muestra las variables de estado en esta condición, y la Figura 4.73 refleja la evolución de los parámetros fisiológicos. Disminuye la sensibilidad SIE y aumenta EGP0. Se ve que rápidamente vuelven a sus valores nominales, por lo que una corrección considerando que el objetivo es de 100 mg/dL, podría ser una cantidad de insulina bastante considerable y posteriormente llevar al paciente a hipoglucemia.

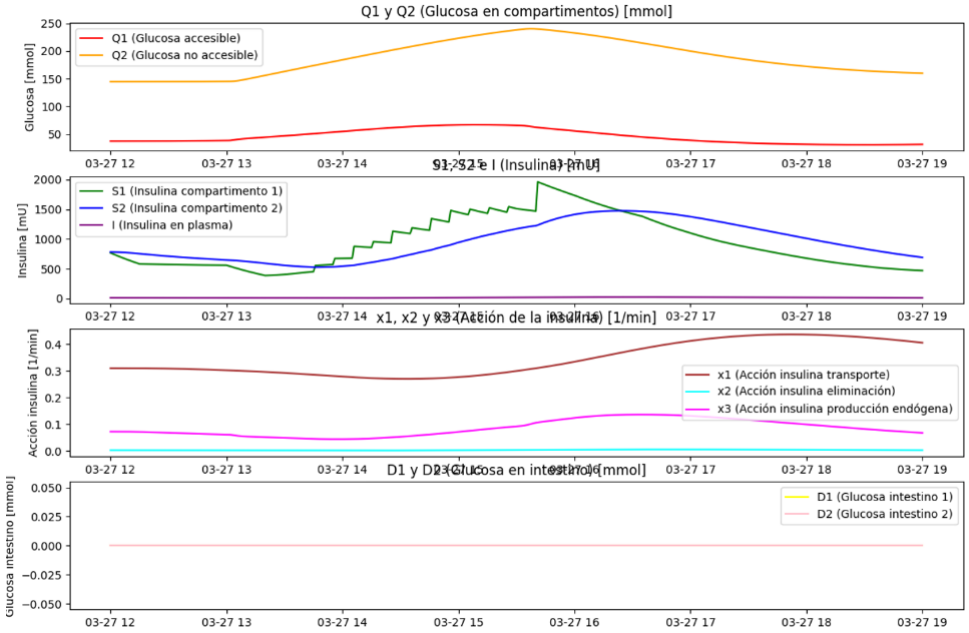


Figura 4.72: Evolución completa de variables de estado para el caso de estrés.

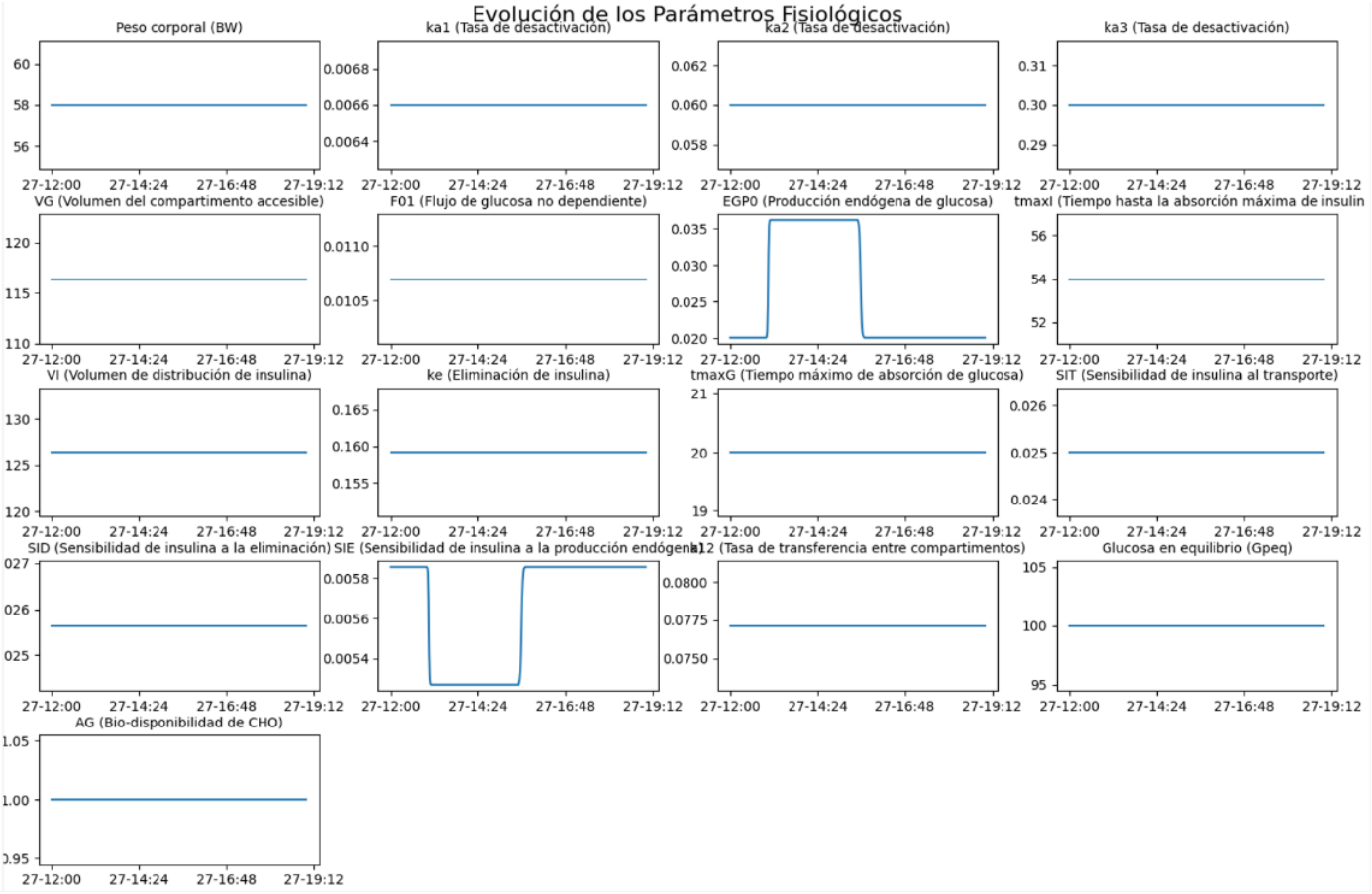


Figura 4.73: Evolución completa de parámetros fisiológicos para el caso de estrés.

En contraste, en la Figura 4.74, se ilustra la pantalla de estado anímico cambiando la situación de estrés por relajación.

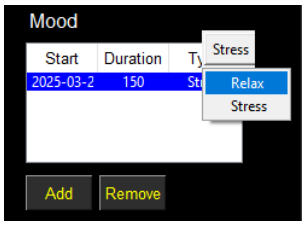


Figura 4.74: Pantalla de estado anímico (relajación).

En las Figuras 4.75, 4.76 y 4.77, se observan los cambios en glucosa, insulina, variables de estado y parámetros fisiológicos durante esta condición. Aquí, el algoritmo considera que la sensibilidad a la insulina aumenta un 10 %.

Una traza para esta situación es la siguiente:

```
==============Basal delivery calculation=================
-> Time     = 2025-03-27 14:39:11.096096
-> TimeOld  = 2025-03-27 14:33:51.096096
-> Glucose  = 112 mg/dL
-> Error    = 12.0000 mg/dL
-> P: error = 12.0 mg/dL,          rate = 0.08 U/h
-> I: error = 0 mg/dL * h,   rate = 0.0 U/h
```

```
-> D: error = 8.44 mg/dL / h,  rate = 0.01 U/h
-> Basal Rate = 0.6954 U/h   (Nominal was 0.6 U/h)
-> Final Basal Rate = 0.6954 U/h
Basal saved: {'start': '2025-03-27T14:39:11', 'duration': 5.3, 'iir':
    0.6954, 'type': 'Basal'}
-> Basal = 0.06 U
**********Correction bolus calculation**********:
-> Current BG = 112 mg/dL
-> Nominal Target BG = 100 mg/dL
->  Target BG = 100.0 mg/dL
-> Active Insulin = 0.0704 U
-> Effective Active Insulin = 0.0704 U
-> Basal Insulin = 0.06 U
-> Effective Basal Insulin = 0.0600 U
-> Correction bolus = 12.0 mg/dL / (60.50000000000001 mg/dL/U) - (0.07
    04 U) - (0.06 U) = 0.0679 U
-> Apply correction: True
-> Effective correction bolus: 0.1000
Correction bolus saved: {'start': '2025-03-27T14:39:11', 'i': 0.1, '
    type': 'Correction'}

```

Código 4.19: Estado anímico (II).

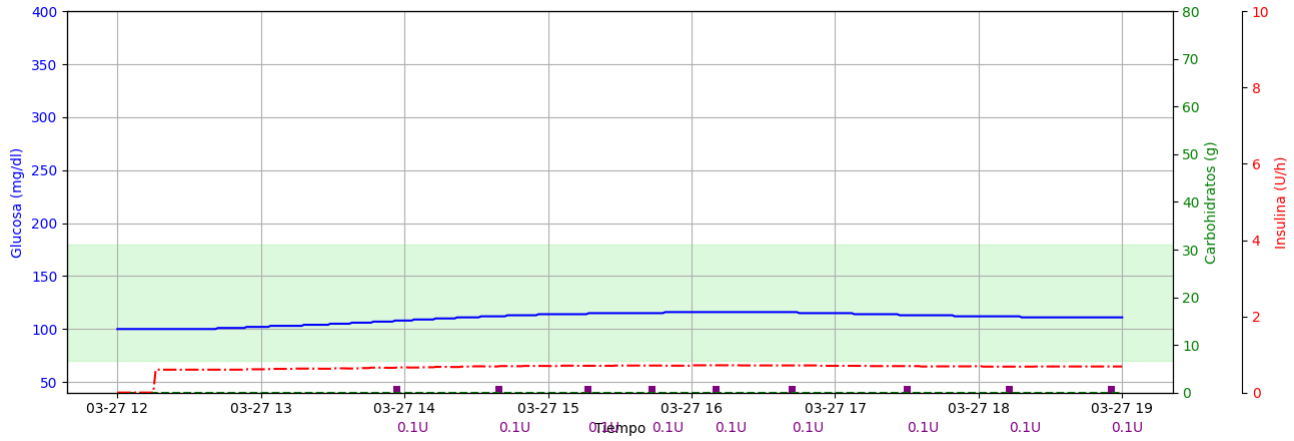


Figura 4.75: Evolución completa de glucosa e insulina para el caso de relajación.

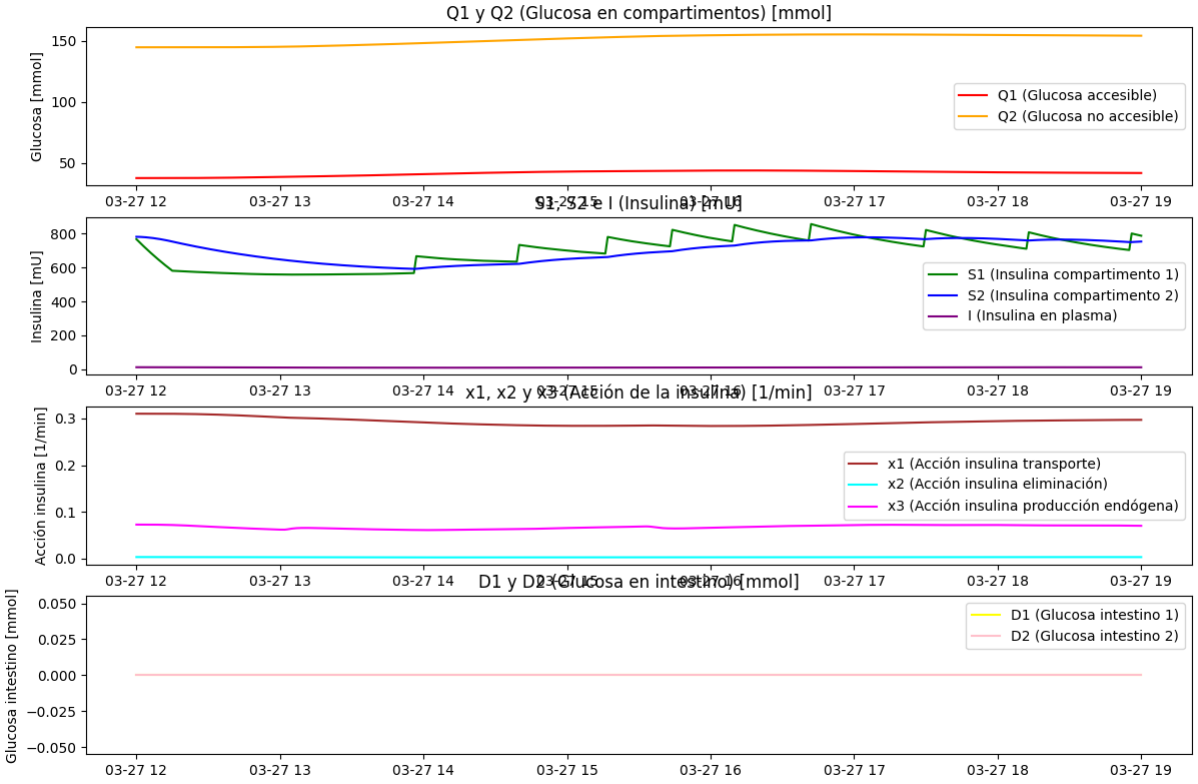


Figura 4.76: Evolución completa de variables de estado para el caso de relajación.

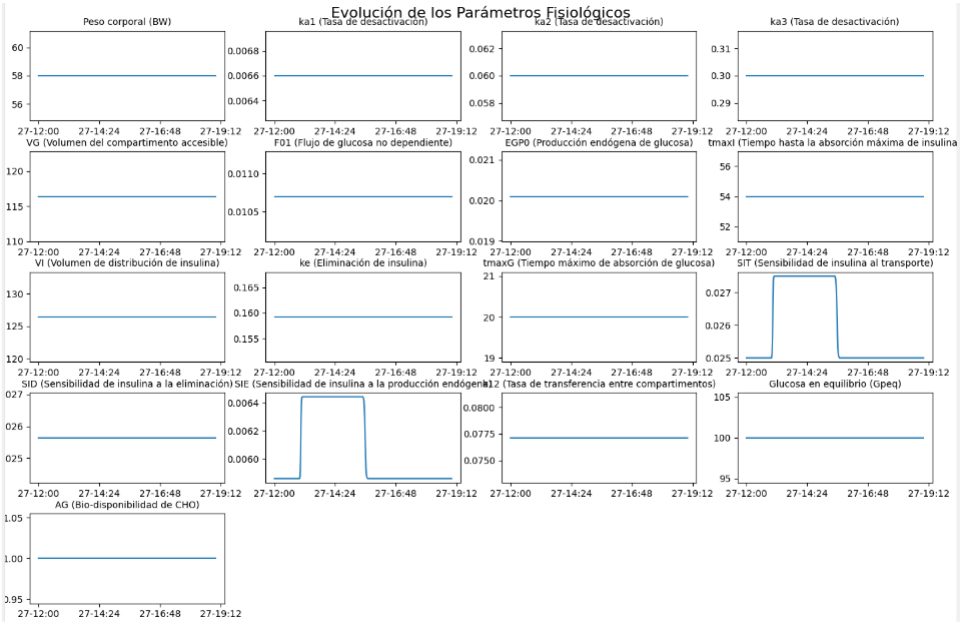


Figura 4.77: Evolución completa de parámetros fisiológicos para el caso de relajación.

4.6.2.5. Cambio de reservorio de insulina

Finalmente, en cuanto al cambio de reservorio de insulina y la duración del efecto máximo de insulina, las Figuras 4.78 y 4.79 muestran las pantallas dedicadas a configurar estas variables.

Normalmente, a partir de 2.5 días la insulina comienza a hacer efecto. Si el reservorio se cambia cada 3 días, habrá medio día durante el cual la eficacia de la insulina es menor y por tanto se necesita suministrar un poco más.

Figura 4.78: Pantalla de días de máximo efecto de insulina.

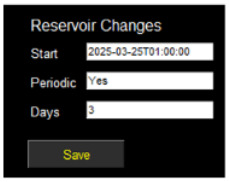


Figura 4.79: Pantalla de cambio de reservorio.

En la Figura 4.80, se visualiza la evolución de glucosa, insulina y carbohidratos cuando se cambió el reservorio el día 25 de marzo a la 1h de la madrugada. En este caso, la insulina tiene máximo efecto hasta el día 27 a las 13h. Desde las 13h hasta las 01h del día siguiente, la insulina va perdiendo efecto. Esta simulación se compara con la de la Figura 4.81, que muestra la evolución cuando la insulina está en pleno efecto, ya que el reservorio se cambió el día 27 de marzo a las 00h.

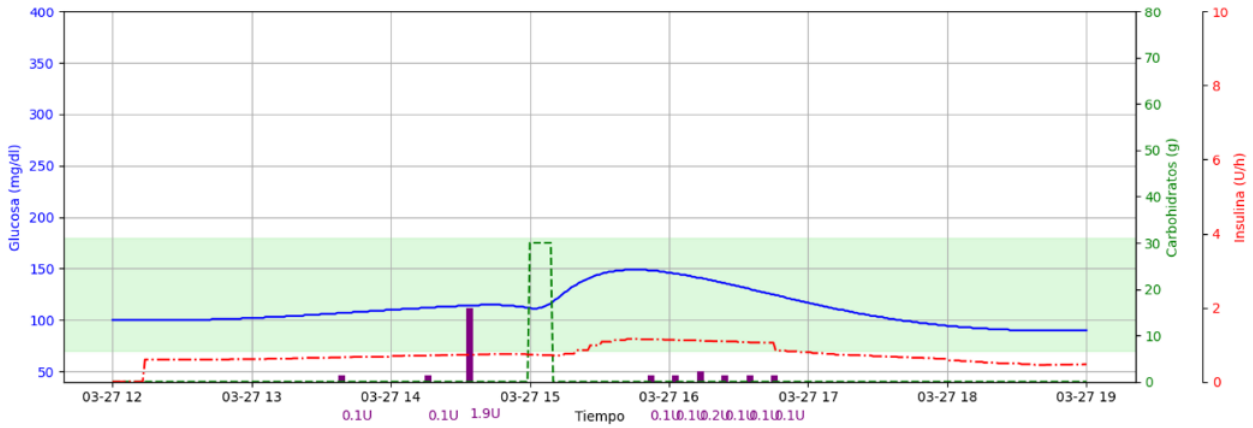


Figura 4.80: Evolución completa de glucosa, insulina y carbohidratos para el caso de insulina que ha perdido efecto.

Una traza para el caso de insulina con menos efecto:

```
1
2 ==============Basal delivery calculation=================
3 -> Time    = 2025-03-27 16:13:11.096096
4 -> TimeOld = 2025-03-27 16:07:51.096096
5 -> Glucose = 145 mg/dL
6 -> Error = 45.0000 mg/dL
7 -> P: error = 45.0 mg/dL,          rate = 0.32 U/h
8 -> I: error = 41.07 mg/dL * h,  rate = 0.12 U/h
9 -> D: error = -16.88 mg/dL / h,  rate = -0.02 U/h
```

```
-> Basal Rate = 1.0154 U/h   (Nominal was 0.6 U/h)
-> Final Basal Rate = 1.0971333262213778 U/h
Basal saved: {'start': '2025-03-27T16:13:11', 'duration': 5.3, 'iir':
    1.0971333262213778, 'type': 'Basal'}
-> Basal = 0.1 U
**********Correction bolus calculation**********:
-> Current BG = 145 mg/dL
-> Nominal Target BG = 100 mg/dL
->  Target BG = 100.0 mg/dL
-> Active Insulin = 0.6161 U
-> Effective Active Insulin = 0.5702 U
-> Basal Insulin = 0.1 U
-> Effective Basal Insulin = 0.0926 U
-> Correction bolus = 45.0 mg/dL / (55.0 mg/dL/U) - (0.6161 U) - (0.09
    255028315447546 U) = 0.1554 U
-> Apply correction: True
-> Effective correction bolus: 0.2000
Correction bolus saved: {'start': '2025-03-27T16:13:11', 'i': 0.2, '
    type': 'Correction'}


```

Código 4.20: Cambio de reservorio de insulina.

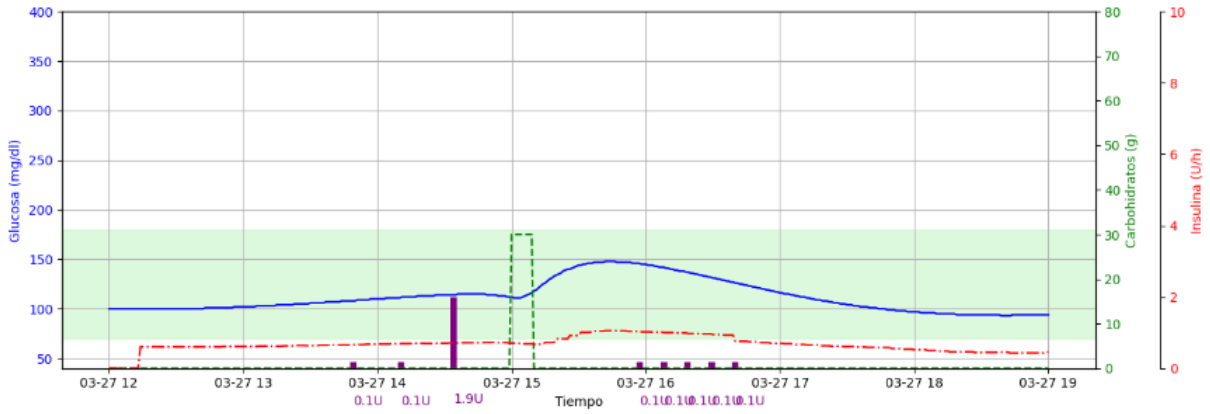


Figura 4.81: Evolución completa de glucosa, insulina y carbohidratos para el caso de insulina con pleno efecto.

Se puede ver como la evolución de la glucosa es análoga en ambos caos, pero para el caso donde la insulina tiene menos efecto, se calculan varias correcciones más. Incluso una es de 0.2 U.

El Capítulo 7 contiene una sección dedicada a la validación de este modelo de diabetes.

4.7. Conclusiones

En este capítulo se ha desarrollado el algoritmo de control que regula el funcionamiento de una bomba de insulina inteligente. Este algoritmo se basa en un controlador PID adaptativo para el cálculo de insulina basal y bolos de corrección, cuyos parámetros se ajustan dinámicamente en

función de la estabilidad de la glucosa y el tiempo en rango del paciente. Se han detallado los mecanismos de decisión para la administración de bolos prandiales, incluyendo la incorporación de UGPs, y se han considerado múltiples eventos externos que modifican la sensibilidad a la insulina o los valores objetivo.

Se ha presentado además la arquitectura modular del sistema, con una implementación clara y extensible que permite su integración en dispositivos físicos como una Raspberry Pi. Finalmente, se han realizado pruebas de simulación para evaluar el comportamiento del sistema ante distintos escenarios, mostrando la capacidad del algoritmo para responder adecuadamente a cambios en la dieta, actividad física o estados fisiológicos.

Estos resultados permiten concluir que el algoritmo implementado constituye una base sólida para la automatización parcial del tratamiento de la diabetes tipo 1, y sienta las bases para su validación con datos reales y posterior integración en un entorno clínico más completo.

Capítulo 5

Integración del algoritmo en dispositivo

5.1. Introducción

En el capítulo anterior se ha descrito en detalle el algoritmo de la bomba de insulina, incluyendo la lógica de control para la administración de insulina basal, bolos de corrección y bolos por comida. En este capítulo se aborda la integración completa de dicho algoritmo en un dispositivo físico, permitiendo su ejecución en condiciones similares a las de una bomba de insulina real.

Para ello, se ha utilizado una Raspberry Pi con pantalla táctil como dispositivo embebido que simula el comportamiento de una bomba de insulina. Esta integración incluye tanto el despliegue del algoritmo como el desarrollo de una interfaz gráfica de usuario que permite visualizar información, configurar parámetros y registrar eventos manuales. La Raspberry Pi se conecta a un PC que simula o descarga datos de glucosa, replicando así la interacción entre un sensor continuo de glucosa y una bomba real.

El capítulo describe el flujo de datos entre componentes, el diseño del entorno de desarrollo, la compartición de archivos mediante Samba, la arquitectura del programa implementado en Python y los principales elementos de la interfaz de usuario.

La estructura de este capítulo es la siguiente:

- En la Sección 5.2, se explica el **diagrama de flujo de datos**, desde el origen de los datos de glucosa hasta su procesamiento por el algoritmo de la bomba.
- En la Sección 5.3, se describe la **integración física en la Raspberry Pi**, incluyendo la preparación del entorno, los requisitos de hardware y la configuración de red para la compartición de archivos.
- En la Sección 5.4, se analiza la **arquitectura del programa**, detallando la organización modular del código, las principales dependencias y el funcionamiento del bucle principal de actualización.
- En la Sección 5.5, se presenta la **interfaz gráfica**, ilustrando los distintos casos de uso y pantallas disponibles para la interacción del usuario con la bomba simulada.

5.2. Diagrama de flujo de datos

Se explica a continuación el flujo de datos entre los datos de glucosa del paciente diabético (modelo de diabetes o datos reales) y el algoritmo de la bomba de insulina, que se integra en

una Raspberry Pi como dispositivo para la administración de insulina. La Raspberry Pi actúa como la bomba de insulina en sí misma, ejecutando el algoritmo de control basado en los datos de glucosa del paciente y en la insulina activa explicado en el Capítulo 4.

En la Figura 5.1 se muestra un diagrama del flujo de datos para el algoritmo de la bomba de insulina.

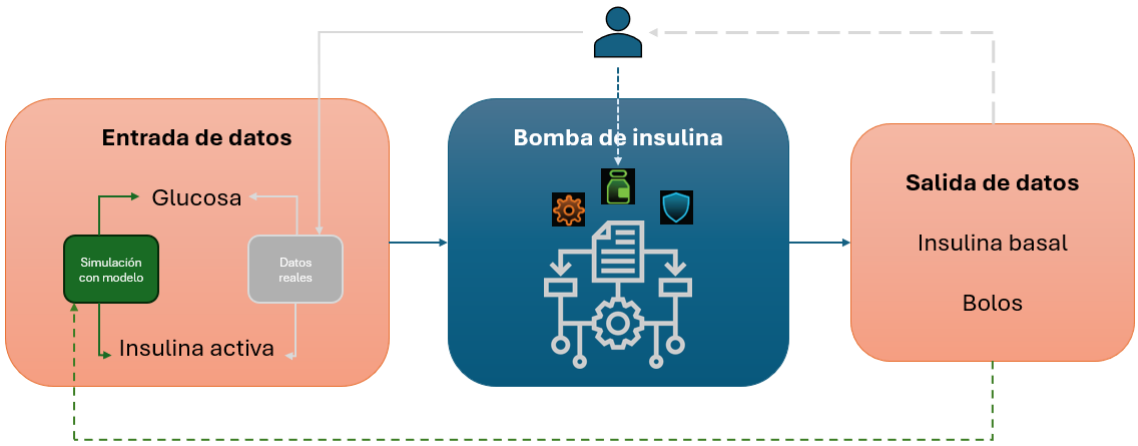


Figura 5.1: Diagrama de flujo de datos.

Entrada de datos

Los datos de entrada provienen de dos fuentes principales:

- Niveles de glucosa en sangre.
- Insulina activa, que se calcula a partir de las dosis previas de insulina administradas y el modelo farmacocinético de absorción.

Ambos, se obtienen de uno de los dos siguientes:

1. Simulación: Datos generados por el modelo de diabetes implementado en un PC, explicado en el Capítulo 3. Se representa con el cuadro de color verde oscuro en la Figura 5.1.
2. Datos reales: Recuperados desde la nube de CareLink, a la que el ordenador tiene acceso. Se explica en el siguiente Capítulo 6, y por tanto se representa sin color en la Figura 5.1 (más adelante se explicará el flujo de datos correspondiente a la adquisición de estos datos reales).

Ambos datos de entrada incluyen una estampa de tiempo, simulando el comportamiento de un sensor de glucosa en tiempo real.

Procesamiento en la bomba de insulina

La Raspberry Pi ejecuta la aplicación que contiene el algoritmo de bucle híbrido, el cual procesa los datos de glucosa y de insulina activa para determinar la cantidad de insulina a administrar,

en función de los parámetros y configuración del paciente. Este algoritmo ajusta dinámicamente la dosis considerando factores adicionales como la ingesta de carbohidratos, y eventos el estado anímico o la actividad física.

Salida de datos

Las salidas del sistema son la tasa de infusión de insulina (insulina basal) y los bolos de corrección o por comidas, que sirve como input (inyección) para el paciente o para el simulador del modelo de diabetes. En el caso de la simulación, la administración de insulina modifica la evolución de la glucosa, generando nuevos datos de entrada en el siguiente ciclo del algoritmo.

5.3. Raspberry Pi como bomba de insulina

La Raspberry Pi con pantalla táctil es utilizada para simular la bomba de insulina, permitiendo tanto la administración automática como la interacción manual con el usuario. En este proyecto se emplea una Raspberry Pi 3 Modelo B+, que incorpora un procesador Broadcom BCM2837B0 de cuatro núcleos a 1.4 GHz, 1 GB de memoria RAM LPDDR2 y conectividad inalámbrica Wi-Fi de doble banda (2.4 GHz y 5 GHz). Gracias a sus reducidas dimensiones (85.6 mm x 56.5 mm) y a su bajo consumo energético (requiere una fuente de alimentación de 5 V y 2.5 A), esta placa resulta especialmente adecuada para su integración en un dispositivo médico portátil, como una bomba de insulina automatizada. Además, su bajo coste y compatibilidad con Linux la hacen ideal para la implementación del sistema. A continuación se muestra el diagrama de la Figura 5.2 incluido en la estructura PC-usuario-Raspberry Pi.

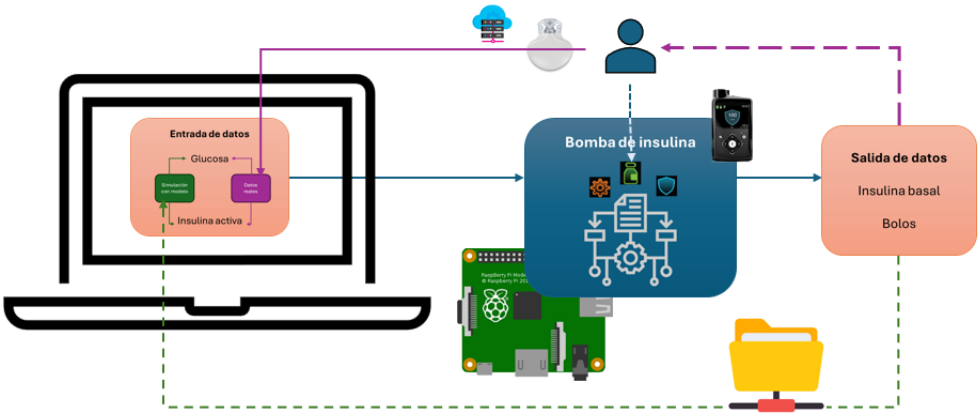


Figura 5.2: Raspberry Pi como bomba de insulina.

5.3.1. Preparación del entorno de desarrollo

Para garantizar la correcta ejecución del software en la Raspberry Pi, se llevan a cabo los siguientes pasos:

- Configuración del hardware: Instalación del sistema operativo (véase el Apéndice C para más detalles sobre la instalación) y configuración de la pantalla táctil y teclado.

- Implementación del algoritmo: desarrollo del software en Python para el procesamiento de datos y la administración de insulina.
- Interfaz gráfica: Creación de una aplicación con interfaz gráfica diseñada para la Raspberry Pi, que contiene el algoritmo de la bomba de insulina, que permita visualizar los niveles de glucosa y la insulina administrada, facilitando la interacción con el usuario si es necesario.

5.3.2. Medios necesarios

- PC estándar: Para el desarrollo del software y simulación del modelo de diabetes.
- Raspberry Pi con pantalla táctil: Para la ejecución del algoritmo y la visualización de datos.
- Sistema operativo Linux: Por su estabilidad y compatibilidad con herramientas de desarrollo de la Raspberry Pi, instalado en una tarjeta Micro SD.

Figura 5.3: Raspberry Pi con pantalla táctil, bomba de Medtronic a la izquierda.

5.3.3. Compartión de datos

Tanto el PC (datos de glucosa del sensor o del modelo de diabetes) como la Raspberry Pi son dos dispositivos diferentes, ya que el PC hace de recogida de datos (lo que se correspondería con un sensor de monitorización continua) y la Raspberry Pi sería la bomba de insulina.

El sensor Guardian Connect de Medtronic utiliza tecnología Bluetooth Low Energy (BLE) para transmitir datos de glucosa en tiempo real a dispositivos móviles compatibles. Cada cinco minutos, el sensor mide los niveles de glucosa en el líquido intersticial y envía esta información al dispositivo móvil del usuario a través de una conexión inalámbrica segura. La aplicación Guardian™ Connect instalada en el dispositivo móvil recibe y muestra estas lecturas, permitiendo a los usuarios monitorear continuamente sus niveles de glucosa y recibir alertas predictivas de posibles episodios de hipoglucemia o hiperglucemia con hasta 60 minutos de anticipación.

El transmisor del sensor Guardian™ Connect es pequeño y discreto, y se conecta directamente al sensor para enviar las lecturas de glucosa al dispositivo móvil. Es importante destacar que este

transmisor está diseñado específicamente para funcionar con la aplicación Guardian™ Connect y no es compatible con otros sistemas de Medtronic, como las bombas de insulina MiniMed de la serie 600, que utilizan un transmisor diferente sin conectividad Bluetooth. Medtronic Diabetes

La implementación de BLE en el sensor Guardian Connect permite una comunicación eficiente y de bajo consumo energético, esencial para dispositivos médicos que requieren transmisiones frecuentes de datos sin agotar rápidamente la batería del transmisor o del dispositivo móvil. Además, la conexión Bluetooth facilita la configuración y el uso del sistema, eliminando la necesidad de receptores adicionales y permitiendo a los usuarios utilizar sus propios dispositivos móviles para el monitoreo continuo de glucosa.

Para simular esta conexión entre los datos de glucosa y la bomba, se realiza integración de datos entre la Raspberry Pi y el ordenador que simula los niveles de glucosa, mediante un mecanismo de compartición de archivos a través de Samba (véase el Apéndice C para su instalación y configuración), permitiendo que ambos dispositivos accedan a la misma carpeta compartida en la red.

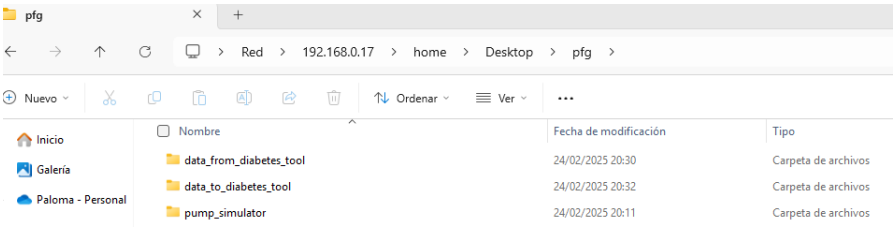


Figura 5.4: Contenido de la carpeta compartida.

5.4. Arquitectura del programa

En esta sección se comenta la arquitectura del programa a alto nivel, así como dependencias entre módulos y librerías externas. En la Figura 5.5 se presenta un diagrama completo de los diferentes módulos implicados, relaciones más importantes entre ellos y algunas funciones relevantes. Más adelante se muestran imágenes de los grupos más en detalle.

El programa main encargado de arrancar el programa que simula la bomba de insulina se encuentra en `main.py`, representado por un triángulo rojo (botón play) en el diagrama. Las librerías más utilizadas están representadas en la parte superior izquierda. Estas son `tkinter`, `json`, `datetime`, `time` y `matplotlib`. Sobre cada módulo se encuentra una abreviación de las librerías más relevantes (tki, json, dt, t y mplt respectivamente).

El archivo `data.json` (naranja a la derecha del diagrama) es el archivo que contiene los datos de glucosa del paciente y que son leídos por el programa. El archivo `inputs.json` es el archivo que contiene los datos de insulina (tanto basal como bolos sueltos por comida o correcciones). En este archivo se añaden todos los datos de insulina que la bomba va calculando.

El paquete `tkinter` («interfaz Tk») es la interfaz por defecto de Python para el kit de herramientas de GUI Tk. Se ha utilizado este paquete dado que el objetivo es integrar el algoritmo con una interfaz de usuario fácil de utilizar e intuitiva en una Raspberry Pi.

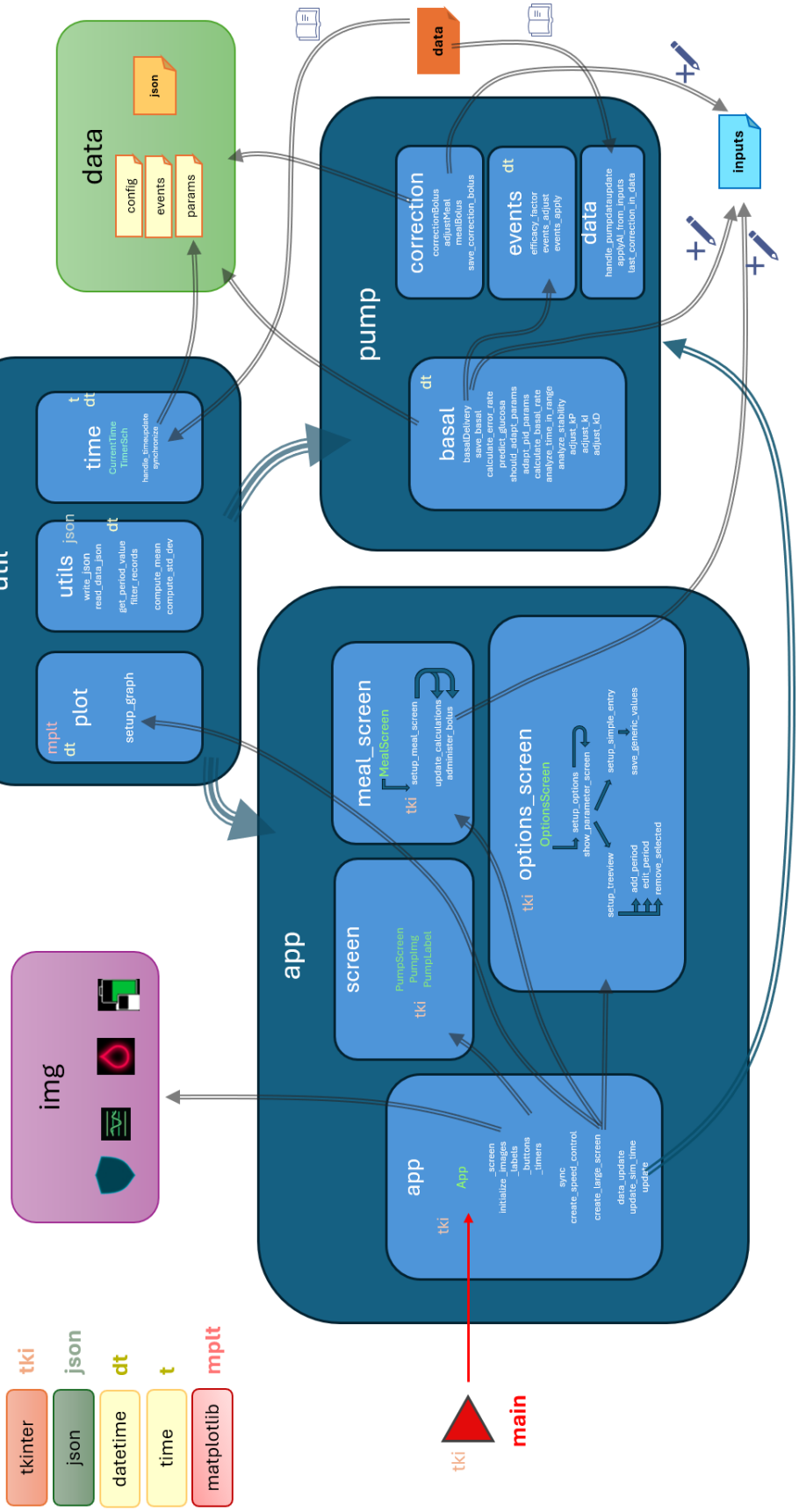


Figura 5.5: Arquitectura del programa.

El grupo `app`, representado en el diagrama de la Figura 5.6 contiene los scripts `app.py`, `screen.py`, `meal_screen.py` y `options_screen.py`. Son los encargados de conformar toda la interfaz gráfica de la aplicación, así como sus diferentes pantallas y vistas. Al ejecutarse el programa, se genera una instancia de la clase `App` que inicializa las diferentes parte de la GUI de la bomba con los métodos `initialize_screen`, `_images` etc. Estos métodos utilizan las imágenes almacenadas en la carpeta `img`. La pantalla principal, las imágenes y los labels se inicializan con isntancias de las clases `PumpScree`, `PumpImg` y `PumpLabel` respectivamente (clases dentro del script `screen.py`).

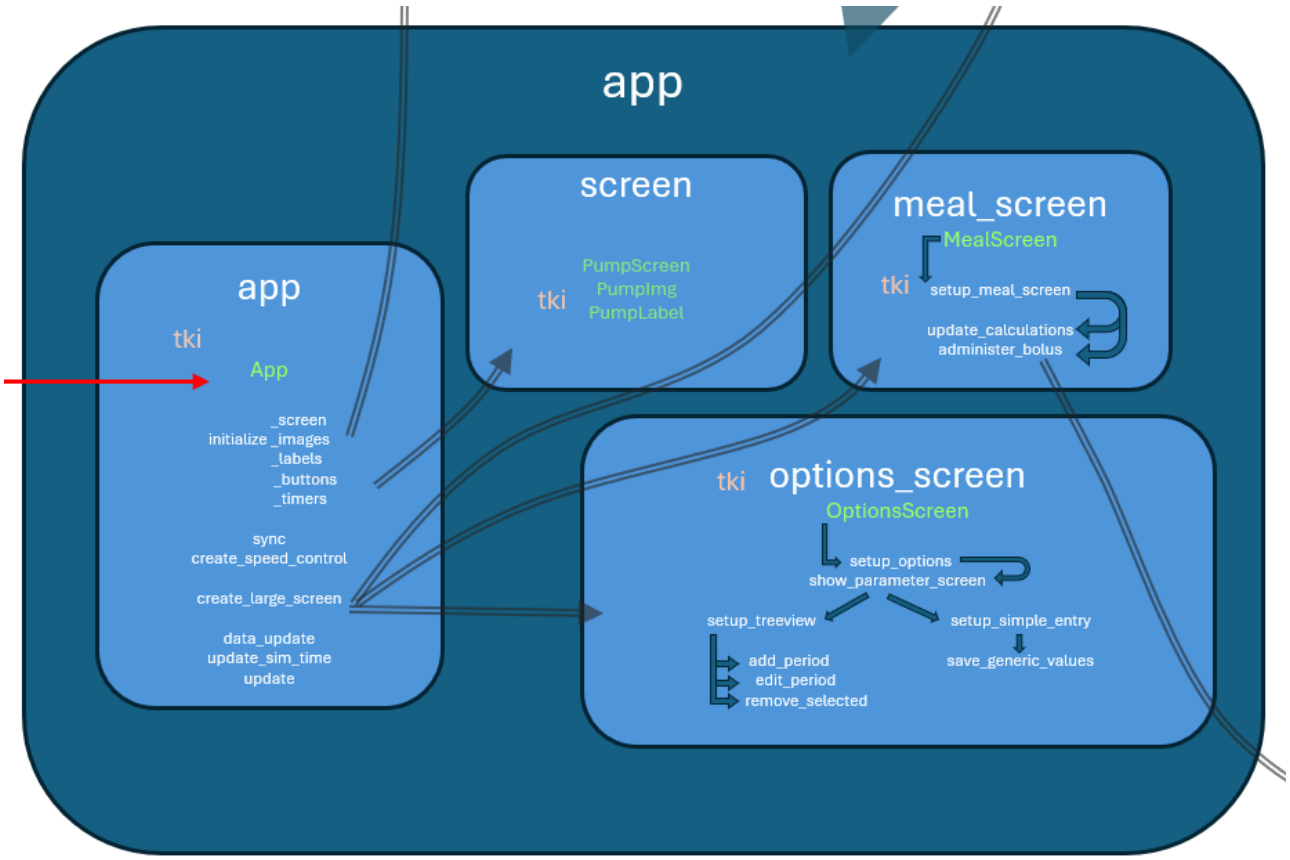


Figura 5.6: Detalle de app.

Se generan también los botones y las pantallas desplegadas al seleccionar cada uno de los botones.

- El botón de gráfica genera una gráfica ayudada del método `setup_graph`, del script `plot.py` (que se encuentra en otro grupo de scripts, en el de util).

- El botón de comidas, al seleccionarse, crea una instancia de la clase `MealScreen`. El método `setup_meal_screen` es el encargado de configurar esta pantalla, donde aparecen los campos editables de Hidratos de Carbono y UGPs. Cada vez que se edita alguno de estos campos, se ejecuta el método `update_calculations` automáticamente. Se esta forma se consigue un comportamiento dinámico donde el usuario puede ver la cantidad de insulina que se administraría al hacer click en el botón de Administrar bolo. Este botón llama a `administrer_bolus` y escribe en el archivo inputs.json el bolo por comida.

- Tanto el botón de configuración como el de eventos, al seleccionarse, crean una instancia de la clase `OptionsScreen`. El método `setup_options` es el encargado de configurar la pantalla, donde aparecen las listas de eventos y parámetros configurables. El método `show_parameters_screen` es el encargado de configurar estas pantallas en función del tipo de parámetro.

El método `update_sim_time` es el responsable de actualizar el reloj de la bomba de insulina, que depende de si se ha sincronizado (con el método `sync` de la clase `App`, que llama al método `synchronize` del módulo `time`) o no con los datos de `data.json`, y de actualizar el tiempo en función de la velocidad del reloj (que se puede modificar con el método `create_speed_control`). Para ello, `update_sim_time` opera con una instancia generada al principio (en `initialize_timers`) de `TimerSch`, que también se encuentra en el módulo `time` (grupo `util`).

Por último, el método `update` es el encargado de llamar a todos los métodos necesarios de la bomba, del grupo `pump`. Concretamente, son llamados:

1. `handle_pumpdataupdate`: del módulo `data`, que lee los datos de glucosa de `data.json`.
2. `basalDelivery`: del módulo `basal`, que calcula la basal y la escribe en `inputs.json` si procede.
3. `predict_glucose`: del módulo `basal`, que predice la glucosa para dentro de 30 minutos indicando si va a haber hipoglucemia o no.
4. `correction_bolus`: del módulo `correction`, que calcula el bolo de corrección y lo escribe en `inputs.json` si procede.
5. `handle_timeupdate`: del módulo `time`, que actualiza la hora de la bomba.

Estos métodos se explican junto con su diagrama en la Figura 4.1 del Capítulo 4.

El grupo `util`, representado en la Figura 5.7 contiene los scripts `plot.py`, `utils.py` y `time.py`.

- `plot`: contiene el método `setup_graph`
- `time`: aparte de los métodos `handle_timeupdate` y `synchronize`, contiene las clases:
  - `CurrentTime`: reloj de la bomba de insulina, que indica y setea cuando sea necesaria la hora del reloj
  - `TimerSch`: es el programador de temporizadores, que se encarga de indicar cuándo hay que actualizar la hora del reloj y cuándo hay que leer un dato de glucosa de `data.json` para mostrar en la bomba y hacer después los cálculos pertinentes.
- `utils`: contiene funciones auxiliares de escritura y lectura de json, entre otras.

En general, los métodos de este grupo son utilizados tanto por los del grupo `app` como `pump`.

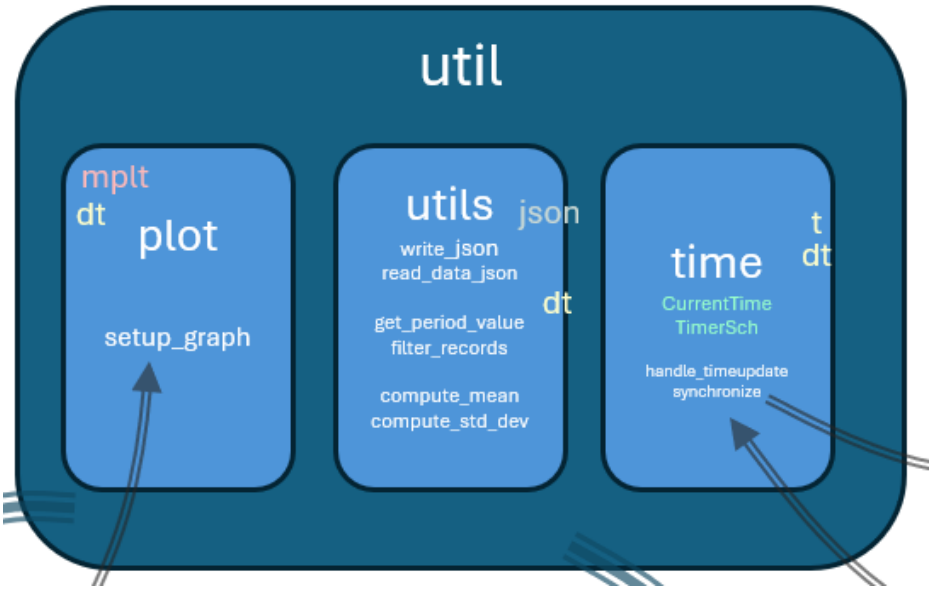


Figura 5.7: Detalle de util.

5.4.1. Pseudocódigo del bucle principal de actualización

El Pseudocódigo 5.1 y el Pseudocódigo 5.2 muestran el bucle principal de actualización del algoritmo de la bomba de insulina, que se ejecuta continuamente para procesar los datos de glucosa y administrar insulina según sea necesario. Este bucle es el corazón del sistema, donde se gestionan las actualizaciones de datos, se calculan las dosis de insulina y se actualiza la interfaz gráfica.

Pseudocódigo 5.1 Bucle principal de actualización (I).

```
Función update():
/* Procesar nuevos datos de la bomba si están disponibles */
Si is_pump_data_update entonces:
    lastSG, lastUpdateTm, activeInsulin ← handle_pumpdataupdate()
/* Si faltan datos, hacer solo actualización visual */
    Si lastUpdateTm = None ∨ lastSG = None ∨ activeInsulin = None entonces:
        data_update()
        dateOld ← date.getTime()
        is_pump_data_update ← False
    Si no:
/* Calcular tasa basal con PID adaptativo */
        basal, bR, eIntegral, errorRate ← basalDelivery(date.getTime(),
dateOld, lastSG, errorHistory, eIntegral)
```

Pseudocódigo 5.2 Bucle principal de actualización (II).

```
/* Actualización visual y datos generales */
        data_update()
/* Predecir glucosa futura */
        predicted_glucose ← predict_glucose(lastSG, errorRate)
/* Verificar si el último bolo corrector ya fue considerado */
        consider ← last_correcion_in_data(dateOld, activeInsulin,
lastActiveInsulin)
/* Aplicar bolo corrector si la predicción no indica hipoglucemia y el anterior ya
fue aplicado */
        Si predicted_glucose >70 ∧ dateOld ≠ None ∧ consider entonces:
            correctionBolus(date.getTime(), lastSG, activeInsulin,
basal)
/* Actualizar memoria del PID y estado anterior */
        eIntegral ← eIntegral
        dateOld ← date.getTime()
        lastActiveInsulin ← activeInsulin
        is_pump_data_update ← False
/* Actualizar visualización de la hora si es necesario */
Si is_time_update entonces:
    handle_timeupdate(date.getTime(), labelTime, labelLastData,
lastUpdateTm)
    is_time_update ← False
/* Reprogramar la próxima actualización */
update_id ← root.after(100, update())
```

5.5. Interfaz gráfica

En el Capítulo 4, en la Sección de 4.6 ya se incluían algunas imágenes de la interfaz gráfica de la bomba de insulina que se muestran en la pantalla física de la Raspberry Pi.

En el Apéndice B se incluye una guía de uso para la aplicación de la bomba de insulina, que funciona con el algoritmo explicado en 4 y con los módulos y pantallas mencionadas en el presente capítulo.

A continuación, se resumen brevemente varios casos de uso de la aplicación de la bomba de insulina, de los cuales muchos de ellos han servido para realizar la validación en 4.6.

En las figuras 5.8 y 5.9 se observa la ejecución del algoritmo de la bomba de insulina en la Raspberry Pi, junto con las trazas de la terminal de la DiabetesTool que hacen referencia a los archivos de la carpeta compartida.

En la pantalla mostrada en la Figura 5.10, se observa la interfaz inicial de la bomba sin ninguna

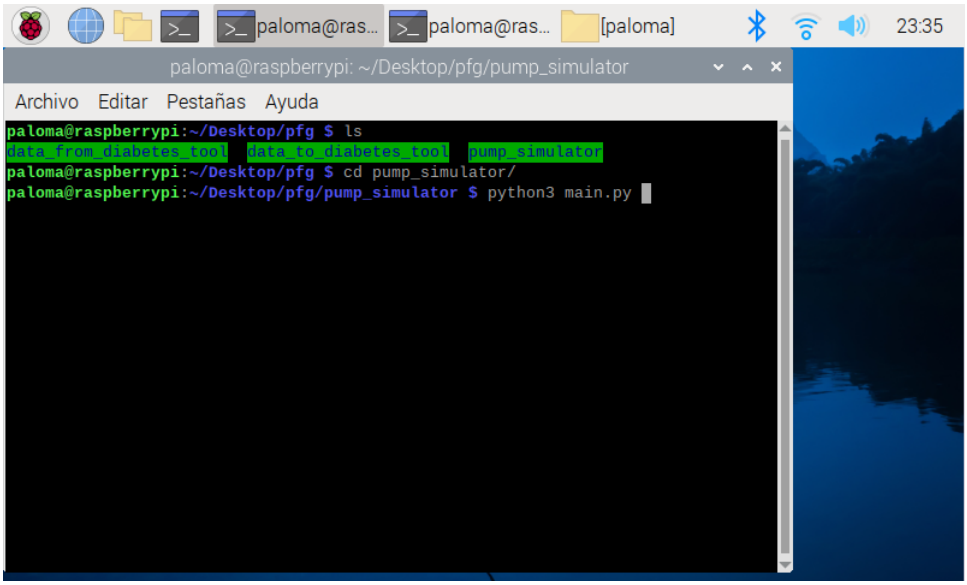


Figura 5.8: Ejecución del algoritmo de la bomba de insulina en la Raspberry Pi.

```
[2025-05-11 23:38:43:INFO] Usando inputs del archivo \\192.168.0.17\home\Desktop\pfg\data_to_diabetes_tool\inputs.json.
[2025-05-11 23:38:43:INFO] Escribir output data en archivo \\192.168.0.17\home\Desktop\pfg\data_from_diabetes_tool\data.json.
```

Figura 5.9: Trazas de la terminal de la DiabetesTool referenciando a los archivos de la carpeta compartida.

selección.

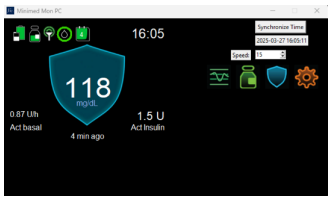


Figura 5.10: Pantalla de bomba sin selección.

Y en la pantalla mostrada en la Figura 5.11, se observa la interfaz y el algoritmo de la bomba corriendo en la Raspberry Pi, con el menú de comida seleccionado.

5.5.1. Gráfica

Al seleccionar la opción de "Gráfica", como se muestra en la Figura 5.12, se accede a una pantalla que permite visualizar la evolución de la glucosa.

5.5.2. Comidas

Cuando se desea ingresar o modificar las comidas, la interfaz se despliega tal y como se observa en la Figura 5.13. Esta opción permite al usuario configurar los HC y UGPs de la comida y administrar el bolo correspondiente.

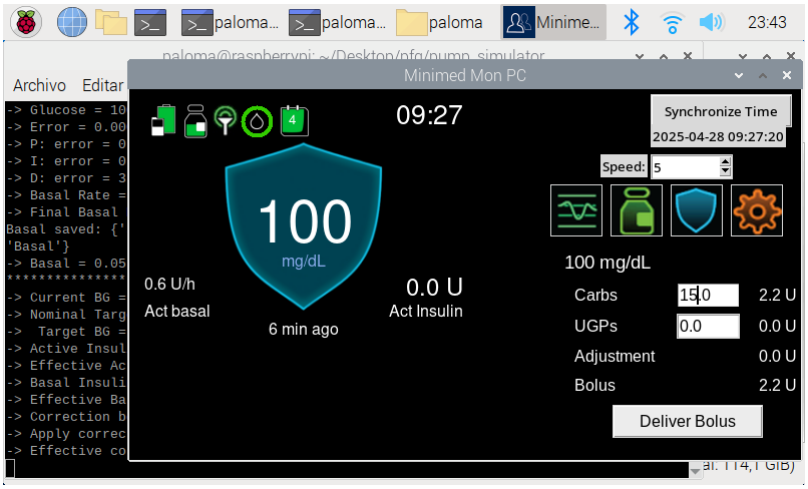


Figura 5.11: Pantalla de bomba en la Raspberry Pi.

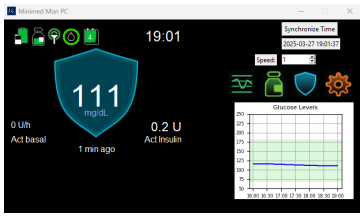


Figura 5.12: Pantalla de bomba, selección de gráfica.

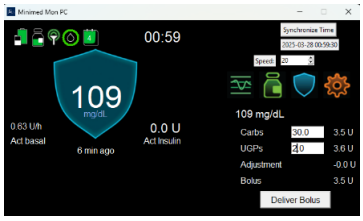


Figura 5.13: Pantalla de bomba, selección de comidas.

5.5.3. Eventos

La Figura 5.14 muestra la pantalla en la que se pueden gestionar los eventos relacionados con la bomba de insulina.

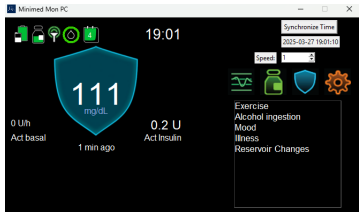


Figura 5.14: Pantalla de bomba, selección de eventos.

5.5.3.1. Deporte

La opción de gestionar el ejercicio físico está representada en la serie de pantallas de la Figura 5.15 a la Figura 5.17. En la Figura 5.15, se muestra la pantalla inicial de la sección de deportes, que está vacía hasta que se agrega un ejercicio. La Figura 5.16 muestra cómo se puede añadir un nuevo ejercicio o modificar el valor de un ejercicio existente, mientras que la Figura 5.18 ilustra cómo se puede borrar un ejercicio de la lista. Finalmente, la Figura 5.17 presenta la pantalla para añadir un nuevo ejercicio a la lista.

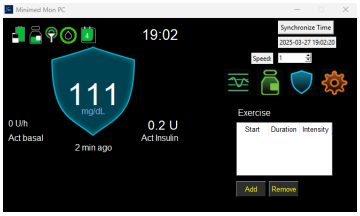


Figura 5.15: Pantalla de ejercicio, lista vacía.

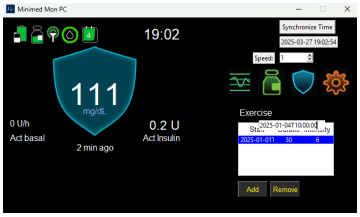


Figura 5.16: Pantalla de ejercicio, añadir ejercicio y modificar valor.

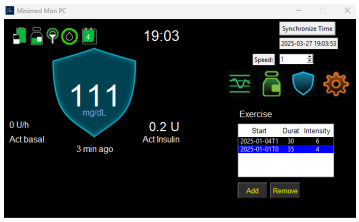


Figura 5.17: Pantalla de ejercicio, añadir nuevo ejercicio.

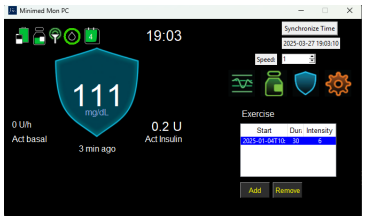


Figura 5.18: Pantalla de ejercicio, borrar ejercicio.

5.5.3.2. Alcohol

La Figura 5.19 muestra la pantalla de gestión de alcohol, donde el usuario puede ingresar información relacionada con el consumo de alcohol.

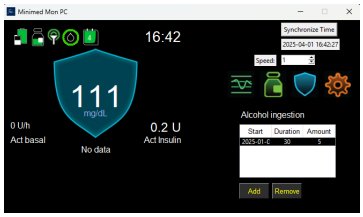


Figura 5.19: Pantalla de alcohol.

5.5.3.3. Estado anímico

La gestión del estado anímico se puede realizar en la pantalla de la Figura 5.20, donde el usuario puede seleccionar el estado anímico correspondiente. Además, la Figura 5.21 ilustra cómo se puede elegir un tipo específico de estado anímico de entre las opciones disponibles.

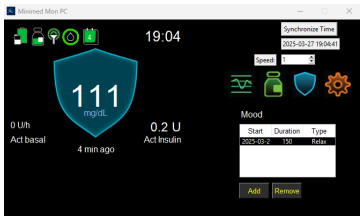


Figura 5.20: Pantalla de estado anímico.

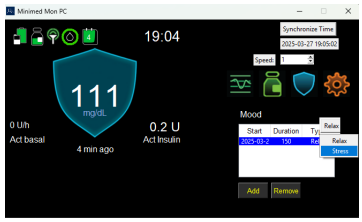


Figura 5.21: Pantalla de estado anímico, selección de tipo.

5.5.3.4. Enfermedad

La pantalla de selección de tipo de enfermedad está ilustrada en la Figura 5.22, donde el usuario puede seleccionar el tipo de enfermedad relevante para ajustar el tratamiento.

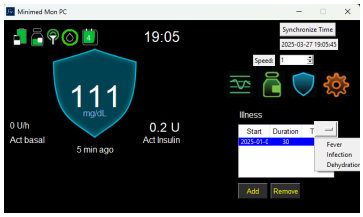


Figura 5.22: Pantalla de enfermedad, selección de tipo.

5.5.3.5. Cambio de reservorio de insulina

Cuando es necesario cambiar el reservorio de insulina, la Figura 5.23 muestra la pantalla correspondiente para realizar este ajuste.

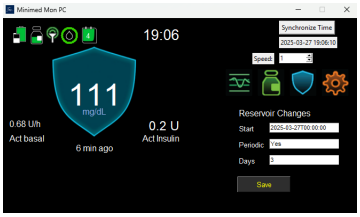


Figura 5.23: Pantalla de cambio de reservorio.

5.5.4. Configuración

La configuración general de la bomba de insulina está disponible en la pantalla de la Figura 5.24, que permite acceder a las opciones para personalizar diversos parámetros.

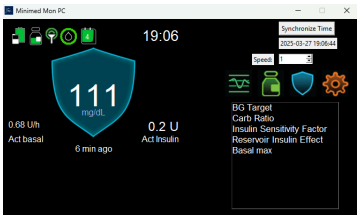


Figura 5.24: Pantalla de bomba, selección de configuración.

5.5.4.1. BG Target

La gestión de los objetivos de glucosa en sangre (BG Target) está disponible en las pantallas de la Figura 5.25 y Figura 5.26. La Figura 5.25 muestra la pantalla principal para configurar el objetivo de glucosa en sangre, mientras que la Figura 5.26 ilustra cómo se puede modificar este valor según las necesidades del usuario.

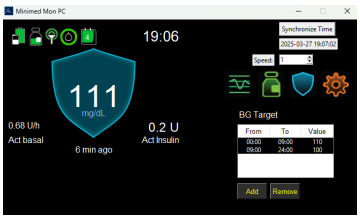


Figura 5.25: Pantalla de BG target.

5.5.4.2. Carb Ratio

La Figura 5.27 muestra la pantalla donde se puede ajustar el ratio de carbohidratos para calcular la cantidad de insulina necesaria en función de los carbohidratos consumidos.

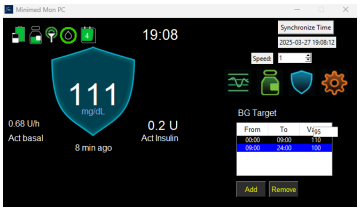


Figura 5.26: Pantalla de BG target, modificar valor.

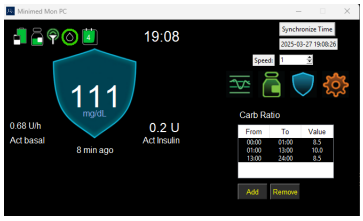


Figura 5.27: Pantalla de ratio de HC.

5.5.4.3. Factor de sensibilidad a la insulina

La pantalla del factor de sensibilidad a la insulina está ilustrada en la Figura 5.28, donde se puede ajustar la sensibilidad a la insulina en función de las necesidades del paciente.

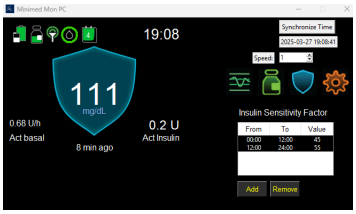


Figura 5.28: Pantalla de factor de sensibilidad a la insulina.

5.5.4.4. Efecto del reservorio de insulina

El efecto del reservorio de insulina se gestiona desde la pantalla mostrada en la Figura 5.29.

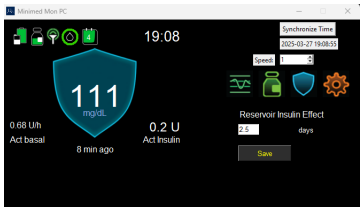


Figura 5.29: Pantalla de efecto del reservorio de insulina.

5.5.4.5. Basal máxima

Finalmente, la configuración de la basal máxima está representada en la Figura 5.30, donde el usuario puede ajustar el valor máximo de la tasa basal de insulina.

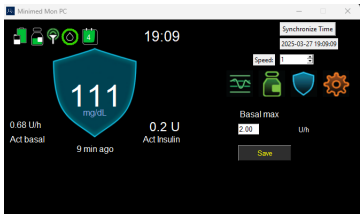


Figura 5.30: Pantalla de basal máxima.

5.6. Conclusiones

Este capítulo ha descrito el proceso completo de integración del algoritmo de la bomba de insulina en un entorno hardware, utilizando una Raspberry Pi como plataforma embebida. Se ha implementado una solución funcional que simula el comportamiento de una bomba comercial, permitiendo tanto el cálculo automático de insulina como la interacción con el usuario mediante una interfaz gráfica.

La Raspberry Pi ejecuta en tiempo real el algoritmo de control, conectado mediante una carpeta compartida al PC que actúa como generador de datos de glucosa. Esta arquitectura permite simular el entorno de un sistema híbrido de administración de insulina, donde los datos del sensor son procesados por la bomba para calcular las dosis necesarias.

Se ha desarrollado una interfaz intuitiva y funcional que permite al usuario visualizar la evolución de la glucosa, registrar comidas y eventos, y modificar parámetros clave del tratamiento. Esta integración proporciona un entorno de prueba robusto para validar el algoritmo en condiciones realistas y sienta las bases para futuras implementaciones en entornos clínicos o dispositivos médicos reales.

Capítulo 6

Obtención de datos reales

6.1. Introducción

La validación de un algoritmo de control automático de insulina requiere contrastar sus decisiones frente a datos reales de pacientes. Por ello, este capítulo se centra en la obtención, análisis y uso de datos reales descargados desde la plataforma CareLink de Medtronic. Estos datos incluyen valores de glucosa, dosis de insulina administradas (tanto bolos como basales automáticas), eventos registrados (comidas, correcciones, etc.) y parámetros del dispositivo.

Para integrar esta funcionalidad en el proyecto, se han desarrollado tres módulos principales dentro de la herramienta `DiabetesTool`:

- **Downloader**, que permite conectar con la API de CareLink y descargar automáticamente los datos del paciente.
- **Estimator**, encargado de ajustar los parámetros del modelo de simulación de glucosa a partir de los datos descargados, personalizando el comportamiento del modelo a cada paciente.
- **Comparator**, que ofrece herramientas de visualización para comparar la evolución de la glucosa simulada frente a la real, y para ejecutar simulaciones dinámicas basadas en datos actualizados.

A lo largo del capítulo se describen los siguientes aspectos:

- En la Sección 6.2, se presenta el funcionamiento de la API de CareLink y el sistema de arquitectura que permite la recogida automatizada de datos desde la nube de Medtronic.
- En la Sección 6.3, se detalla la estructura del archivo JSON descargado y se identifican los campos más relevantes para el algoritmo de la bomba.
- La Sección 6.4 actualiza el diagrama de flujo de datos, integrando el acceso a datos reales en el proceso de simulación y control.
- En la Sección 6.5, se explica el algoritmo de estimación de parámetros fisiológicos, basado en minimización del error cuadrático medio (RMS), y se detallan los parámetros ajustables.
- Por último, en la Sección 6.6, se describen las distintas visualizaciones generadas por el sistema, incluyendo la descarga de datos, la estimación de parámetros, la comparación de curvas de glucosa y la simulación dinámica en tiempo real con datos reales.

6.2. API de Carelink

Para automatizar la descarga de estos datos, se ha implementado una herramienta dentro de `DiabetesTool`, basada en el proyecto `carelink-python-client` desarrollado por `ondrej1024` [37]. Esta biblioteca en Python permite acceder a los datos almacenados en la nube de CareLink, incluyendo información sobre niveles de glucosa y administración de insulina.

6.2.1. Arquitectura del sistema

La arquitectura del sistema de recogida de datos se basa en la infraestructura en la nube proporcionada por Medtronic. La bomba de insulina transmite periódicamente los datos al teléfono del paciente mediante la aplicación Minimed Mobile, que actúa como puerta de enlace a la nube. Esta aplicación envía los datos a la plataforma CareLink Cloud. Posteriormente, una aplicación cliente, como `carelink-python-client`, puede acceder a estos datos a través de la API REST de CareLink.

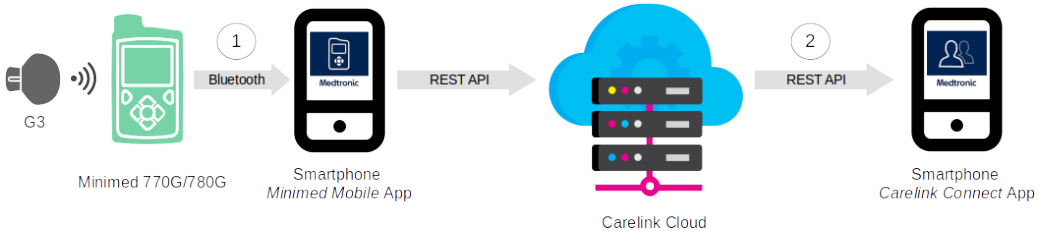


Figura 6.1: Arquitectura de comunicación de datos de CareLink.

6.2.2. Requisitos del sistema

Para el correcto funcionamiento de la herramienta de descarga de datos, se requiere:

- Una cuenta de paciente o de cuidador en CareLink, con las credenciales correspondientes.
- Python 3 instalado en el sistema.
- Las bibliotecas necesarias especificadas en el archivo `requirements.txt` del proyecto `carelink-python-client`.
- Un archivo `logindata.json` generado mediante el script de inicio de sesión proporcionado por `carelink-python-client`, que almacena los datos de autenticación necesarios para acceder a la API de CareLink.

6.3. Datos descargados

Al conectarse a la API de CareLink, se obtiene un archivo en formato `.json` que contiene un conjunto extenso de datos registrados por la bomba de insulina y el sensor continuo de glucosa

(CGM). Este archivo lleva por nombre el prefijo `data-` seguido de la fecha y hora de la descarga, por ejemplo: `data-20241024_190055.json`. Cada descarga contiene la información acumulada desde un día antes de la fecha de generación del archivo hasta el momento de la descarga.

6.3.1. Conexión con CareLink

La descarga de datos desde la plataforma CareLink se realiza mediante una conexión a la API oficial que utiliza la propia aplicación móvil *CareLink Connect* de Medtronic. Para ello, DiabetesTool emplea una biblioteca personalizada denominada `carelink_client2.py`, que actúa como cliente HTTP de la API y permite obtener los datos más recientes registrados por la bomba de insulina y el sensor CGM.

El acceso a la API requiere autenticación OAuth2, y se basa en un archivo de token llamado `logindata.json`. Este archivo contiene información sensible de autenticación, como el `access_token`, `refresh_token`, `client_id`, `client_secret`, y otros campos necesarios para establecer la conexión segura. Dicho archivo se genera previamente mediante el script `carelink_carepartner_api_login.py`.

Este segundo script emula el comportamiento de la aplicación CareLink Connect. Abre automáticamente una ventana del navegador Firefox y redirige al usuario a la página oficial de inicio de sesión de CareLink. Allí se introducen manualmente las credenciales del usuario (correo y contraseña). Una vez completado este proceso, se obtienen los datos de autenticación necesarios y se almacenan en el archivo `logindata.json`.

Posteriormente, el componente `Downloader` de DiabetesTool invoca la función `getRecentData()` definida en `carelink_client2.py`. Esta función valida la vigencia del `access_token`, y realiza una solicitud a la API con los encabezados adecuados y el identificador del paciente. La respuesta contiene un bloque de datos estructurados en formato JSON que incluye registros de glucosa, dosis de insulina, eventos alimenticios, alarmas y otros indicadores clínicos, los cuales son posteriormente almacenados y utilizados por el sistema.

6.3.2. Estructura y contenido del archivo JSON

El archivo JSON incluye múltiples bloques de información que describen el estado del sistema, niveles de glucosa registrados, dosis de insulina administradas, eventos de comida y otros indicadores relevantes. A continuación se resumen los campos más relevantes para este proyecto:

- `reservoirRemainingUnits`: cantidad de unidades de insulina restantes en el reservorio. Ejemplo:

```
1         "reservoirRemainingUnits": 81.7
2
```

Código 6.1: Estructura y contenido del archivo JSON (I).

- `medicalDeviceBatteryLevelPercent`: porcentaje de batería restante en la bomba de insulina.

```
1     "medicalDeviceBatteryLevelPercent": 75
2
```

Código 6.2: Estructura y contenido del archivo JSON (II).

- `sLastSensorTime`: fecha y hora de la última lectura del sensor.

```
1     "sLastSensorTime": "2024-10-24T18:57:01.000Z"
2
```

Código 6.3: Estructura y contenido del archivo JSON (III).

- `lastSG`: última medida de glucosa registrada por el sensor, incluyendo el valor y la marca temporal.

```
1     "lastSG": {
2         "sg": 127,
3         "datetime": "2024-10-24T18:57:01.000Z",
4         ...
5     }
6
```

Código 6.4: Estructura y contenido del archivo JSON (IV).

- `activeInsulin`: cantidad de insulina activa estimada en el cuerpo del paciente.

```
1     "activeInsulin": {
2         "amount": 1.8,
3         "datetime": "2024-10-24T18:57:10.929Z",
4         ...
5     }
6
```

Código 6.5: Estructura y contenido del archivo JSON (V).

- `sgs`: lista de valores de glucosa registrados cada 5 minutos aproximadamente. Cada entrada incluye el valor de glucosa, el instante de tiempo y su desfase temporal en segundos respecto al momento de descarga.

```
1     "sgs": [
2         {
3             "sg": 156,
4             "datetime": "2024-10-23T19:02:00.000Z",
5             "relativeOffset": -86110
6         },
7         {
8             "sg": 155,
9             "datetime": "2024-10-23T19:07:00.000Z",
10            ...
11        },
12        ...
```

```
    ]

```

Código 6.6: Estructura y contenido del archivo JSON (VI).

- `limits`: límites personalizados de glucosa para el usuario (hipoglucemia e hiperglucemia).

```
    "limits": [
        {
            "highLimit": 245,
            "lowLimit": 70,
            ...
        }
    ]

```

Código 6.7: Estructura y contenido del archivo JSON (VII).

- `markers`: lista de eventos registrados, entre ellos:
  - `"type": "MEAL"`: representa la ingesta de hidratos de carbono.
  - `"type": "INSULIN"`: bolus de insulina administrados, ya sea recomendados por el sistema (`RECOMMENDED`) o de corrección automática (`AUTOCORRECTION`).
  - `"type": "AUTO_BASAL_DELIVERY"`: dosis de insulina basal automatizada.

  Cada evento tiene su propia marca temporal (`dateTime`) y un `relativeOffset` en segundos desde el presente. Ejemplo de evento de bolus:

```
    {
        "type": "INSULIN",
        "dateTime": "2024-10-23T21:26:32.000-00:00",
        "deliveredFastAmount": 2.4,
        "bolusType": "FAST",
        "activationType": "RECOMMENDED"
    }

```

Código 6.8: Estructura y contenido del archivo JSON (VIII).

- `averageSG`: media de los valores de glucosa del periodo descargado.

```
    "averageSG": 130

```

Código 6.9: Estructura y contenido del archivo JSON (IX).

6.3.3. Datos utilizados por el algoritmo

Para el funcionamiento del algoritmo de bucle cerrado propuesto en este trabajo, solo se emplea una parte de la información contenida en el archivo. Concretamente, se utilizan los siguientes campos:

- `lastSG.sg` y `lastSG.datetime`: último valor de glucosa disponible y su timestamp.
- `sLastSensorTime`: para sincronizar temporalmente el sistema.
- `sgs[i].sg` y `sgs[i].datetime`: valores históricos de glucosa, empleados en el análisis de tendencia y validación del modelo.
- `activeInsulin.amount` y `activeInsulin.datetime`: cantidad de insulina activa que debe considerarse en la toma de decisiones.

El resto de campos pueden emplearse para análisis secundarios, supervisión del estado del dispositivo o cálculos estadísticos adicionales, pero no se integran directamente en el proceso de decisión automática del algoritmo de insulina.

6.4. Diagrama de flujo de datos

El diagrama de flujo de datos es ahora el mostrado en la siguiente figura.

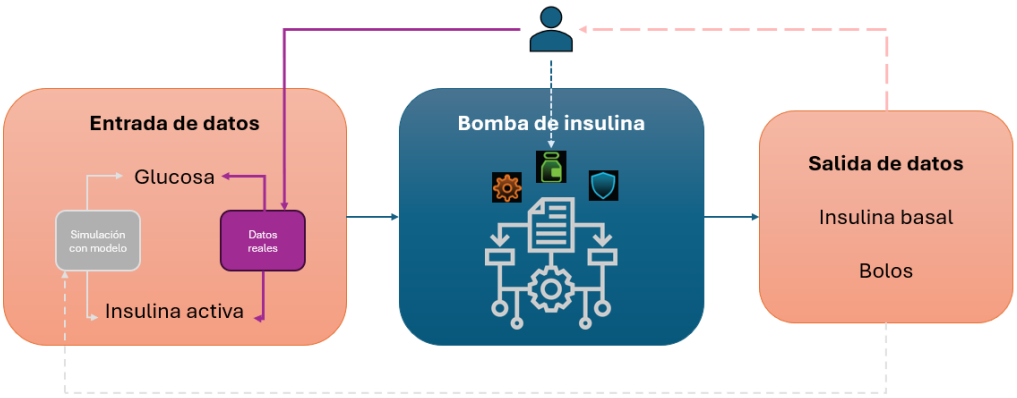


Figura 6.2: Diagrama de flujo de datos con datos reales.

6.5. Estimador de parámetros

Dentro de la herramienta `DiabetesTool` se ha implementado un módulo específico para la estimación de parámetros del modelo de simulación de glucosa. Gracias a la descarga de datos reales desde la nube de Medtronic (descritos en la sección anterior), es posible ajustar de forma personalizada los valores de ciertos parámetros fisiológicos del simulador, con el objetivo de mejorar la precisión de la simulación en relación a la evolución real de la glucosa en el paciente.

Este proceso de ajuste se basa en comparar la curva de glucosa generada por el modelo de simulación con los valores de glucosa reales medidos por el sensor. Ambos se comparan bajo los mismos inputs de referencia: las ingestas de carbohidratos introducidas manualmente por el paciente y las dosis de insulina administradas automáticamente por la bomba de insulina.

6.5.1. Parámetros ajustables

El simulador incorpora una serie de parámetros fisiológicos cuya estimación permite personalizar el comportamiento del modelo a las características específicas del paciente. Algunos de los parámetros clave que pueden ser ajustados son los siguientes:

- V_G: Volumen de distribución de la glucosa.
- SIT: Sensibilidad a la insulina en el transporte de glucosa.
- SID: Sensibilidad a la insulina en la eliminación de glucosa.
- SIE: Sensibilidad a la insulina en la producción endógena de glucosa.
- EGP_0: Tasa basal de producción de glucosa endógena.

Estos parámetros están definidos en la Sección 3.4.1.

6.5.2. Algoritmo de estimación basado en RMS

La estimación se realiza mediante un algoritmo iterativo que busca minimizar el error cuadrático medio (Root Mean Square, RMS) entre los valores simulados y los valores reales de glucosa. El RMS es una métrica estadística que cuantifica la diferencia promedio al cuadrado entre dos conjuntos de datos, y se utiliza ampliamente en contextos de ajuste de modelos [38].

En cada iteración, se realiza una simulación del modelo variando ligeramente el parámetro a estimar, y se calcula la diferencia entre los niveles de glucosa simulados y los valores reales del sensor. La simulación se recorta al intervalo de tiempo definido por el usuario, y los residuos se calculan como la diferencia puntual entre las dos curvas de glucosa. El algoritmo selecciona el valor del parámetro que minimiza el RMS de estos residuos.

Este enfoque permite identificar el valor óptimo del parámetro que mejor reproduce la evolución de la glucosa observada en la realidad, utilizando únicamente datos reales como referencia. Se trata, por tanto, de un proceso de ajuste personalizado que adapta el modelo teórico al comportamiento fisiológico individual del paciente.

6.5.3. Resultados del estimador

Una vez completadas las iteraciones, el algoritmo devuelve:

- El valor óptimo estimado del parámetro.
- La serie de residuos obtenida durante el mejor ajuste.
- La curva de glucosa simulada más precisa.

6.6. Visualización

6.6.1. Descarga de datos

Durante el proceso de descarga, los datos obtenidos desde la nube de Medtronic se almacenan en un archivo JSON denominado `data.json`. En la Figura 6.3 se muestran las trazas correspondientes a esta operación en la terminal. Además, se genera automáticamente una versión simplificada del archivo, llamada `simpledata.json`, que contiene únicamente los datos esenciales para la simulación. Ambos archivos se almacenan en la misma ruta.

En caso de no haber seleccionado la opción "Save data", sólo se generaría el archivo `simpledata.json`. Como parte del proceso, también se genera una visualización de la evolución de la glucosa en las últimas 24 horas, que corresponde al intervalo de tiempo descargado desde la nube.

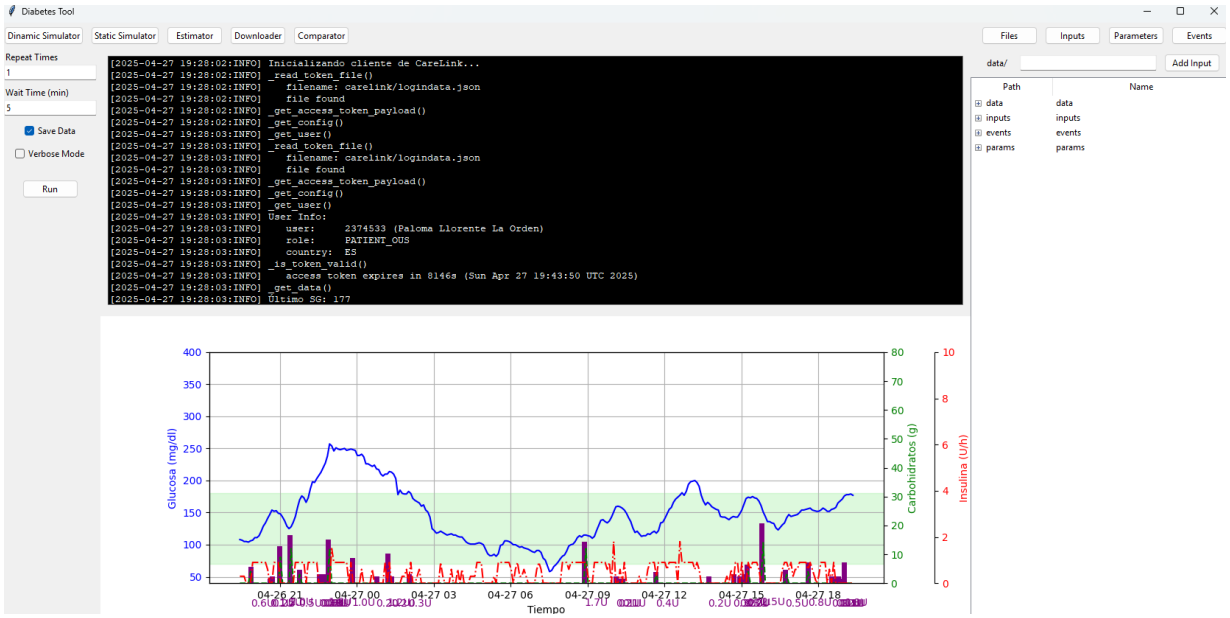


Figura 6.3: Trazas de la terminal durante la descarga de datos y visualización de las últimas 24h de glucosa.

6.6.2. Estimador de parámetros

El estimador de parámetros permite ajustar los valores del modelo de glucosa en una franja temporal determinada. Cada estimación está enfocada en un único parámetro, el cual se optimiza dentro de un rango definido por el usuario (valores mínimos y máximos), con un número determinado de iteraciones para minimizar el error cuadrático medio (RMS).

Cuando se realizan varias estimaciones consecutivas, el modelo puede ajustarse de manera más precisa al comportamiento real del paciente, ya que cada iteración reduce progresivamente la diferencia entre la curva simulada y los datos reales de glucosa.

En la Figura 6.4 se muestran los residuos obtenidos tras la estimación, mientras que la Figura 6.5 ilustra la comparación entre la curva de glucosa simulada y la curva real.

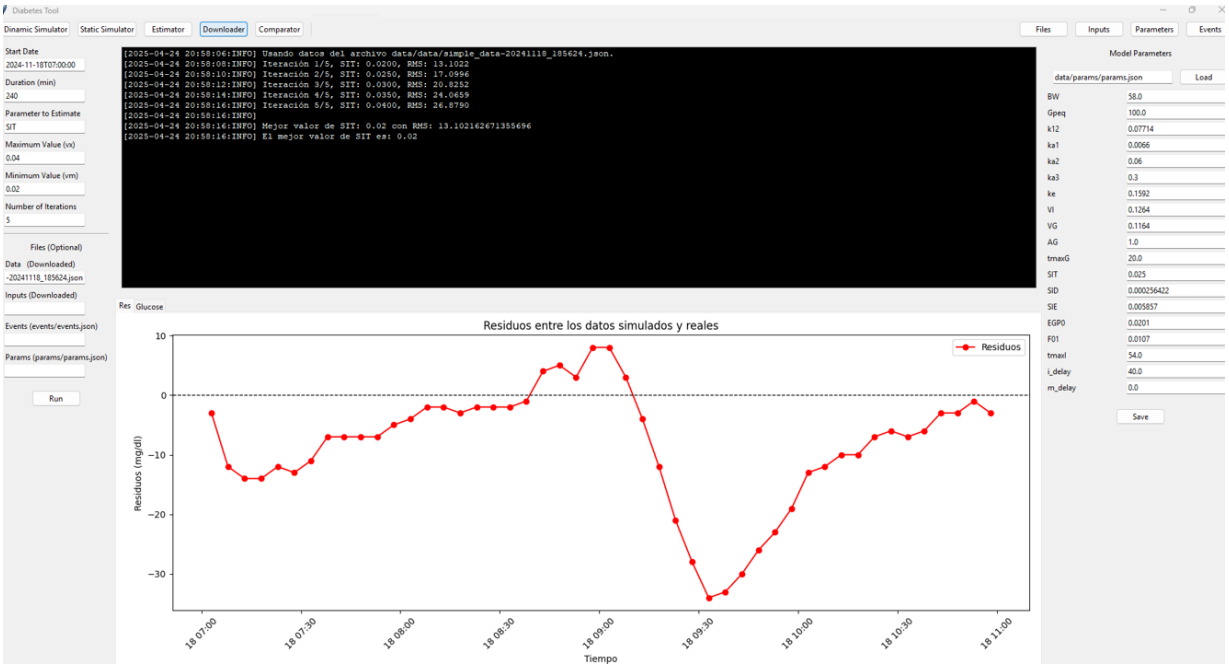


Figura 6.4: Estimación de parámetros: residuos.

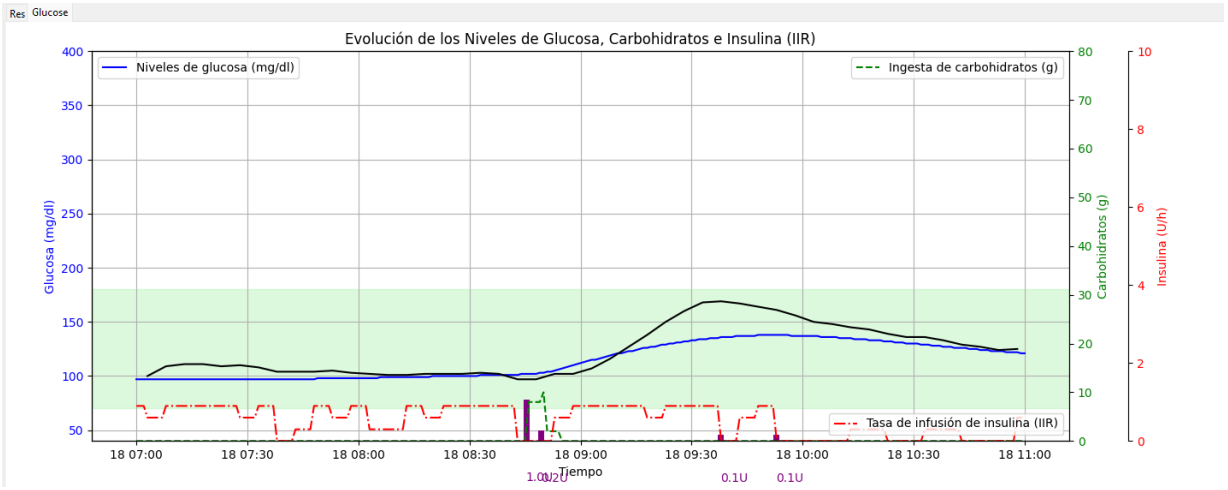


Figura 6.5: Comparación entre la curva de glucosa simulada y la real tras la estimación.

6.6.3. Comparador de modelo y datos reales

Esta funcionalidad permite comparar directamente la curva de glucosa simulada por el modelo con los datos reales registrados por la bomba de insulina. Los datos reales pueden ser los recién descargados o también pueden provenir de un archivo histórico. Esta comparación es útil para evaluar la precisión del modelo frente a situaciones reales.

La Figura 6.6 muestra un ejemplo de esta comparación.

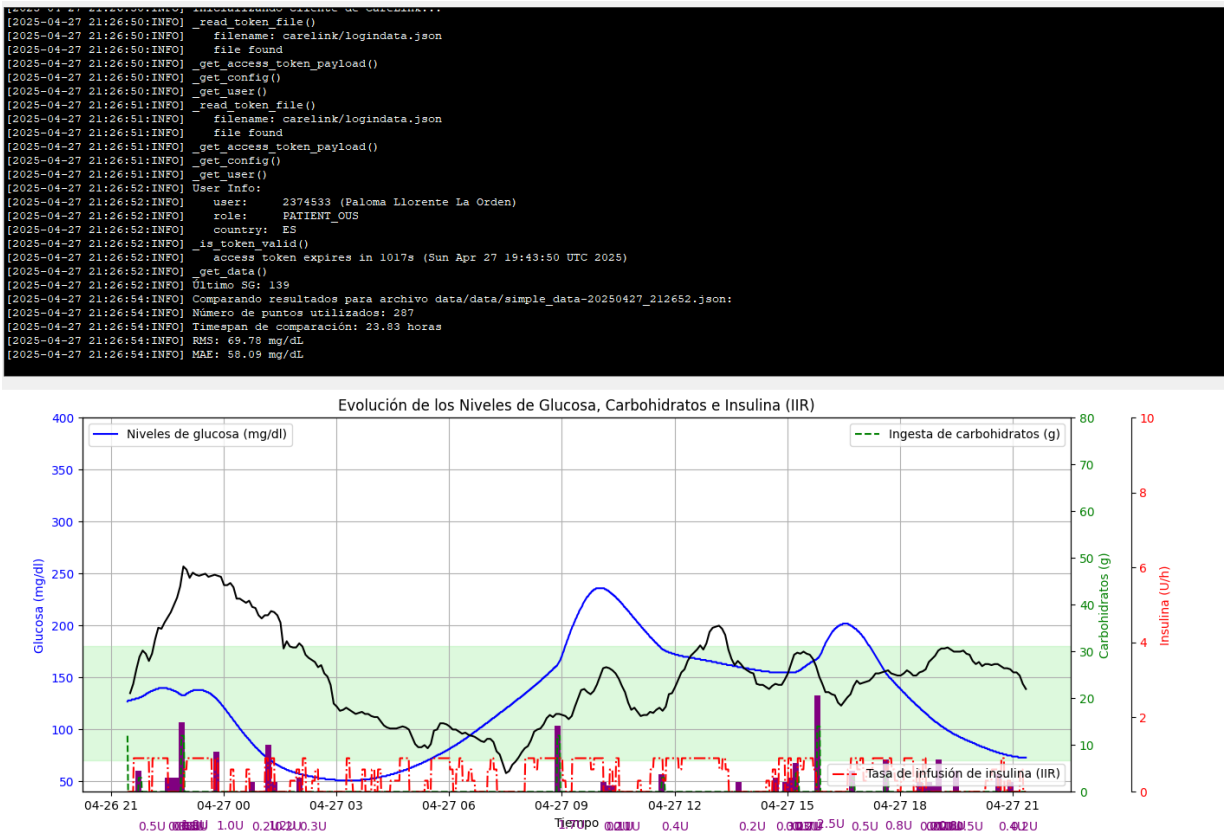


Figura 6.6: Comparación entre la curva del modelo y los datos reales de glucosa.

6.6.4. Simulación dinámica con datos reales

Esta parte de la herramienta de DiabetesTool permite simular la evolución de la glucosa en tiempo real, utilizando los datos reales obtenidos de la bomba de insulina. En un primer momento, se descargan los datos de las últimas 24 horas y se parte de ellos. En función de la fecha inicial de la simulación, se genera el plot desde el primer dato de la descarga hasta la fecha inicial de la simulación (aunque se trunca el plot a 12h para observar bien el detalle de la gráfica). Este plot incluye tanto la evolución de la glucosa, como las comidas y las dosis de insulina administradas reales. A continuación, se va descargando cada 5 minutos nuevos datos de la bomba de insulina (mediante la nube de Carelink como se ha comentado en la sección). Estos nuevos datos de glucosa se van añadiendo al plot, y se va actualizando la curva de glucosa simulada. Sin embargo, ahora no se incluyen los datos de comidas e insulina reales.

Además, se va actualizando un fichero de output con formato `data.json`, que es el que utilizaría el algoritmo de la bomba de insulina para tomar decisiones en la administración de insulina. Para validar este algoritmo de la bomba de insulina, es importante que los outputs generados por el algortimo de la bomba de insulina sean similares a los que genera la bomba real.

La Figura 6.7 muestra el entorno visual de esta simulación dinámica en funcionamiento.

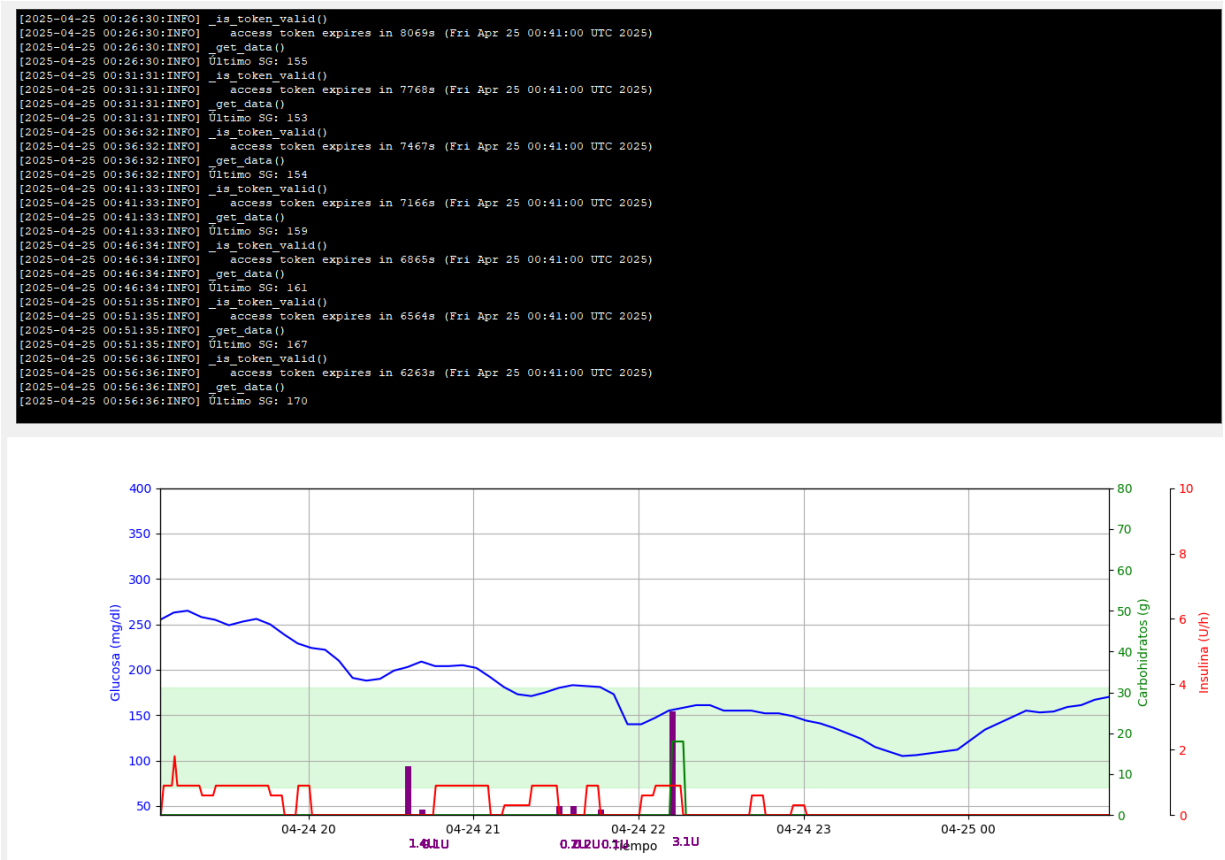


Figura 6.7: Simulación dinámica basada en datos reales de la bomba de insulina.

6.7. Conclusiones

En este capítulo se ha descrito el proceso completo de adquisición y utilización de datos reales de pacientes diabéticos tipo 1, como parte esencial en la validación del algoritmo de la bomba de insulina. Gracias a la integración con la API de CareLink, es posible automatizar la descarga de lecturas de glucosa e historiales de insulina administrada por una bomba real.

El uso de estos datos ha sido clave en varios frentes. Por un lado, permite alimentar directamente al algoritmo de control con datos actualizados, simulando un entorno realista. Por otro, la herramienta `Estimator` permite ajustar el modelo de simulación a cada paciente concreto, mejorando así la precisión del sistema. Finalmente, mediante el módulo `Comparator`, se ha validado el comportamiento del sistema frente a escenarios reales, incluyendo comparaciones gráficas entre la curva simulada y la curva real, y simulaciones dinámicas que permiten replicar situaciones reales en tiempo real.

El conjunto de módulos desarrollados en este capítulo amplía considerablemente la utilidad de `DiabetesTool`, al convertirla en una plataforma capaz no sólo de simular y predecir el comportamiento de la glucosa, sino también de ajustarse y compararse frente a la realidad clínica. Esto refuerza la base para el desarrollo de sistemas más avanzados de control automatizado de insulina personalizados y robustos.

Capítulo 7

Validación

7.1. Introducción

Este capítulo presenta el proceso de validación de los dos principales componentes desarrollados en este trabajo: el modelo de simulación de glucosa en pacientes con diabetes tipo 1 y el algoritmo de la bomba de insulina. La validación se ha realizado empleando datos reales extraídos desde la plataforma CareLink de Medtronic, previamente procesados mediante los módulos de la herramienta `DiabetesTool` descritos en el Capítulo 6.

En primer lugar, en la Sección 7.2, se validan las capacidades del modelo de simulación de glucosa para reproducir dinámicas fisiológicas realistas bajo distintos escenarios, como comidas estándar, ingestas con alto contenido en grasas y proteínas (UGPs), y ejercicio físico. Esta validación se basa en comparaciones cuantitativas y visuales entre la curva simulada y los datos reales del paciente. Para ajustar el modelo a la respuesta específica de cada escenario, se utilizan herramientas como el módulo `Estimator`, que permite estimar los parámetros fisiológicos más representativos, y el módulo `Comparator`, que proporciona métricas e imágenes comparativas.

A continuación, en la Sección 7.3, se valida el comportamiento del algoritmo de la bomba de insulina, enfrentándolo a datos reales de glucosa obtenidos en distintos contextos del día a día. El objetivo en este caso es comprobar si el algoritmo implementado es capaz de tomar decisiones adecuadas, similares a las que tomaría una bomba comercial, como la administración de bolo por comidas o correcciones, así como la liberación de insulina basal.

Finalmente, en la Sección 7.4, se analiza de forma explícita cómo los resultados de validación permiten afirmar que se han cumplido los requisitos funcionales y no funcionales definidos en el Capítulo 3.

7.2. Modelo de diabetes

El objetivo es validar que el modelo de simulación de glucosa refleja de manera realista el comportamiento fisiológico en pacientes con diabetes tipo 1, tanto en condiciones normales como en eventos especiales.

7.2.1. Metodología

- Comparación de resultados usando el Comparator en diferentes escenarios.

- Ajuste de parámetros usando el Estimator para minimizar el error entre modelo y datos de referencia.
- Evaluación basada en indicadores cuantitativos y análisis visual. Para los indicadores se consideran:
  - RMS: Raíz del error cuadrático medio.
  - MAE: Error absoluto medio.
  - Max Abs Error: Máximo error absoluto.
  - Bias: media de los errores.

7.2.2. Escenarios de validación

La validación se realiza sobre varios escenarios representativos. Para cada uno se presentan las comparaciones gráficas, los resultados numéricos y una breve interpretación.

7.2.2.1. Comida estándar

Descripción: Este escenario corresponde a una jornada alimentaria típica compuesta por tres ingestas consecutivas: desayuno, almuerzo y comida. Para el desayuno se toma un café con leche semidesnatada; en el almuerzo se consume una manzana, y durante la comida principal se ingieren guisantes y solomillo de cerdo, finalizando con un capuchino con leche como postre. La combinación de comidas, junto con su distribución temporal, genera una dinámica glucémica compleja que constituye un buen caso para evaluar la capacidad del modelo de adaptarse mediante estimación de parámetros.

Comparación de resultados inicial: En una primera simulación sin realizar ajustes en los parámetros, se observó un desfase temporal significativo entre el momento de la ingesta y el incremento glucémico en sangre. Tal como se muestra en la Figura 7.3, el modelo inicial subestima la velocidad con la que aparece el pico de glucosa tras la comida. La Figura 7.1 recoge los parámetros iniciales utilizados, y la Figura 7.2 los inputs definidos para la herramienta comparadora.

Los indicadores obtenidos para esta comparación inicial fueron:

- RMS: 19.42 mg/dL
- MAE: 16.36 mg/dL
- Error absoluto máximo: 41.0 mg/dL
- Bias: -9.98 mg/dL

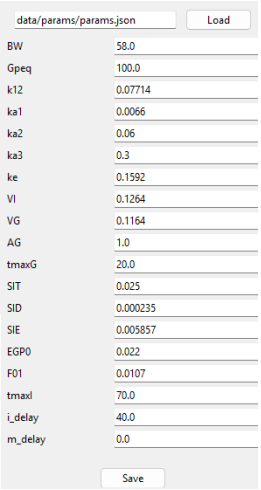


Figura 7.1: Parámetros iniciales para la comida estándar.

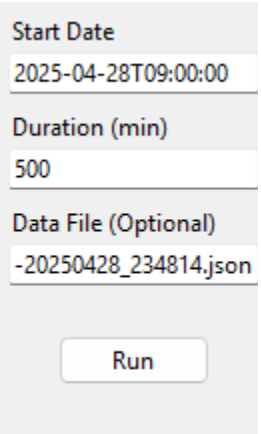


Figura 7.2: Inputs comparador inicial para la comida estándar.

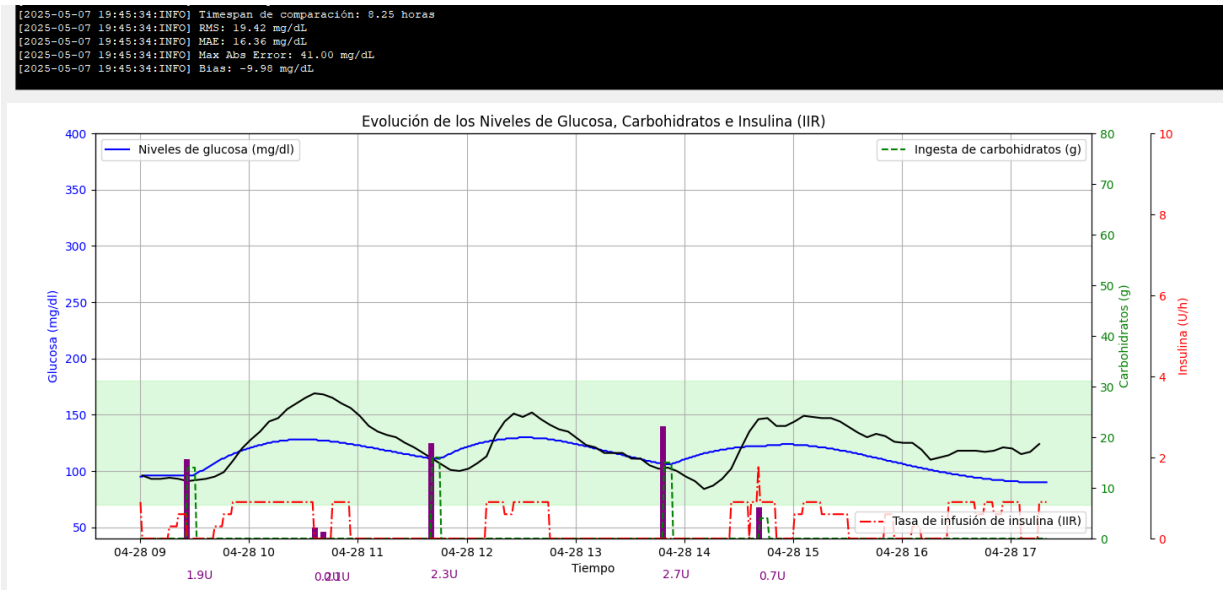
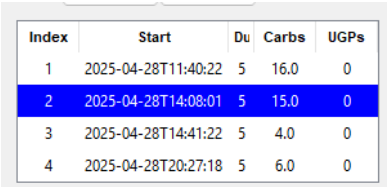


Figura 7.3: Plots de la comparación inicial para la comida estándar.

Estimación de parámetros: Dada la lentitud en la aparición del pico postprandial, se realizó una primera estimación del parámetro `m_delay`, que representa el retardo entre la ingesta y su efecto sobre la glucemia. Esto se justifica tanto por la naturaleza de los alimentos consumidos como por el hecho de que se pospuso ligeramente la comida principal, introduciendo así un retardo efectivo adicional. Se modificó manualmente el tiempo de ingesta unos 20 minutos para reflejar este comportamiento (Figura 7.4), y se procedió a estimar `m_delay` usando la herramienta correspondiente (Figura 7.5).

Los resultados de esta estimación se muestran en las Figuras 7.6 y 7.7, que ilustran la mejora del ajuste en la evolución de la glucosa y la disminución de residuos, respectivamente. Los nuevos valores del parámetro pueden verse en la Figura 7.8.



Index	Start	Du	Carbs	UGPs
1	2025-04-28T11:40:22	5	16.0	0
2	2025-04-28T14:08:01	5	15.0	0
3	2025-04-28T14:41:22	5	4.0	0
4	2025-04-28T20:27:18	5	6.0	0

Figura 7.4: Modificación tiempo de comida para la comida estándar.

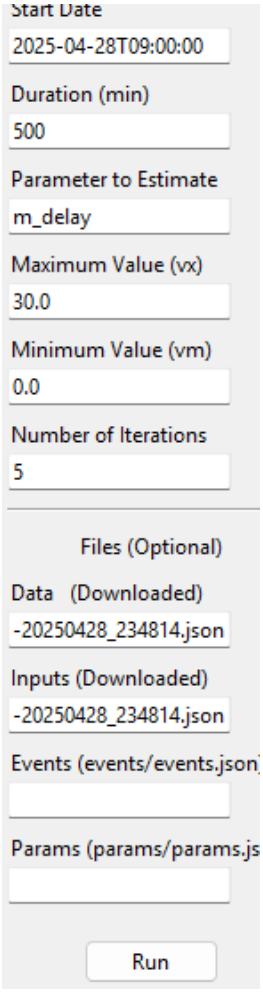


Figura 7.5: Inputs estimador para la comida estándar: m_delay.

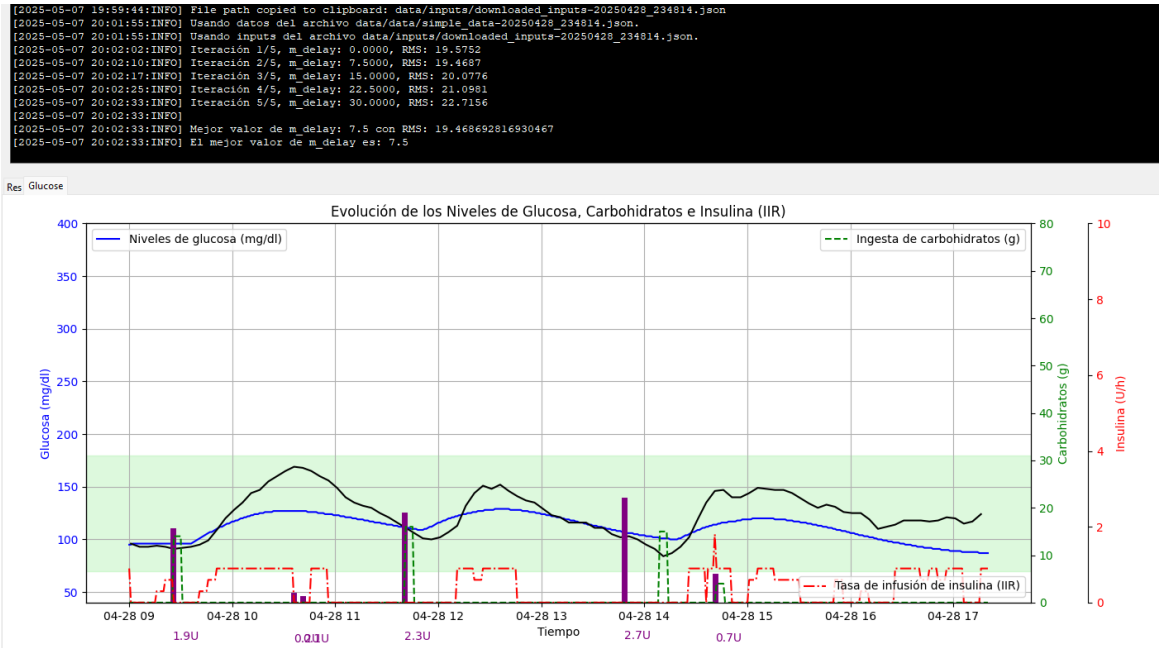


Figura 7.6: Estimación m_delay para comida estándar: evolución de glucosa.

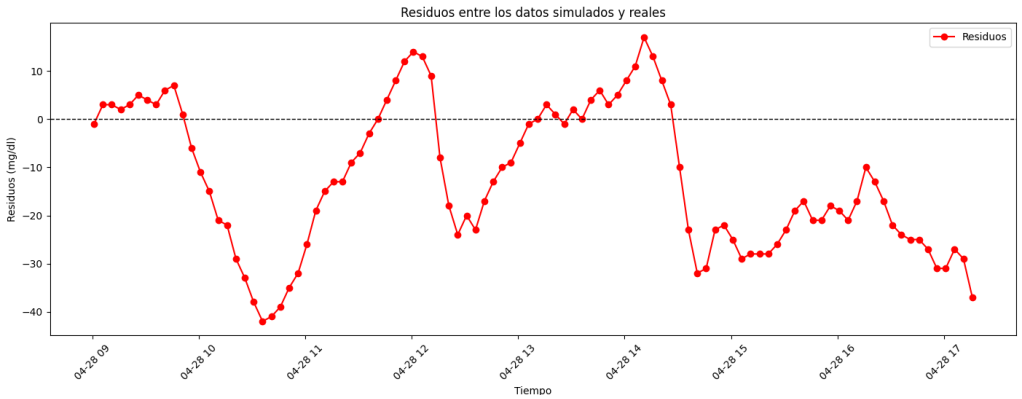


Figura 7.7: Estimación m_delay para comida estándar: evolución de residuos.

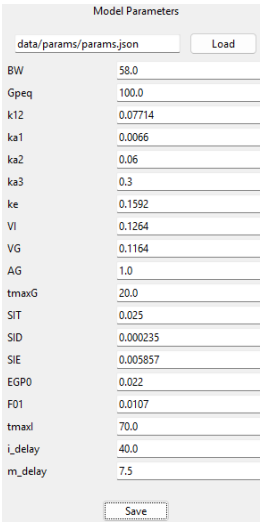


Figura 7.8: Parámetros para la comida estándar: modificación de m_delay.

Para ajustar el modelo a los datos reales observados tras la ingesta de comida, se detecta que la glucosa simulada no alcanza un pico suficientemente alto en las primeras dos horas, a pesar de converger correctamente a largo plazo. Entre los parámetros disponibles (SIT, SID, SIE y EGP0), se considera que `SIT` (Insulin Sensitivity of Glucose Distribution/Transport) es el más adecuado para ser ajustado. Como se detalla en el artículo de Hovorka et al. (2002), esto se debe a que un valor elevado de SIT implica una captación rápida de glucosa desde el compartimento plasmático hacia los tejidos, lo que puede atenuar de forma prematura el pico postprandial. Por tanto, reducir el valor de SIT permite que la glucosa permanezca más tiempo en sangre, lo cual aumenta la altura del pico de glucosa tras la ingesta y mejora el ajuste del modelo en la fase inicial posterior a la comida. La Figura 7.9 muestra la evolución de la glucosa con el nuevo valor de `SIT`, y la Figura 7.10 su impacto en los residuos.

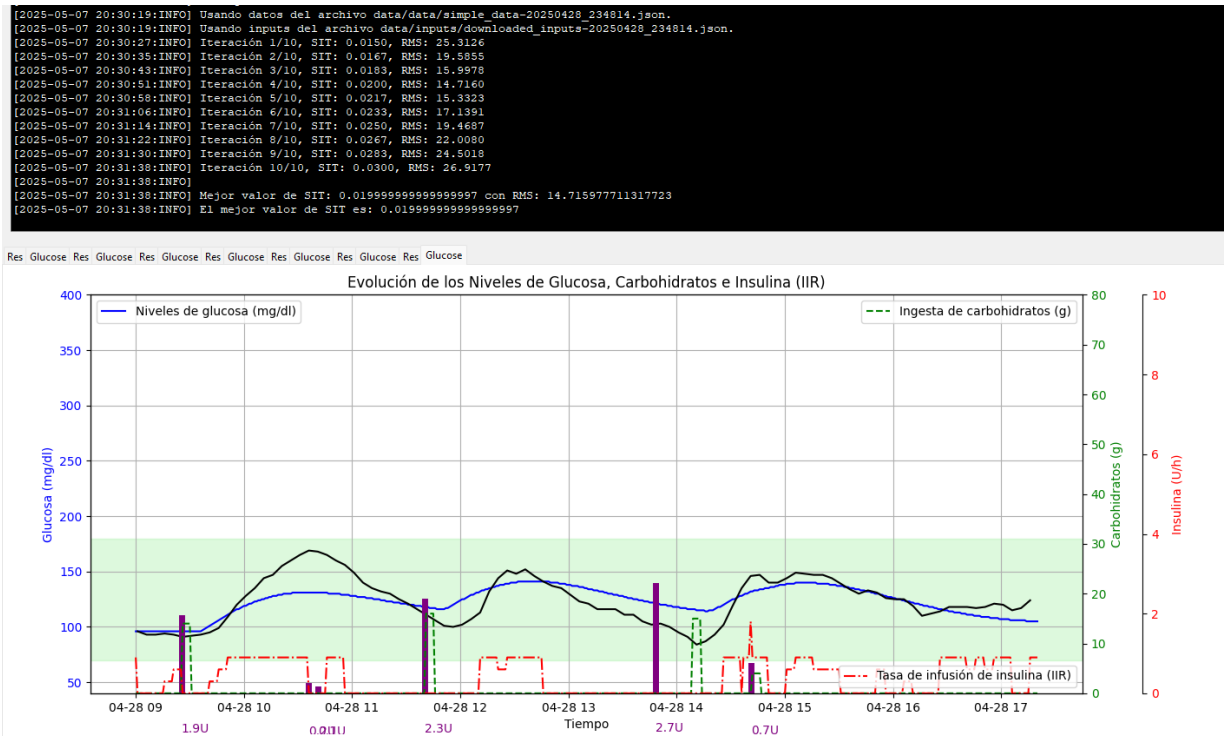


Figura 7.9: Estimación SIT para comida estándar: evolución de glucosa.

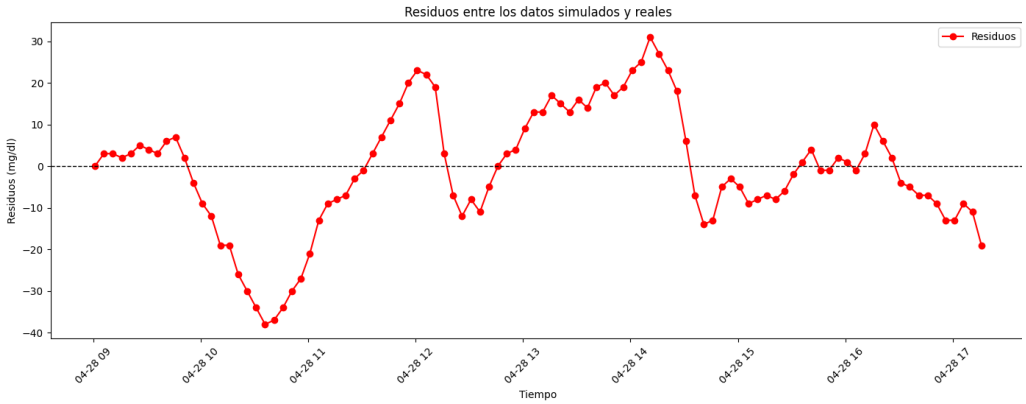


Figura 7.10: Estimación SIT para comida estándar: evolución de residuos.

Comparación de resultados final: Una vez ajustados los parámetros `m_delay` y `SIT`, se repite la simulación completa. La Figura 7.12 muestra el resultado de esta segunda comparación, y la Figura 7.11 recoge los parámetros finales utilizados. La mejora en la calidad del ajuste es evidente visualmente y también cuantitativamente, como muestran los indicadores.

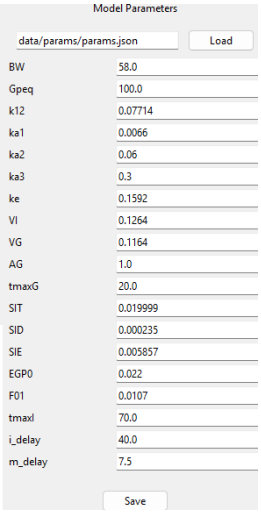


Figura 7.11: Parámetros para la comida estándar: modificación de SIT.

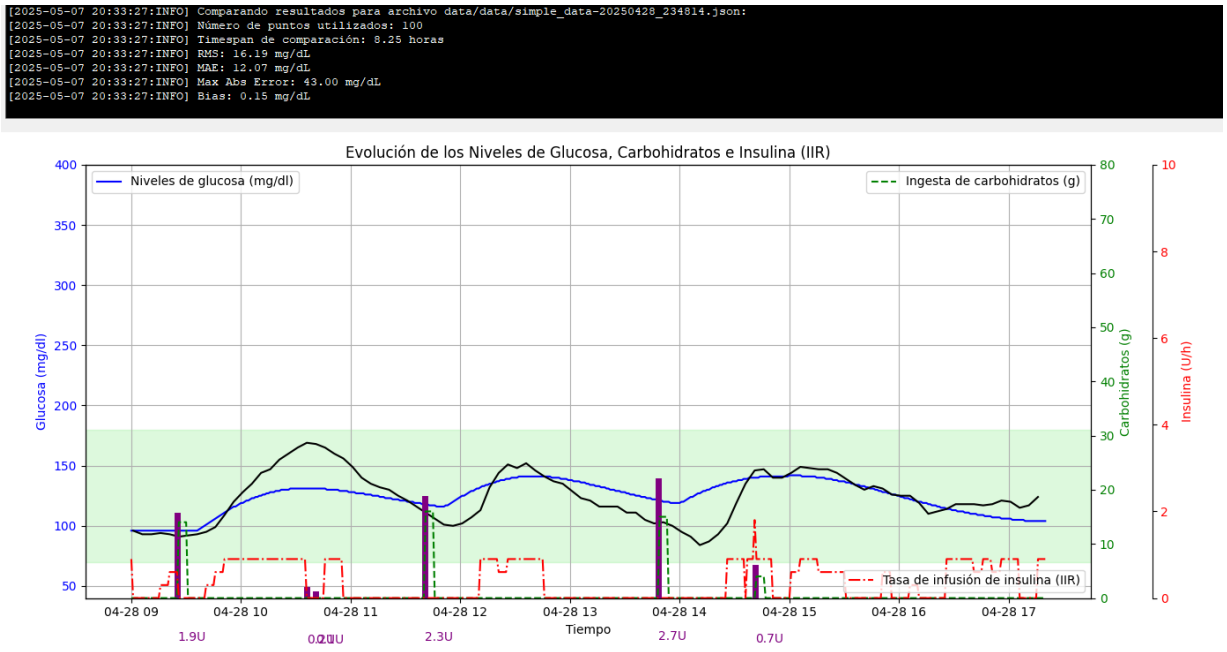


Figura 7.12: Plots de la comparación final para la comida estándar.

Indicadores de evaluación:

- RMS: 16.19 mg/dL
- MAE: 12.07 mg/dL
- Error absoluto máximo: 43.0 mg/dL
- Bias: 0.15 mg/dL

La reducción del RMS y MAE respecto a la comparación inicial indica una mejora en el ajuste general del modelo. Además, el cambio en el bias de -9.98 mg/dL a 0.15 mg/dL muestra que la desviación sistemática del modelo ha sido prácticamente eliminada, reforzando la validez del proceso de estimación.

Interpretación: Este caso demuestra cómo pequeñas modificaciones en los parámetros `m_delay` y `SIT`, apoyadas en la interpretación fisiológica del comportamiento glucémico, permiten adaptar el modelo a datos reales de forma efectiva. La modificación del `m_delay` permitió reflejar un retardo en la absorción de la glucosa debido tanto a la composición de los alimentos como al retraso en la ingesta. Por su parte, el ajuste de `SIT` permitió elevar el pico glucémico, ajustándose mejor a la respuesta real observada. Ambos ajustes contribuyen de forma sinérgica a la mejora sustancial de la precisión del modelo.

7.2.2.2. Comida alta UGP

Descripción:

Este escenario corresponde a una comida realizada en un restaurante, caracterizada por una larga duración y por un alto contenido en alimentos con alto UGP. A pesar de no realizar más ingestas durante las horas posteriores, la glucosa experimenta un incremento sostenido varias horas después de haber finalizado la comida. Por la noche, únicamente se toma una chocolatina, sin intervención adicional relevante. Este tipo de patrón postprandial es especialmente complejo de modelar, ya que el aumento glucémico diferido no se corresponde con una ingesta inmediata y puede inducir a error en los algoritmos de predicción y control de las bombas actuales.

Comparación de resultados:

En una primera comparación se emplean los inputs tal como fueron registrados por la bomba real. Como se observa en la Figura 7.13, al no tener en cuenta el alto contenido en UGP, el modelo simula una glucosa que tiende a descender de manera continua. Esto contrasta con la evolución real, en la que la glucosa sigue aumentando varias horas tras la ingesta debido a la absorción prolongada de las UGPs. En la práctica, la bomba real introduce varias correcciones a lo largo de la tarde para contrarrestar ese aumento no previsto, lo cual complica aún más el ajuste del modelo.

Estimación de parámetros:

Para poder simular adecuadamente este escenario y realizar una estimación fiable del parámetro `SIT`, fue necesario modificar los inputs originales, ajustándolos a la realidad del comportamiento observado. En primer lugar, se corrigieron las horas de inicio de las comidas (en realidad es una comida dividida en 3 partes ya que iba inyectando insulina a medida que comía), ya que en el restaurante se comenzó a comer más tarde de lo que se indicó inicialmente en la bomba, por lo que se introdujo un retardo manual. Además, se añadieron dos unidades de UGP en el primer bolo, y otras tres en el segundo, que fue administrado con cierto desfase y con una mayor cantidad de hidratos de carbono (HC) estimada, ya que inicialmente no se había cuantificado correctamente. La Figura 7.14 muestra los inputs modificados tras este ajuste.

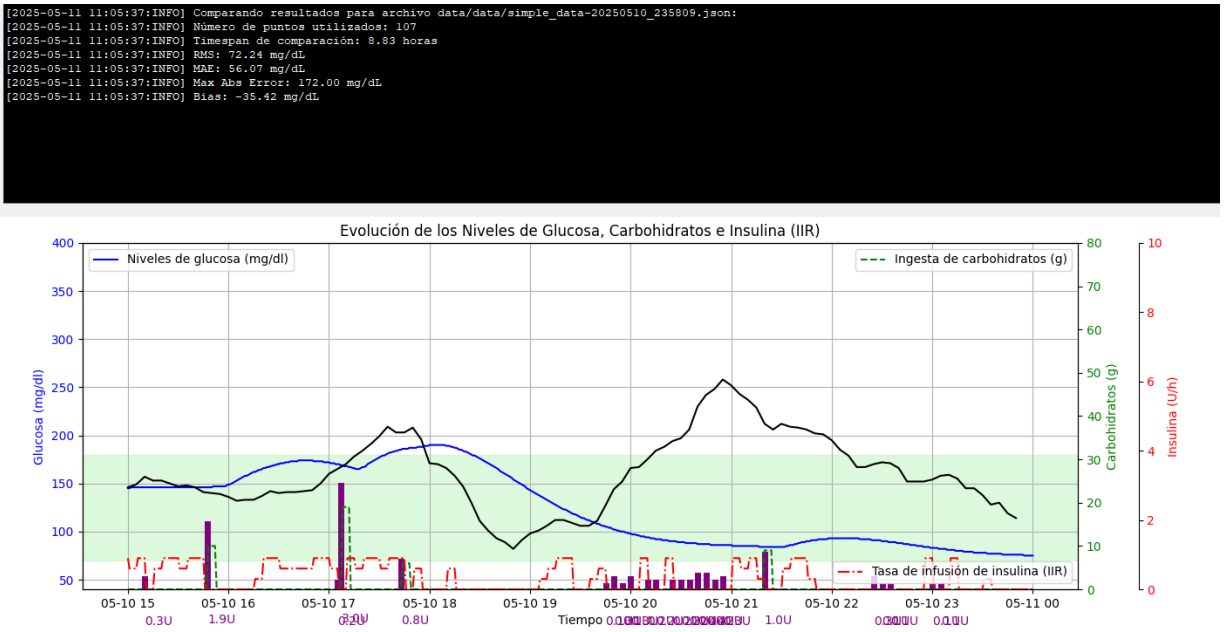


Figura 7.13: Plots de la comparación para la comida alta UGP.

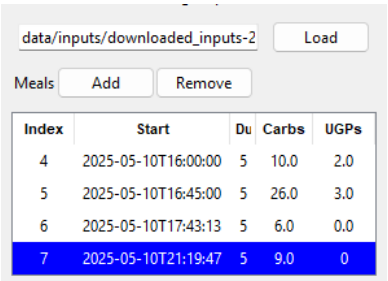


Figura 7.14: Modificación inputs para la comida alta UGP.

Una vez introducidos los nuevos valores, se ejecuta la herramienta de estimación del parámetro `SIT`. Los resultados obtenidos se muestran en las Figuras 7.15 (parámetros estimados), 7.16 (evolución de la glucosa) y 7.17 (residuos).

```
[2025-05-11 11:33:09:INFO] Usando datos del archivo data/data/simple_data-20250510_235809.json.
[2025-05-11 11:33:09:INFO] Usando inputs del archivo data/inputs/downloaded_inputs-20250510_235809.json.
[2025-05-11 11:33:19:INFO] Iteración 1/5, SIT: 0.0150, RMS: 108.2791
[2025-05-11 11:33:34:INFO] Iteración 2/5, SIT: 0.0212, RMS: 59.9213
[2025-05-11 11:33:49:INFO] Iteración 3/5, SIT: 0.0275, RMS: 39.7352
[2025-05-11 11:34:04:INFO] Iteración 4/5, SIT: 0.0338, RMS: 37.8127
[2025-05-11 11:34:20:INFO] Iteración 5/5, SIT: 0.0400, RMS: 43.8820
[2025-05-11 11:34:20:INFO]
[2025-05-11 11:34:20:INFO] Mejor valor de SIT: 0.03375 con RMS: 37.81274571249431
[2025-05-11 11:34:20:INFO] El mejor valor de SIT es: 0.03375
[2025-05-11 11:34:20:INFO] Comparando resultados para archivo data/data/simple_data-20250510_235809.json:
[2025-05-11 11:34:20:INFO] Número de puntos utilizados: 107
[2025-05-11 11:34:20:INFO] Timespan de comparación: 8.83 horas
[2025-05-11 11:34:20:INFO] RMS: 37.81 mg/dL
[2025-05-11 11:34:20:INFO] MAE: 27.07 mg/dL
[2025-05-11 11:34:20:INFO] Max Abs Error: 104.00 mg/dL
[2025-05-11 11:34:20:INFO] Bias: -2.08 mg/dL
```

Figura 7.15: Resultados estimación SIT para la comida alta UGP.

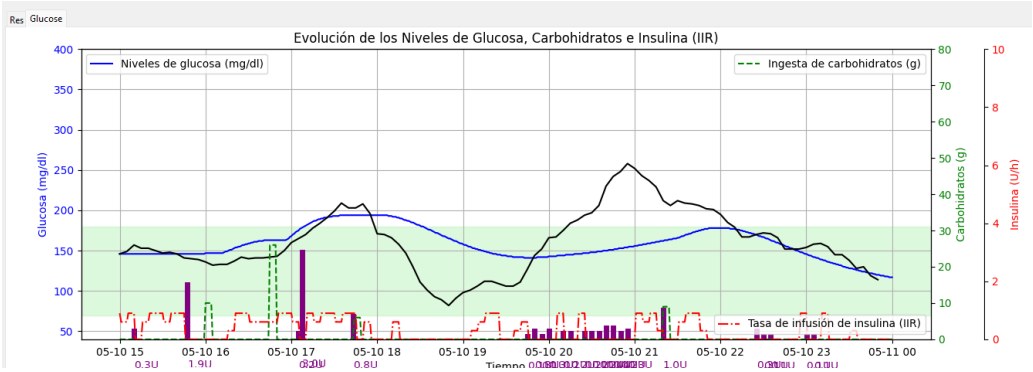


Figura 7.16: Evolución de glucosa para la comida alta UGP.

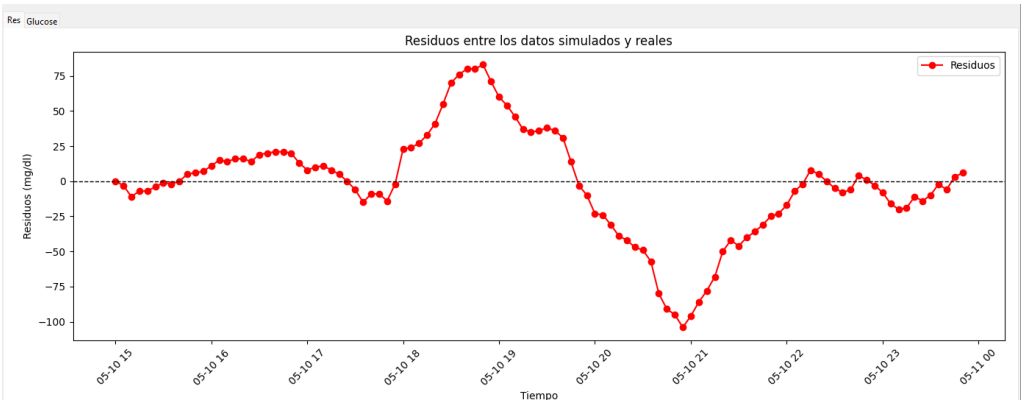


Figura 7.17: Residuos para la comida alta UGP.

Indicadores de evaluación:

- RMS: 37.81 mg/dL
- MAE: 27.07 mg/dL
- Error absoluto máximo: 104.0 mg/dL
- Bias: -2.08 mg/dL

Interpretación:

Este escenario pone de manifiesto la dificultad de modelar comidas con alto contenido en UGP, especialmente cuando la absorción de glucosa se produce de forma muy diferida en el tiempo. El modelo inicial, basado únicamente en los inputs reales sin modificación, no era capaz de capturar esta dinámica, por lo que fue necesario introducir cambios tanto en los tiempos como en las características de los bolo de comida (aumentando HC y UGP).

Aunque los errores absolutos son elevados, especialmente el máximo, el valor del bias es bajo (-2.08 mg/dL), lo que indica que el modelo no presenta una desviación sistemática significativa.

7.2.2.3. Ejercicio físico

Descripción:

Este escenario corresponde a un día en el que se realiza actividad física moderada en el gimnasio. El ejercicio principal consiste en una sesión de aproximadamente 60 minutos de intensidad media (valor 5), a lo que se considera añadido el efecto de los desplazamientos de ida y vuelta caminando (unos 10 minutos cada uno), que también influyen de manera significativa en el descenso progresivo de la glucosa plasmática. Este tipo de actividad física genera una respuesta glucémica compleja, caracterizada por una mayor sensibilidad a la insulina y un aumento del consumo de glucosa por parte de los tejidos, lo cual debe ser adecuadamente representado por el modelo.

Comparación de resultados:

Se introduce un evento de ejercicio físico de 60 minutos con intensidad 5 en el fichero de eventos, como se muestra en la Figura 7.18. El modelo, con los parámetros por defecto, ya reproduce de manera razonablemente adecuada el descenso de la glucosa inducido por el ejercicio, tal y como se observa en la Figura 7.19.

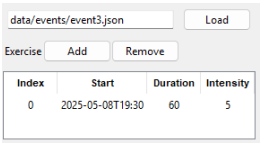


Figura 7.18: Evento asociado al ejercicio físico.

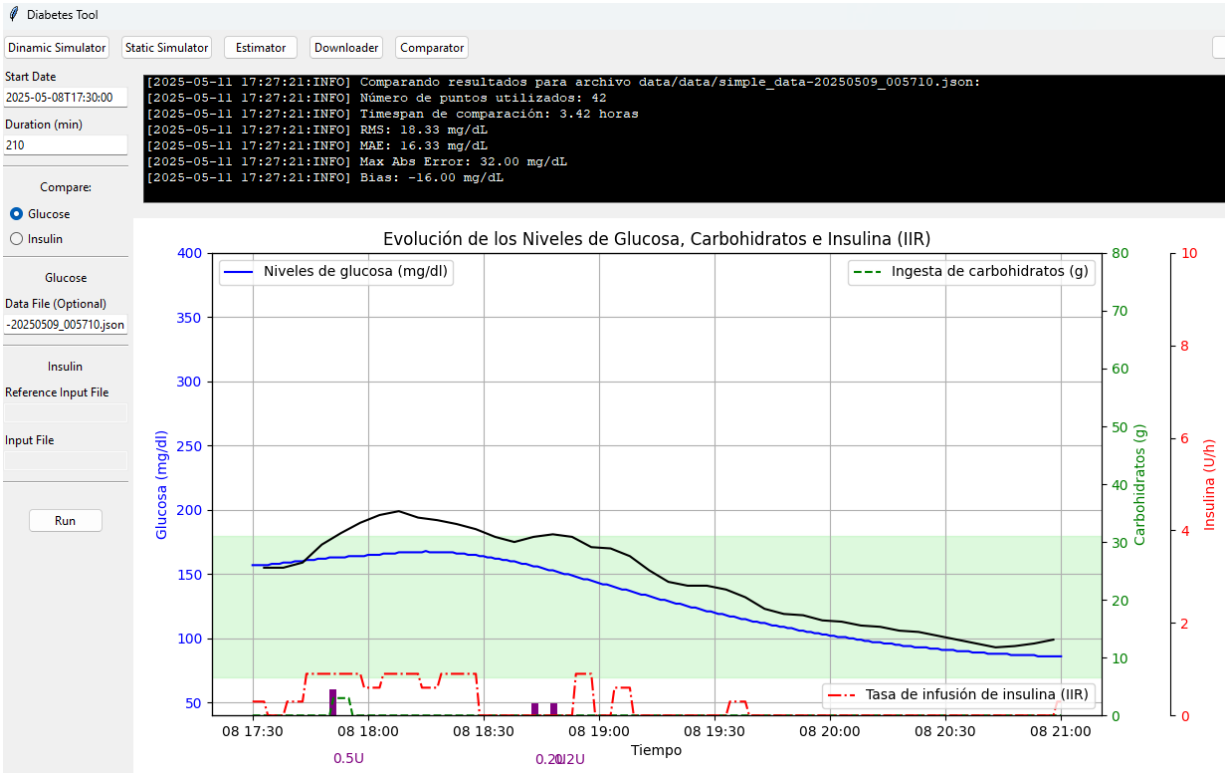


Figura 7.19: Plots de la comparación para el ejercicio físico.

Estimación de parámetros:

Para mejorar el ajuste, se realiza en primer lugar la estimación del parámetro `SIT`, asociado a la sensibilidad de la insulina en la distribución y transporte de la glucosa. Esta estimación permite que el modelo responda de forma más eficaz al aumento de sensibilidad inducido por el ejercicio, mejorando tanto la forma como el momento del descenso de glucosa. Los resultados se muestran en las Figuras 7.20 y 7.21.

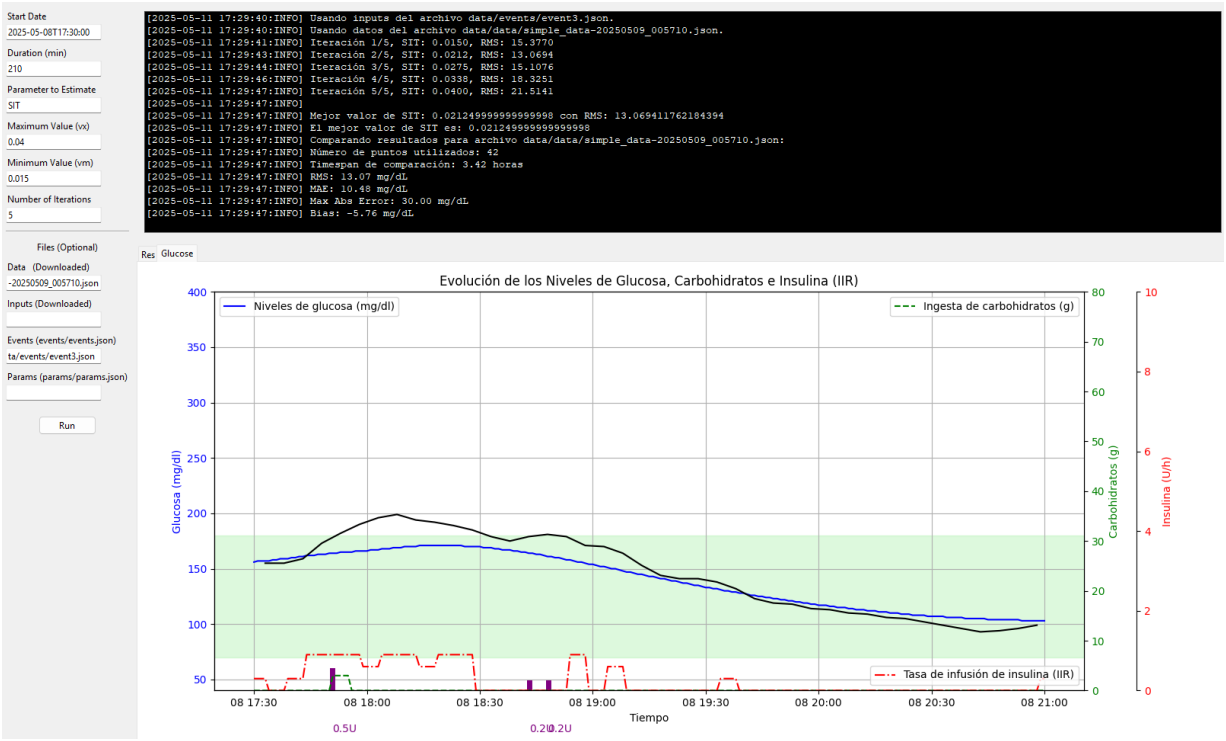


Figura 7.20: Estimación de SIT para el ejercicio físico: evolución de glucosa.

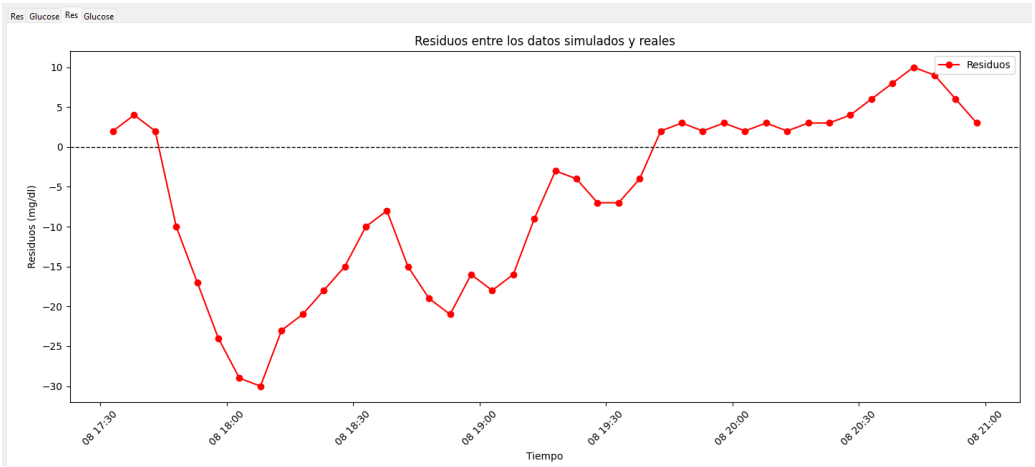


Figura 7.21: Estimación de SIT para el ejercicio físico: evolución de residuos.

Posteriormente, se estimó el parámetro `EGP0`, que representa la producción endógena de glucosa por parte del hígado. La Figura 7.22 muestra el resultado de esta estimación, observándose una mejora sustancial en la precisión del modelo. No se han inlcuído imágenes de los plots ya que son bastante similares a los de la estimación de SIT, y no aportan información adicional relevante.

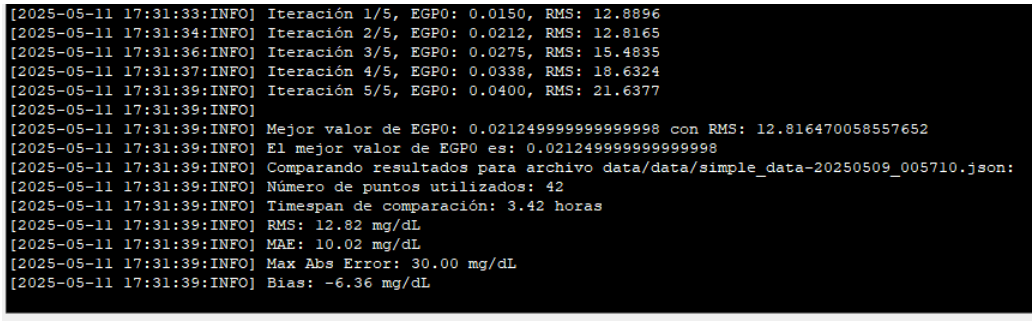


Figura 7.22: Estimación de EGP0 para el ejercicio físico: terminal.

Indicadores de evaluación:

- RMS: 12.82 mg/dL
- MAE: 10.02 mg/dL
- Error absoluto máximo: 30.00 mg/dL
- Bias: -6.36 mg/dL

Interpretación:

Este escenario confirma la capacidad del modelo de simular con precisión la respuesta glucémica inducida por el ejercicio físico, siempre que se realicen las estimaciones de parámetros adecuados. Los indicadores cuantitativos reflejan un ajuste satisfactorio, con errores bajos y sin desviaciones sistemáticas importantes. Este caso demuestra que el modelo puede ser aplicado con éxito incluso en contextos dinámicos donde intervienen factores fisiológicos adicionales como el ejercicio físico.

7.3. Algoritmo de la bomba de insulina

El objetivo es validar que el algoritmo de administración de insulina es capaz de generar decisiones razonables que ayuden a mantener la glucosa dentro de un rango saludable, enfrentándolo a datos de glucosa en tiempo real.

7.3.1. Metodología

- Ejecutar el algoritmo de la bomba frente a la evolución real de la glucosa en algún caso del pasado.
- Comparar las decisiones del algortimo con las acciones tomadas realmente por la bomba real.

- Evaluación basada análisis visual, comparación de cantidad de insulina e interpretación de la comparación. Se compara:
  - Bolos de comidas.
  - Bolos de correcciones.
  - Cantidad basal total.

A continuación se muestran los ratios, tarets y sensibilidades en la interfaz de la aplicación de la insulina, que son los mismos que los de la bomba real de Medtronic.

Figura 7.23: Ratios de CH en función de la hora del día.

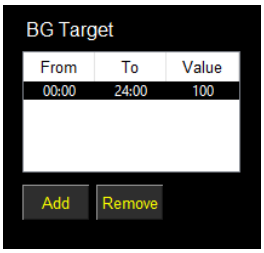


Figura 7.24: Target de glucosa en función de la hora del día.

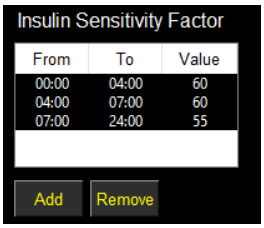


Figura 7.25: Factor de sensibilidad de la insulina en función de la hora del día.

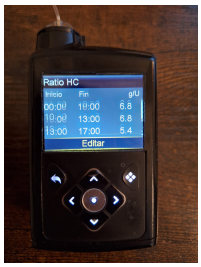


Figura 7.26: Ratios de CH en función de la hora del día en la bomba real.

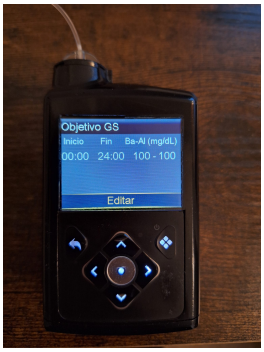


Figura 7.27: Target de glucosa en función de la hora del día en la bomba real.

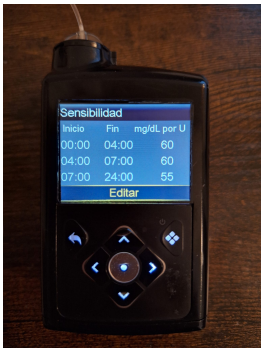


Figura 7.28: Factor de sensibilidad de la insulina en función de la hora del día en la bomba real.

7.3.2. Escenarios de validación

Se analizan diferentes situaciones extraídas de la vida real con el objetivo de validar el comportamiento del algoritmo de la bomba de insulina desarrollado en este trabajo.

7.3.2.1. Comida estándar

Descripción: Para este escenario se utilizan los mismos datos reales que en la validación del modelo de simulación de glucosa en la Sección 7, concretamente en el subapartado sobre la comida estándar. No obstante, en este caso el objetivo no es validar el modelo de simulación, sino comprobar el rendimiento del algoritmo de la bomba de insulina. Se simula el comportamiento del sistema únicamente hasta la segunda comida del día, con el fin de comparar las decisiones tomadas por el algoritmo frente a las que tomó la bomba real utilizada por el paciente.

La Figura 7.29 muestra las ingestas reales cargadas desde el archivo de inputs correspondiente a ese día. En el inicio de la simulación (Figura 7.30), se visualiza un gráfico donde se representa la evolución de la glucosa previa al instante actual, junto con la insulina administrada hasta ese momento. A partir de ahí, el simulador avanza en tiempo acelerado (en este caso con un factor de velocidad `x5`), y va actualizando en tiempo real la curva de glucosa y los eventos de insulina correspondientes, tanto reales como simulados.

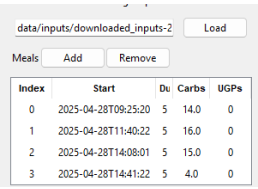


Figura 7.29: Comidas del archivo de inputs de los datos reales para el escenario de comida estándar.

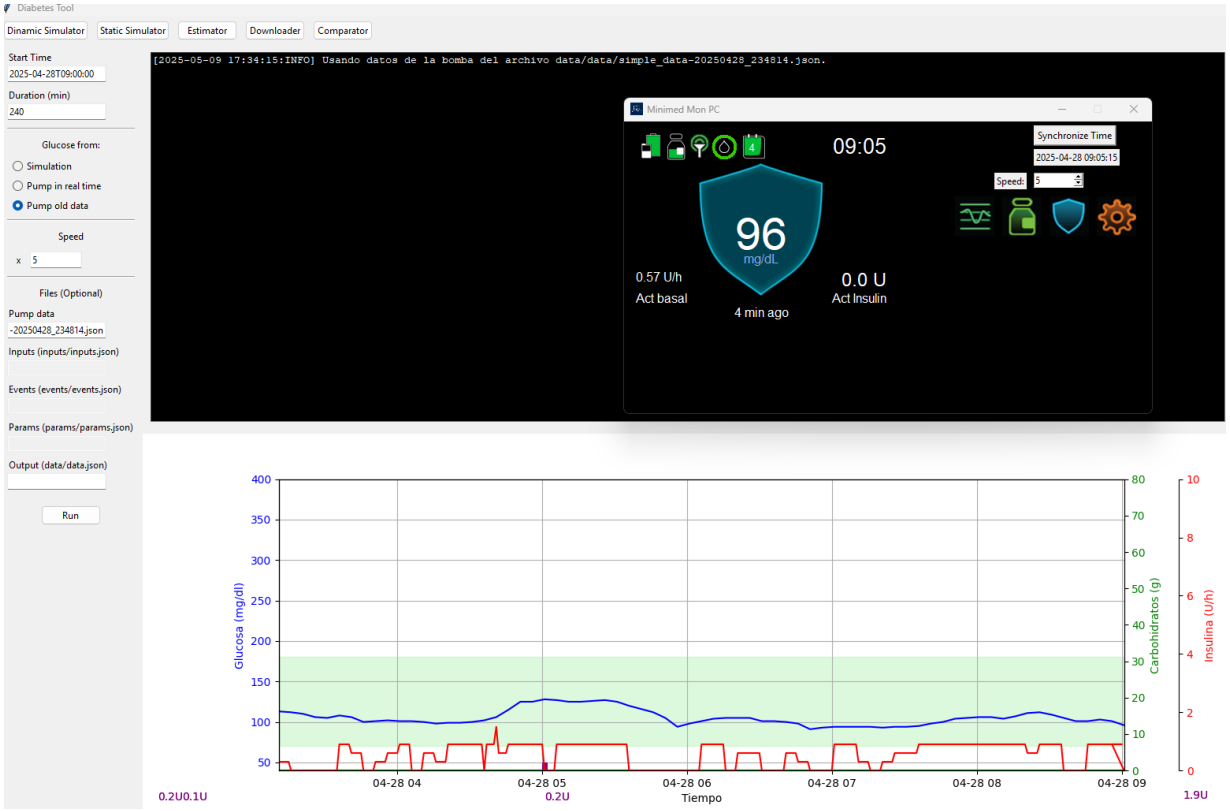


Figura 7.30: Simulador al inicio de la simulación para el escenario de comida estándar.

Comparación de decisiones: A continuación, se muestran las decisiones de la bomba simulada por el algoritmo frente a las decisiones reales tomadas por la bomba del paciente, incluyendo bolo de comida, correcciones y liberación basal. La Figura 7.31 y la Figura 7.32 ilustran los eventos de administración de insulina asociados a las dos primeras ingestas. Finalmente, en la Figura 7.33 se presenta una comparativa general entre los diferentes tipos de insulina administrados.

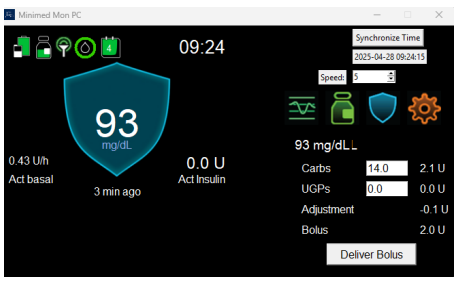


Figura 7.31: Primera comida para el escenario de comida estándar.

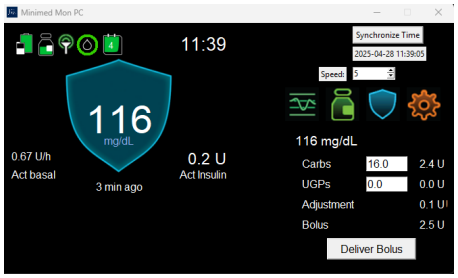


Figura 7.32: Segunda comida para el escenario de comida estándar.

Resumen cuantitativo de insulina administrada (algoritmo vs real):

- **Comida 1:** 2.0 U (algoritmo) vs 1.9 U (real)
- **Comida 2:** 2.5 U (algoritmo) vs 2.3 U (real)
- **Correcciones:** 1.3 U (algoritmo) vs 0.3 U (real)
- **Basal:** 3.37 U (algoritmo) vs 1.55 U (real)

```
[2025-05-10 00:43:49:INFO] === Meal Boluses Comparison ===
[2025-05-10 00:43:49:INFO] Meal at 2025-04-28 09:24:00 — Compare: 2.0 U vs Ref: 1.9 U (Δt = 0:01:20)
[2025-05-10 00:43:49:INFO] Meal at 2025-04-28 11:38:45 — Compare: 2.5 U vs Ref: 2.3 U (Δt = 0:01:37)
[2025-05-10 00:43:49:INFO] === Correction Boluses Comparison ===
[2025-05-10 00:43:49:INFO] Total Correction — Compare: 1.30 U vs Ref: 0.30 U
[2025-05-10 00:43:49:INFO] === Basal Comparison ===
[2025-05-10 00:43:49:INFO] Total Basal      — Compare: 3.37 U vs Ref: 1.55 U
```

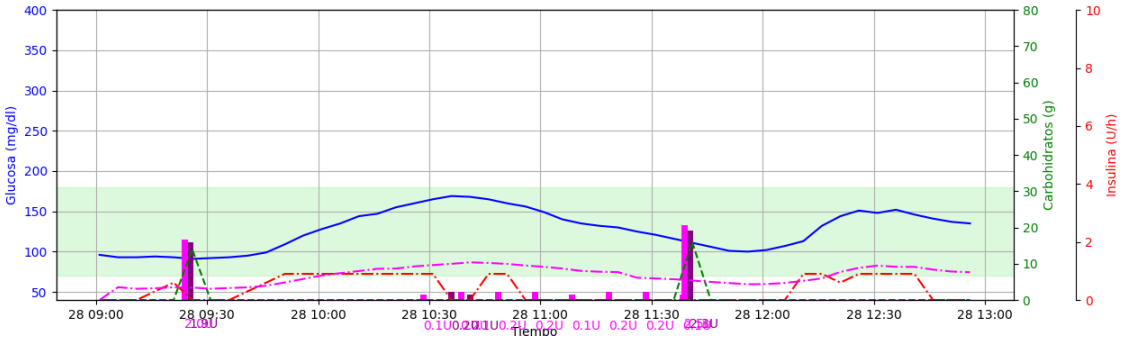


Figura 7.33: Comparación de insulinas para el escenario de comida estándar.

Interpretación: En cuanto a los bolo administrados por comida, se observa una coincidencia muy cercana entre las decisiones del algoritmo y las de la bomba real, lo que refleja una buena estimación de los requerimientos de insulina prandial. En cambio, el algoritmo de la bomba propone una mayor cantidad de insulina para correcciones. Esto se debe a que, tras la primera corrección, el sistema lee los datos reales de glucosa pero no tiene en cuenta que en dichos datos la insulina activa proviene únicamente de la bomba real, y no considera las correcciones adicionales ya administradas por el propio algoritmo. En lo relativo a la insulina basal, el algoritmo administra una cantidad significativamente superior. Esto puede explicarse por la configuración de la bomba real, que mantiene una tasa basal más baja de lo habitual. Como consecuencia, la glucosa tiende a aumentar de forma sostenida, lo que induce al algoritmo a compensar esta subida mediante una mayor liberación de insulina basal.

7.3.2.2. Comidas alta UGP

Descripción:

Este escenario corresponde al mismo caso descrito previamente en la validación del modelo de glucosa (ver Sección 7.13). Se trata de una comida de alto contenido en unidades grasa proteína (UGP), realizada en un restaurante y seguida de una larga sobremesa sin nuevas ingestas. Debido a la complejidad del perfil glucémico posterior y la aparición de un pico tardío, este escenario resulta especialmente interesante para evaluar el comportamiento del algoritmo de la bomba de insulina. **Comparación de decisiones:**

Resumen cuantitativo de insulina administrada (algoritmo vs real):

- **Comida 1:** 2.5 U (algoritmo) vs 1.9 U (real)
- **Comida 2:** 3.5 U (algoritmo) vs 3.0 U (real, el SW detecta que es la corrección de 0.2 U previa, ya que busca el bolo mas cercano en el tiempo a la comida)
- **Comida 3:** 0.4 U (algoritmo) vs 0.8 U (real)
- **Almuerzo:** 3.0 U (algoritmo) vs 1.0 U (real)
- **Correcciones:** 18.00 U (algoritmo) vs 4.3 U (real, corregido por el error en el bolo de la comida 2, que considera su insulina asociada como una corrección)
- **Basal:** 13.47 U (algoritmo) vs 2.83 U (real)

La Figura 7.34 muestra gráficamente esta comparación para el conjunto del día.

Interpretación:

El comportamiento del algoritmo refleja un esfuerzo constante por contrarrestar la hiperglucemia sostenida que se observa en este escenario. En la Figura 7.35 se incluyen anotaciones explicativas sobre las principales decisiones del algoritmo:

- **Basal excesiva:** Se observa una administración continuada de insulina basal debido al alto error integral en el controlador PID. Este comportamiento responde al mantenimiento prolongado de niveles elevados de glucosa.
- **Correcciones elevadas:** El algoritmo aplica múltiples correcciones que no están reflejadas en el fichero de inputs de referencia, lo cual hace que siga intentando aplicar correcciones adicionales.
- **Sobreajuste en el almuerzo:** La insulina administrada durante el almuerzo es superior a la real. Este efecto se debe al mismo fenómeno observado en las correcciones: el algoritmo no parte de un estado inicial que refleje correctamente la insulina activa previa, y responde a valores elevados de glucosa aplicando un ajuste de corrección más grande.

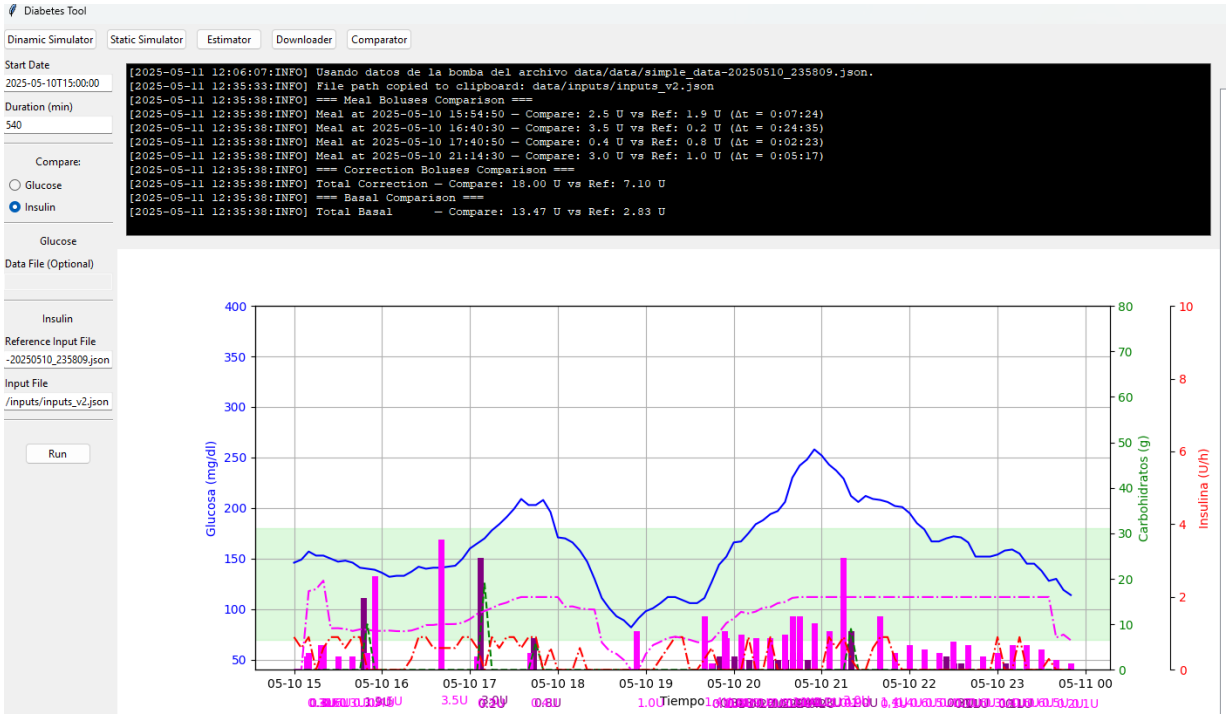


Figura 7.34: Comparación de insulinas para el escenario de comida alta UGP.

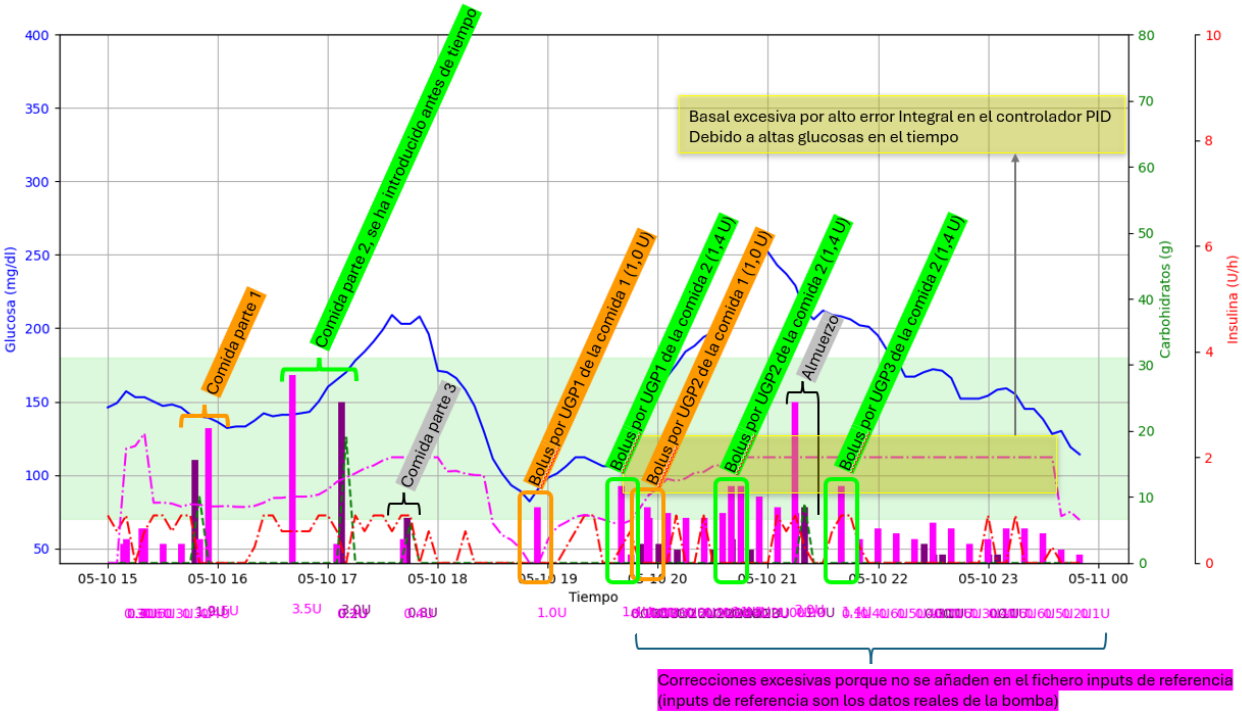


Figura 7.35: Explicación decisiones del algoritmo para el escenario de comida alta UGP.

7.3.2.3. Ejercicio físico

Descripción:

Este escenario reproduce exactamente el mismo caso utilizado en la validación del modelo de

simulación de glucosa: un día en el que se realiza una sesión de gimnasio de 60 minutos (intensidad 5), junto con desplazamientos caminando que también impactan en el perfil glucémico. El objetivo en este caso es evaluar cómo responde el algoritmo de la bomba de insulina ante el descenso glucémico inducido por el ejercicio físico. La aplicación utilizada por el algoritmo permite registrar eventos de ejercicio físico, como se muestra en la Figura 7.36.

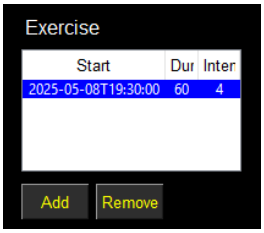


Figura 7.36: Panel de ejercicio en la aplicación de la bomba de insulina.

Comparación de decisiones: Resumen cuantitativo de insulina administrada (algoritmo vs real):

- **Comida:** 0.00 U (algoritmo) vs 0.50 U (real)
- **Correcciones:** 0.00 U (algoritmo) vs 0.40 U (real)
- **Basal:** 1.50 U (algoritmo) vs 0.85 U (real)

En este caso, la comparación entre el algoritmo y los datos reales refleja un comportamiento prudente por parte del algoritmo, que evita administrar bolo durante o inmediatamente después del ejercicio. El perfil de basales es similar al ser admisnitrado únicamente cuando la glucosa es mayor a 150 mg/dL.

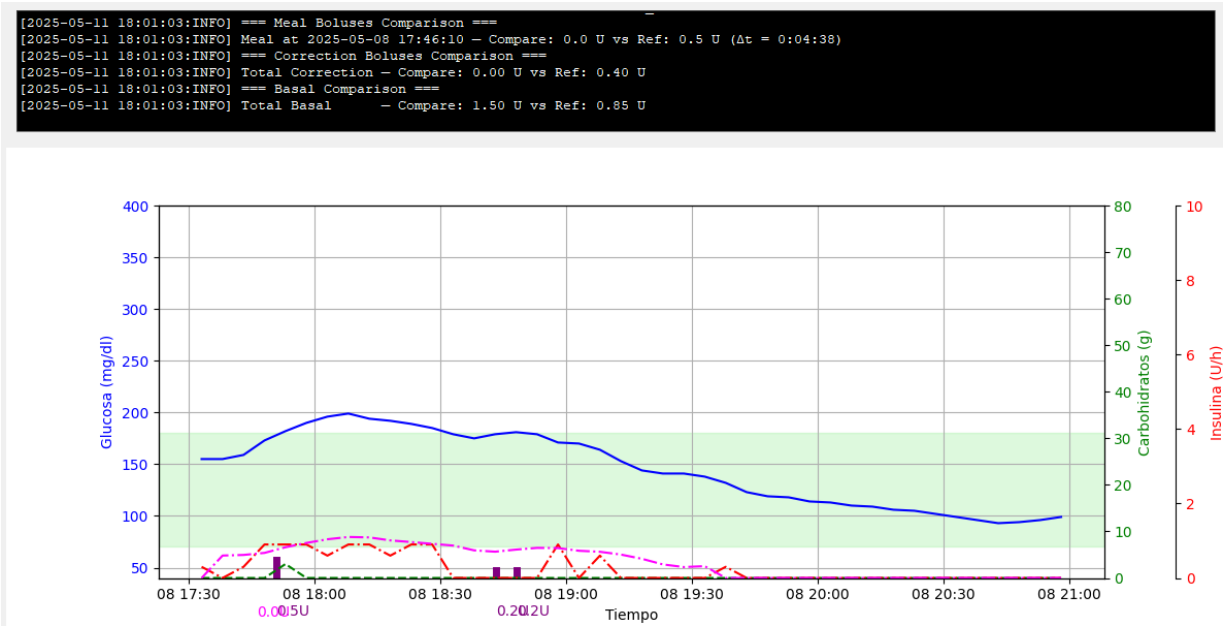


Figura 7.37: Comparación de insulinas para el escenario de ejercicio físico.

Interpretación: El algoritmo de la bomba actúa de forma conservadora ante la presencia de ejercicio físico, suprimiendo los bolo tanto de comida como de corrección, y priorizando una liberación basal ligeramente superior. Esta estrategia busca minimizar el riesgo de hipoglucemia, una de las complicaciones más frecuentes durante y después del ejercicio. La bomba real administró un pequeño bolo de comida y una corrección, probablemente porque en ese momento no se registró correctamente el evento de ejercicio o se aplicaron decisiones manuales. El aumento moderado de la basal por parte del algoritmo se debe a su diseño interno, que responde al descenso de glucosa de forma continua, a través del componente integral del controlador PID.

7.4. Satisfacción de los requisitos del sistema

A partir de los resultados obtenidos en los distintos escenarios de validación, se puede afirmar que el sistema cumple satisfactoriamente los requisitos funcionales y no funcionales definidos en la Sección 3.2.

Requisitos funcionales

- **RF1: Simulación de la glucosa en sangre** — Validado en los escenarios del modelo de simulación (comida estándar, comidas altas en UGP y ejercicio físico), donde se ha comprobado que el modelo reproduce fielmente la evolución glucémica del paciente.
- **RF2: Integración de eventos externos** — Verificado mediante la inclusión de eventos como comidas, ejercicio, infecciones o alcohol, los cuales han sido correctamente interpretados por el sistema y reflejados en la evolución simulada de la glucosa.
- **RF3: Visualización gráfica de resultados** — Cumplido mediante las interfaces de simulación estática y dinámica, donde se presentan gráficamente tanto las variables del modelo como las decisiones de la bomba.
- **RF4: Configuración mediante ficheros JSON** — Confirmado mediante el uso recurrente de los ficheros `params.json`, `inputs.json` y `events.json` en todos los escenarios de validación.

Requisitos no funcionales

- **RNF1: Eficiencia numérica del motor de simulación** — Validado por el correcto funcionamiento del solver adaptativo RK1–RK2 en escenarios de distinta duración y complejidad.
- **RNF2: Diseño modular y extensible** — Confirmado por la capacidad de modificar parámetros fisiológicos, definir nuevos eventos o reconfigurar módulos de manera independiente, sin necesidad de alterar el núcleo del sistema.

- **RNF3: Interfaz gráfica funcional** — Validado a través del uso efectivo de la interfaz desplegada en Raspberry Pi, que permite interactuar con el sistema y visualizar los datos de forma clara y en tiempo real.

7.5. Conclusiones

El proceso de validación ha permitido constatar que tanto el modelo de simulación de glucosa como el algoritmo de la bomba de insulina implementados son capaces de reproducir con un alto grado de realismo situaciones reales experimentadas por pacientes con diabetes tipo 1.

Respecto al modelo de simulación, se ha demostrado que mediante un ajuste adecuado de parámetros clave como `m_delay`, `SIT` o `EGP0`, el sistema puede adaptarse a escenarios diversos, incluyendo picos glucémicos postprandiales, absorciones prolongadas debidas a alimentos ricos en grasas y proteínas, o descensos glucémicos inducidos por el ejercicio físico. Las mejoras observadas en las métricas tras la estimación (como la reducción del RMS o la corrección del bias) refuerzan la validez del enfoque basado en datos reales y ajustes personalizados.

En cuanto al algoritmo de la bomba de insulina, los resultados muestran que sus decisiones (bolo de comida, correcciones, administración de basal) son en general coherentes con las de una bomba comercial. Aunque en algunos casos el algoritmo tiende a sobreadministrar insulina (por ejemplo, en escenarios prolongados de hiperglucemia), este comportamiento está justificado por las características internas del controlador PID adaptativo y por la ausencia de una retroalimentación completa del sistema real. Aun así, el algoritmo ha demostrado sensibilidad a eventos fisiológicos como el ejercicio o las comidas con UGPs, adaptando su respuesta para evitar hipoglucemias y controlar hiperglucemias sostenidas.

Capítulo 8

Conclusiones y trabajos futuros

8.1. Introducción

Este capítulo recoge las conclusiones generales del trabajo desarrollado, así como las posibles líneas de mejora y ampliación que se proponen como trabajos futuros. A lo largo de la presente memoria se ha abordado la problemática de la automatización en la administración de insulina para pacientes con diabetes tipo 1, desarrollando un sistema completo que incluye tanto un modelo de simulación fisiológico como un algoritmo de control embebido en una Raspberry Pi. Asimismo, se ha trabajado con datos reales para validar el comportamiento del sistema, con especial énfasis en la estimación de parámetros y la comparación con una bomba de insulina comercial.

8.2. Conclusiones

A lo largo de este trabajo se ha desarrollado una herramienta modular denominada `DiabetesTool`, capaz de simular, estimar y comparar la evolución de la glucosa en pacientes diabéticos. También se ha desarrollado un algoritmo de control híbrido basado en un modelo PID adaptativo. Las principales conclusiones del trabajo son las siguientes:

- Se ha implementado un modelo de simulación de glucosa basado en el modelo de Cambridge, capaz de representar de forma razonable la evolución de la glucemia ante distintas situaciones como comidas estándar, comidas con alto contenido en grasa y proteína (UGP), y ejercicio físico.
- Se ha creado un algoritmo de control híbrido con componente basal y de corrección, implementado en una Raspberry Pi con pantalla táctil que simula el funcionamiento de una bomba de insulina real. Este algoritmo ha demostrado ser funcional y configurable a través de una interfaz gráfica intuitiva.
- Se ha desarrollado un módulo de descarga de datos reales desde la plataforma CareLink de Medtronic, junto con herramientas específicas para la estimación de parámetros fisiológicos personalizados, como la sensibilidad a la insulina o el retardo en la absorción de carbohidratos. Esto ha permitido validar el modelo de simulación con datos reales y mejorar su precisión.
- Se ha diseñado una metodología sistemática de validación que incluye análisis visuales y cuantitativos (RMS, MAE, Bias) para medir la precisión del modelo y del algoritmo de control frente a datos reales.

- El sistema ha demostrado que, tras una adecuada calibración, es capaz de reproducir de forma precisa distintos perfiles glucémicos reales, y que el algoritmo de la bomba puede tomar decisiones razonables de administración de insulina similares a las de una bomba comercial.

En conjunto, el proyecto ha cubierto los objetivos propuestos y ha demostrado la viabilidad de construir un sistema embebido personalizado para la toma de decisiones en la terapia de la diabetes tipo 1.

8.3. Trabajos futuros

A continuación, se describen algunas líneas de trabajo que podrían desarrollarse en el futuro como continuación natural de este proyecto:

- **Uso de lenguajes y entornos dedicados para modelado fisiológico:** La implementación del modelo de glucosa se ha realizado en Python por motivos de accesibilidad y flexibilidad. No obstante, existen lenguajes dedicados como `Modelica` [39] así como Dymola y OpenModelica, que son dos entornos de modelado que emplean Modelica. De esta forma se podría permitir una descripción más precisa de sistemas fisiológicos mediante ecuaciones diferenciales y arquitecturas orientadas a componentes. Explorar estas herramientas permitiría mejorar la robustez, reutilización y validación formal del modelo.

- **Integración real de las decisiones del algoritmo en la validación con datos reales:** Durante las simulaciones de validación frente a datos reales, el algoritmo de la bomba de insulina detecta niveles elevados de glucosa y decide administrar una corrección o ajustar la insulina basal. Esta decisión se almacena correctamente en el fichero de salida, pero en el siguiente ciclo del algoritmo, la glucosa leída proviene de los datos reales obtenidos desde la bomba comercial, y por tanto no refleja el impacto de la corrección administrada por el algoritmo propio.

  Además, el valor de insulina activa que el algoritmo utiliza para tomar decisiones también es el que proviene de la bomba real, ignorando las decisiones previas del propio algoritmo. Como resultado, se puede generar una repetición innecesaria de bolos o aumentos de basal en ciclos consecutivos, dado que el sistema no percibe un cambio en el estado fisiológico.

  Una mejora futura sería implementar un modelo interno que permita al algoritmo llevar su propio cálculo de la insulina activa basada en las decisiones que va tomando (bolos y basales), en lugar de depender del valor descargado desde la bomba real.

- **Ajuste del comportamiento del PID:** El controlador PID implementado tiende a ser agresivo en la corrección de errores, lo cual puede aumentar el riesgo de hipoglucemias en ciertos casos. Se propone revisar y ajustar su comportamiento para hacerlo más conservador, particularmente en contextos con riesgo elevado de descenso rápido de glucosa, como durante la noche o tras el ejercicio físico.

- **Validación con pacientes reales:** El sistema ha sido validado con datos reales extraídos de un único paciente (yo misma). Una extensión muy valiosa consistiría en contactar con

otras personas con diabetes tipo 1, estimar sus parámetros fisiológicos mediante el módulo de estimación, y evaluar la eficacia del algoritmo de la bomba en distintos perfiles de paciente. Esto permitiría validar la generalización y robustez del sistema.

- **Detección y modelado de eventos externos no explícitos:** Aunque el sistema permite introducir eventos como ejercicio o alcohol manualmente, sería interesante investigar métodos automáticos de detección basados en el análisis del perfil de glucosa, combinando aprendizaje automático con heurísticas clínicas.

- **Evaluación clínica con métricas estandarizadas y entorno de pruebas para profesionales sanitarios**: se propone incorporar una nueva funcionalidad a la herramienta que permita visualizar métricas clínicas clave como el *Time in Range* (TIR), el *Low/High Blood Glucose Index* (LBGI, HBGI), y el *Area Under the Curve* (AUC) de hipoglucemias o hiperglucemias. Estas métricas están validadas clínicamente y facilitarían la evaluación objetiva del sistema. Además, se sugiere desarrollar una interfaz "sandbox" para que médicos o educadores puedan simular casos clínicos modificando parámetros, introduciendo comidas inesperadas o condiciones especiales como ejercicio o enfermedad. Este entorno serviría como herramienta formativa y de validación terapéutica.

- **Exploración de controladores alternativos al PID**: una evolución natural del sistema sería investigar controladores más avanzados como lógica difusa, control adaptativo o incluso redes neuronales entrenadas con datos simulados y reales. Estos enfoques podrían mejorar la adaptabilidad del algoritmo en situaciones complejas (comidas prolongadas, actividad física, enfermedad, etc.), aunque también conllevan mayores requisitos de datos, interpretabilidad y capacidad computacional.

Bibliografía

[1] S. H. Andersen, "Software for in silico testing of an artificial pancreas," *Technical University of Denmark*, January 2014.

[2] Revista Diabetes, "Contaminación del aire y diabetes: ¿Existe relación?" https://www.revistadiabetes.org/miscelanea/contaminacion-del-aire-y-diabetes-existe-relacion/, acceso: noviembre 2024.

[3] A. Thomas and L. Heinemann, "Algorithms for automated insulin delivery: an overview," *Journal of Diabetes Science and Technology*, vol. 16, no. 5, pp. 1228–1238, 2022.

[4] Diabetes Control and Complications Trial Research Group and Nathan, D. M. and Genuth, S. and Lachin, J. and Cleary, P. and Crofford, O. and Davis, M. and Rand, L. and Siebert, C., "The effect of intensive treatment of diabetes on the development and progression of long-term complications in insulin-dependent diabetes mellitus," *The New England Journal of Medicine*, vol. 329, no. 14, pp. 977–986, 1993.

[5] Cleveland Clinic, "Dawn Phenomenon," https://my.clevelandclinic.org/health/diseases/24553-dawn-phenomenon, acceso: noviembre 2024.

[6] Wikipedia, La enciclopedia libre, "Fenómeno de Somogyi," https://es.wikipedia.org/wiki/Fen%C3%B3meno_de_Somogy, acceso: noviembre 2024.

[7] G. A. I. Mason, J. Qian and F. Scheer, "Impact of circadian disruption on glucose metabolism: implications for type 2 diabetes," *Diabetologia*, vol. 63, no. 3, pp. 462–472, Mar 2020, epub 2020 Jan 8, PMID: 31915891, PMCID: PMC7002226.

[8] P. R. López, "Impacto del microbioma intestinal humano en la diabetes," 2016.

[9] Beyond Type 1, "La altitud y la diabetes tipo 1," https://es.beyondtype1.org/la-altitud-y-la-diabetes-tipo-1/, acceso: noviembre 2024.

[10] U.S. Food and Drug Administration (FDA), "Infusion Pumps," https://www.fda.gov/medical-devices/general-hospital-devices-and-supplies/infusion-pumps, acceso: noviembre 2024.

[11] "The Open-Source Framework for Diabetes Simulation," https://lt1.org/, acceso: octubre 2024.

[12] H. Peuscher, "LoopInsight1: An Open-Source Framework for Diabetes Simulation," https://github.com/hpeuscher/loopinsight1, acceso: octubre 2024.

[13] C. D. Man, R. Rizza, and C. Cobelli, "Meal simulation model of the glucose-insulin system," *IEEE Transactions on Biomedical Engineering*, vol. 54, pp. 1740–1749, 2007. [Online]. Available: https://api.semanticscholar.org/CorpusID:206609877

[14] C. Dalla Man, M. D. Breton, and C. Cobelli, "Physical activity into the meal glucose—insulin model of type 1 diabetes: In silico studies," 2009.

[15] J. Manrique-Córdoba, J. D. Romero-Ante, A. Vivas, J. M. Vicente, and J. Sabater-Navarro, "Modelado matemático de ingestas de alimento e infusión de insulina en un paciente con diabetes tipo 1 en lazo cerrado," *Revista Iberoamericana de Automática e Informática industrial*, vol. 17, no. 2, pp. 156–168, 2020.

[16] R. Hovorka, V. Canonico, L. J. Chassin, U. Haueter, M. Massi-Benedetti, M. O. Federici, T. R. Pieber, H. C. Schaller, L. Schaupp, T. Vering *et al.*, "Nonlinear model predictive control of glucose concentration in subjects with type 1 diabetes," *Physiological measurement*, vol. 25, no. 4, p. 905, 2004.

[17] R. Hovorka, F. Shojaee-Moradie, P. V. Carroll, L. J. Chassin, I. J. Gowrie, N. C. Jackson, R. S. Tudor, A. M. Umpleby, and R. H. Jones, "Partitioning glucose distribution/transport, disposal, and endogenous production during ivgtt," *American Journal of Physiology-Endocrinology and Metabolism*, vol. 282, no. 5, pp. E992–E1007, May 2002.

[18] "OpenAPS's documentation," https://openaps.readthedocs.io/en/latest/, acceso: enero 2025.

[19] "OpenAPS GitHub," https://github.com/openaps, acceso: enero 2025.

[20] "Tandem Diabetes Care," https://www.tandemdiabetes.com/es-es/home, acceso: enero 2025.

[21] "Medtronic MiniMed™ 780G insulin pump system," https://www.medtronic-diabetes.com/en-gb/insulin-pump-therapy/minimed-780g-system, acceso: enero 2025.

[22] "LoopDocs," https://loopkit.github.io/loopdocs/, acceso: enero 2025.

[23] "AAPS documentation," https://androidaps.readthedocs.io/en/latest/, acceso: enero 2025.

[24] "AAPS GitHub," https://github.com/nightscout/AndroidAPS, acceso: enero 2025.

[25] F. Chee, T. L. Fernando, A. V. Savkin, and V. Van Heeden, "Expert pid control system for blood glucose control in critically ill patients," *IEEE Transactions on Information Technology in Biomedicine*, vol. 7, no. 4, pp. 419–425, 2003.

[26] M. Fliess and C. Join, "Intelligent pid controllers," in *2008 16th mediterranean conference on control and automation*. IEEE, 2008, pp. 326–331.

[27] T. Niang, C. Zhong *et al.*, "Discussion on nonlinear pid based on nonlinear norm transformation controlling method applied to svc," in *2008 27th Chinese Control Conference*. IEEE, 2008, pp. 61–63.

[28] I. Podlubny, "Fractional-order systems and fractional-order controllers," *Institute of Experimental Physics, Slovak Academy of Sciences, Kosice*, vol. 12, no. 3, pp. 1–18, 1994.

[29] C. Li and R. Hu, "Fuzzy-pid control for the regulation of blood glucose in diabetes," in *2009 WRI Global Congress on Intelligent Systems*, vol. 2. IEEE, 2009, pp. 170–174.

[30] G. M. Steil, K. Rebrin, C. Darwin, F. Hariri, and M. F. Saad, "Feasibility of automating insulin delivery for the treatment of type 1 diabetes," *Diabetes*, vol. 55, no. 12, pp. 3344–3350, 2006.

[31] S. A. Weinzimer, G. M. Steil, K. L. Swan, J. Dziura, N. Kurtz, and W. V. Tamborlane, "Fully automated closed-loop insulin delivery versus semiautomated hybrid control in pediatric patients with type 1 diabetes using an artificial pancreas," *Diabetes care*, vol. 31, no. 5, pp. 934–939, 2008.

[32] C. Yoldi. Las grasas y las proteínas también cuentan. https://diabetes.sjdhospitalbarcelona.org/es/diabetes-tipo-1/consejos/grasas-proteinas-tambien-cuentan. Acceso: Octubre 2024.

[33] Dra G. Sesmilo León y P. Lecegui. Unidad de grasas y proteínas. https://www.endocrino.cat/es/blog-endocrinologia.cfm/ID/15085/ESP/unidad-grasas-porteinas-ugp-.htm. Acceso: Octubre 2024.

[34] G. Tascini, M. G. Berioli, L. Cerquiglini, E. Santi, G. Mancini, F. Rogari, G. Toni, and S. Esposito, "Carbohydrate counting in children and adolescents with type 1 diabetes," *Nutrients*, vol. 10, no. 1, p. 109, 2018.

[35] R. Hanas and P. Adolfsson, "Bolus calculator settings in well-controlled prepubertal children using insulin pumps are characterized by low insulin to carbohydrate ratios and short duration of insulin action time," *Journal of diabetes science and technology*, vol. 11, no. 2, pp. 247–252, 2017.

[36] WeCare Blog , "Frecuencia de cambio de los equipos de infusión y los reservorios minimed: la experiencia de una persona con diabetes como tú," https://www.medtronic-diabetes.com/es-ES/blog/frecuencia-de-cambio-de-los-equipos-de-infusion-y-los-reservorios-minimed-la-experiencia-de, acceso: marzo 2025.

[37] Ondrej1024, "carelink-python-client," https://github.com/ondrej1024/carelink-python-client, 2024, accedido en abril de 2025.

[38] W. contributors, "*Root mean square*, wikipedia, the free encyclopedia," https://en.wikipedia.org/wiki/Root_mean_square, 2024, accedido en abril de 2025.

[39] "Sitio web de la Modelica Association," https://modelica.org/, acceso: mayo 2025.

[40] Asociación Americana de la Diabetes, "El impacto de la deshidratación en el manejo de la diabetes," https://www.amdiabetes.org/post/el-impacto-de-la-deshidrataci%C3%B3n-en-el-manejo-de-la-diabetes, acceso: noviembre 2024.

[41] Wikipedia, "Samba (software)," https://es.wikipedia.org/wiki/Samba_(software), acceso: abril 2025.

[42] "Cómo compartir una carpeta con Samba entre tu Raspberry Pi y otro ordenador de tu red," https://raspberryparanovatos.com/tutoriales/compartir-carpeta-samba-raspberry-pi-windows/, acceso: abril 2025.

Apéndice A

Manual de usuario de DiabetesTool

A.1. Introducción

La herramienta *DiabetesTool* ha sido desarrollada como parte del Proyecto de Fin de Grado para simular, ajustar y comparar la evolución de la glucosa simulada frente a la real en personas con diabetes tipo 1. Esta herramienta ofrece una interfaz gráfica de usuario (GUI) intuitiva que permite ejecutar distintos módulos especializados con diferentes funcionalidades. La herramienta consta de cinco modos de ejecución:

- **Dynamic Simulator**: realiza una simulación de manera dinámica de la evolución de la glucosa en una persona con diabetes, permitiendo configurar la velocidad de simulación y origen de los datos de glucosa (simulación o bomba) y visualizar la evolución en tiempo real.
- **Static Simulator**: ejecuta una simulación de manera estática, útil para realizar análisis rápidos con condiciones fijas. Permite definir la fecha de inicio, duración de la simulación y origen de los datos de glucosa (simulación o bomba).
- **Estimator**: estima parámetros personalizados del modelo de diabetes a partir de datos históricos, ajustando el comportamiento del modelo a las características específicas del paciente.
- **Downloader**: descarga automáticamente los datos históricos desde la nube de Medtronic, haciendo uso de un token previamente generado.
- **Comparator**: compara los datos reales de Medtronic con una simulación estática realizada con los mismos valores de insulina y carbohidratos, permitiendo validar la precisión del modelo.

A.2. Requisitos del sistema

Para ejecutar correctamente la herramienta *DiabetesTool*, se requiere disponer de un entorno con las siguientes características:

- Python 3.9 o superior
- Sistema operativo: Windows, Linux o macOS

- Los siguientes paquetes de Python deben estar instalados:
  - `matplotlib`
  - `numpy`
  - `requests`
  - `python-dateutil`
  - `pyOpenSSL`
  - `pyperclip`
  - `curlify`
  - `selenium-wire`
  - `tkinter` (puede requerir instalación adicional en algunas distribuciones)

Se recomienda instalar todos los paquetes necesarios mediante el comando:

```
pip install -r requirements.txt
```

donde `requirements.txt` contiene la lista completa de dependencias con sus versiones.

A.3. Ejecución de la herramienta

Para ejecutar la herramienta *DiabetesTool*, basta con lanzar el archivo principal desde la terminal, siempre que se esté ubicado en el directorio donde reside el archivo `main.py`. No es necesario ningún instalador ni ejecutable adicional.

Desde una terminal, en el sistema operativo que se utilice (Windows, Linux o macOS), se puede ejecutar el siguiente comando:

```
python main.py
```

o bien, si se tiene configurado el entorno para usar `python3` como comando principal:

```
python3 main.py
```

Una vez lanzado el programa, se abrirá la interfaz gráfica de usuario (GUI) desde la cual se puede acceder a todas las funcionalidades descritas en la siguiente sección.

A.4. Descripción de la interfaz

A continuación se describe la interfaz gráfica de usuario (GUI) de la herramienta *DiabetesTool*, que consta de varios paneles organizados para facilitar la interacción con los distintos modos de ejecución.

A.4.1. Panel superior: selección de modo

En la parte superior de la interfaz se encuentran tantos botones como modos de ejecución tiene la herramienta. Cada uno corresponde a uno de los programas descritos previamente (Dynamic Simulator, Static Simulator, Estimator, Downloader y Comparator).

Cada vez que se selecciona uno de estos modos, el panel izquierdo se actualiza para mostrar los campos de entrada y parámetros específicos asociados a dicho programa.

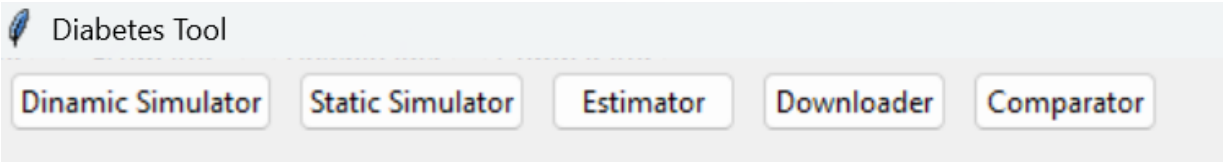


Figura A.1: Panel superior de DiabetesTool con botones de selección de modo.

A.4.2. Panel izquierdo: parámetros del modo

Dependiendo del modo, los campos cambian. Cada vez que se selecciona uno de estos programas mediante los botones superiores, la parte izquierda de la GUI se actualiza con los campos específicos de entrada y control de ese modo (por ejemplo: `Start Time`, `Duration`, `Speed`, etc.).

En cada uno de los casos de uso explicados más adelante se detallarán los parámetros específicos requeridos para cada modo de ejecución.

A.4.3. Panel derecho: pestañas de archivos

A la derecha de la interfaz, existen cuatro pestañas comunes a todos los modos de ejecución:

- **Files**: muestra una vista en forma de árbol (*treeview*) con todos los archivos que lee o escribe la herramienta, organizados por carpetas: `data`, `inputs`, `events` y `params`.
- **Inputs**: muestra el contenido del archivo `inputs.json` con una vista tipo MMI (interfaz hombre-máquina), permitiendo modificar fácilmente los eventos de comidas (carbohidratos), bolos y basales de insulina.

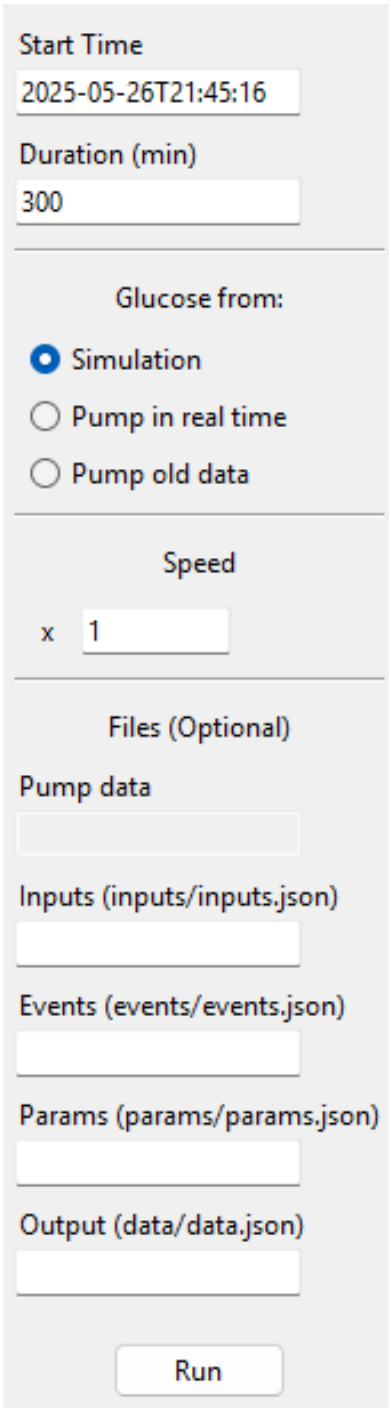


Figura A.2: Panel izquierdo de DiabetesTool con campos de entrada para el modo seleccionado.

- **Parameters**: muestra los parámetros del modelo (como masa corporal, volumen de distribución, coeficientes de absorción, etc.) presentes en `params.json`, que pueden ser editados.
- **Events**: muestra el contenido del archivo `events.json`, también en formato MMI para su edición.

Nota: No se incluye visualización para los archivos de salida (`data.json`), ya que no deben ser modificados manualmente.

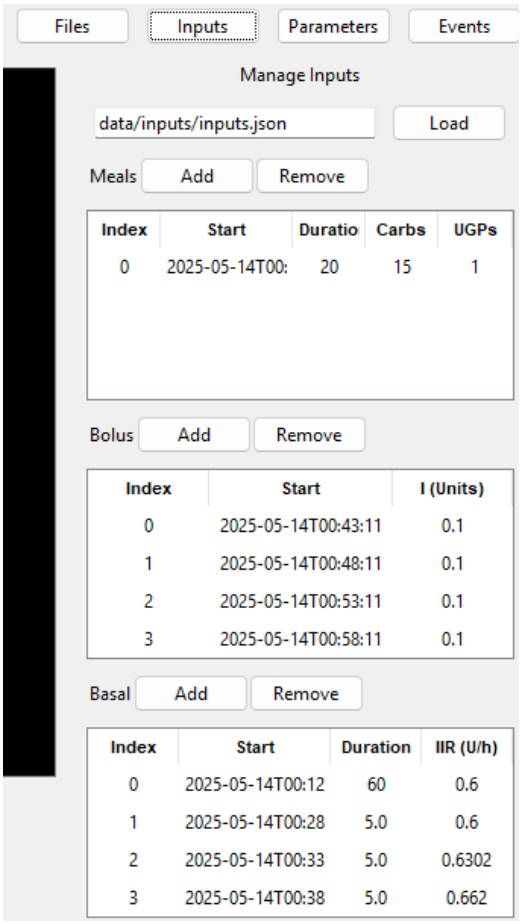


Figura A.3: Panel derecho de DiabetesTool con pestañas de archivos.

A.4.4. Panel central

- Parte superior: consola de salida. En la parte central superior de la GUI se encuentra un panel negro que actúa como consola, mostrando las salidas de terminal de cada ejecución, lo cual es útil para seguimiento en tiempo real y depuración.

- Parte inferior: gráficos de salida. En la parte central inferior se muestran los gráficos generados por cada ejecución (si aplica), incluyendo la evolución de la glucosa, administración de insulina y consumo de carbohidratos.

A.5. Casos de uso

A continuación se describen los diferentes casos de uso de la herramienta.

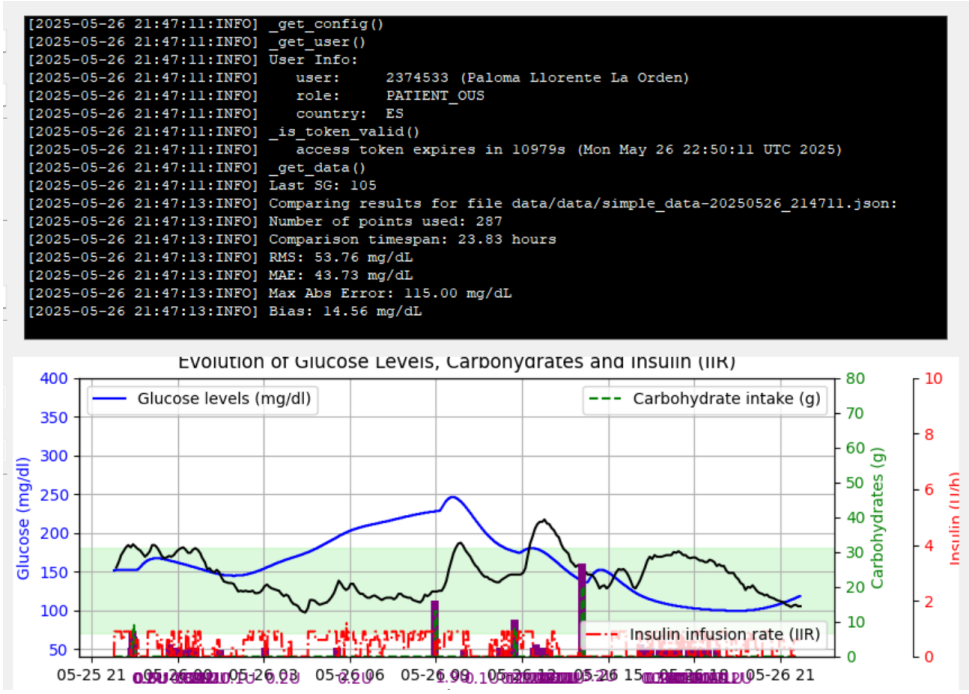


Figura A.4: Panel central de DiabetesTool con la consola y gráficos.

A.5.1. Dynamic Simulator

Este modo permite ejecutar una simulación con visualización dinámica en tiempo real de la evolución de la glucosa. A medida que el tiempo avanza —a una velocidad configurable—, el sistema actualiza progresivamente el estado del paciente simulando la acción de los carbohidratos y la insulina sobre los niveles de glucosa. Esta visualización paso a paso proporciona una percepción temporal continua, similar al comportamiento de un sistema real en operación.

Desde el punto de vista del modelado matemático y de sistemas, el modelo utilizado es inherentemente dinámico, ya que describe cómo cambian las variables del sistema (como la glucosa en sangre) a lo largo del tiempo mediante un conjunto de ecuaciones diferenciales ordinarias (ODEs). En ingeniería de sistemas, un modelo dinámico se caracteriza precisamente por capturar la evolución temporal del estado del sistema, siendo sensible a condiciones iniciales y entradas variables en el tiempo

Parámetros de entrada (panel izquierdo)

Los campos configurables para este modo son:

- **Start Time**: fecha y hora de inicio de la simulación.
- **Duration (min)**: duración total de la simulación en minutos.
- **Glucose from**: origen de los datos de glucosa inicial (opciones: `Simulation`, `Pump in real time` o `Pump old data`).
- **Speed**: controla la velocidad con la que avanza el tiempo en la simulación dinámica.

Archivos utilizados (Obligatorio si datos de `Pump old data`)

El único archivo obligatorio si los datos de glucosa simulados vienen de datos históricos de la bomba real es el archivo:

- **Pump data**: formato `data.json`.

Archivos utilizados (opcional)

Si no se especifican, se usan los valores por defecto. Los archivos que pueden configurarse son:

- **Inputs** (`inputs/inputs.json`): incluye información sobre las ingestas de carbohidratos y administración de insulina.
- **Events** (`events/events.json`): contiene eventos adicionales que afectan la evolución de la glucosa.
- **Params** (`params/params.json`): incluye los parámetros fisiológicos del modelo.
- **Output** (`data/data.json`): archivo de salida donde se guardan los datos generados por la simulación.

Ejecución y resultados

Una vez configurados los parámetros y pulsado el botón **Run**, la herramienta comienza a simular dinámicamente la evolución de la glucosa. En la parte superior central de la interfaz, se muestran mensajes en la consola con el progreso de la ejecución.

Simultáneamente, en la parte inferior central se va generando progresivamente un gráfico que muestra la evolución de la glucosa, la insulina y los carbohidratos en tiempo real.

Además, el archivo de salida especificado (por defecto `data/data.json`) se va completando dinámicamente con los datos simulados y su correspondiente estampa de tiempo.

A.5.2. Static Simulator

Este modo ejecuta el mismo modelo dinámico de evolución de la glucosa, pero presenta los resultados de manera instantánea para todo el intervalo de simulación definido por el usuario. A diferencia del modo anterior, no se muestra el avance progresivo del tiempo, sino que se calcula y visualiza de forma directa la evolución completa de la glucosa entre el instante inicial y final.

Es importante señalar que, aunque este modo se denomina "estático", no implica que el modelo subyacente sea estático en el sentido estricto de la teoría de sistemas. En dicha teoría, un

modelo estático es aquel que describe relaciones algebraicas entre variables sin considerar su comportamiento dinámico ni evolución temporal, como ocurre en modelos en estado estacionario o de equilibrio. En contraste, el modelo utilizado aquí sigue siendo dinámico, ya que incluye explícitamente dependencias temporales y relaciones causales entre entradas y estados del sistema. Lo que cambia entre ambos modos es únicamente la forma de presentar los resultados: de manera animada y continua en el tiempo (modo dinámico), o como una evaluación directa del comportamiento completo (modo estático).

Parámetros de entrada (panel izquierdo)

Los campos configurables para este modo son:

- **Start Time**: fecha y hora de inicio de la simulación.
- **Duration (min)**: duración total de la simulación en minutos.

Archivos utilizados (opcional)

Si no se especifican, se utilizan los archivos por defecto. Los archivos son:

- **Inputs** (`inputs/inputs.json`): contiene la información sobre ingestas de carbohidratos y administración de insulina.
- **Events** (`events/events.json`): define eventos adicionales como ejercicio, enfermedades, etc.
- **Params** (`params/params.json`): parámetros fisiológicos del modelo.

Simulación múltiple (opcional)

El modo estático permite realizar múltiples simulaciones variando un parámetro fisiológico específico. Si se desea realizar esta comparación, deben rellenarse los siguientes campos adicionales:

- **Parameter to Vary**: nombre del parámetro que se desea modificar.
- **Minimum Value**: valor mínimo del parámetro.
- **Maximum Value**: valor máximo del parámetro.
- **Number of Simulations**: número de simulaciones a ejecutar entre los valores mínimo y máximo definidos.

Ejecución y resultados

Al pulsar el botón **Run**, se ejecuta la simulación. En la parte superior central de la GUI se muestra información sobre el proceso, y en la parte inferior se generan los siguientes gráficos:

- Evolución de la glucosa a lo largo del tiempo, junto con las administraciones de insulina y las ingestas de carbohidratos.
- Evolución de los parámetros fisiológicos seleccionados (si aplica).
- Evolución de las variables de estado internas del modelo.

A.5.3. Estimator

Este modo permite estimar el valor óptimo de un parámetro fisiológico del modelo de diabetes a partir de datos reales de glucosa. La herramienta ajusta el modelo para que su comportamiento se asemeje lo máximo posible a los datos observados.

Parámetros de entrada (panel izquierdo)

Los campos configurables para este modo son:

- **Start Time**: fecha y hora de inicio del intervalo de análisis.
- **Duration (min)**: duración total de la estimación en minutos.
- **Parameter to Estimate**: nombre del parámetro que se desea estimar.
- **Minimum Value**: valor mínimo permitido para el parámetro.
- **Maximum Value**: valor máximo permitido para el parámetro.
- **Number of Iterations**: número de iteraciones del proceso de estimación.

Archivos utilizados (opcional)

Si no se especifican, se utilizan los valores por defecto. Los archivos son:

- **Data (downloaded)**: archivo con los datos de glucosa real y sus respectivas estampas de tiempo. Si no se indica, se descarga automáticamente desde la nube de Medtronic.
- **Inputs (downloaded)**: contiene las ingestas de carbohidratos y las administraciones de insulina correspondientes al mismo periodo. También se descargan si no se especifican.
- **Events** (`events/events.json`): eventos adicionales que influyen en la simulación.
- **Params** (`params/params.json`): archivo base con los parámetros fisiológicos del modelo.

Ejecución y resultados

Tras configurar los parámetros y pulsar el botón **Run**, la herramienta inicia el proceso de estimación. En la consola (panel superior central) se muestra el progreso del algoritmo, incluyendo el valor del parámetro y el error en cada iteración.

Como resultado, se generan los siguientes gráficos en la parte inferior:

- Evolución simulada de la glucosa en comparación con la real, usando el parámetro estimado.
- Evolución de carbohidratos e insulina durante el periodo analizado.
- Gráfica de residuos: diferencia entre los valores simulados y reales de glucosa a lo largo del tiempo.

Los residuos se calculan mediante el método **RMS** (Root Mean Square), que proporciona una medida de error global basada en la raíz cuadrada del promedio de los errores al cuadrado.

A.5.4. Downloader

Este modo permite descargar automáticamente los datos históricos de glucosa, insulina y carbohidratos desde la nube de Medtronic. Es útil para obtener datos reales que luego pueden utilizarse en los modos Estimator o Comparator.

Parámetros de entrada (panel izquierdo)

Los campos configurables para este modo son:

- **Repeat Time**: número de veces que se repetirá la descarga.
- **Wait Time (min)**: intervalo de espera (en minutos) entre cada descarga cuando el número de repeticiones es mayor que uno.
- **Save Data (flag)**: si está activado, guarda los datos descargados en los archivos correspondientes.
- **Verbose Mode (flag)**: si está activado, muestra información detallada del proceso en la consola.

Archivos utilizados

No requiere archivos de entrada. Si el flag `Save Data` está activado, los datos descargados se almacenan en los archivos estándar de entrada del programa (como `inputs.json` y `data.json`).

Ejecución y resultados

Al pulsar el botón **Run**, se inicia el proceso de descarga. Si se han activado varias repeticiones, la herramienta esperará el número de minutos definido en `Wait Time` entre una descarga y la siguiente.

Durante la ejecución, se muestra información en la consola (panel superior central), incluyendo el estado de conexión y los datos recibidos (si `Verbose Mode` está activado).

Como salida, se genera un gráfico en la parte inferior de la interfaz que muestra la evolución completa de la glucosa real obtenida desde Medtronic.

A.5.5. Comparator

Este modo permite comparar la evolución de la glucosa real registrada por la bomba (obtenida desde Medtronic) con una simulación estática utilizando los mismos valores de insulina y carbohidratos. También permite comparar entre dos ficheros con datos de entrada de insulina, para poder compararlos. Es útil para validar la precisión del modelo bajo condiciones reales.

Parámetros de entrada (panel izquierdo)

Los campos configurables para este modo son:

- **Start Time**: fecha y hora de inicio del intervalo de comparación.
- **Duration (min)**: duración total del periodo en minutos.

Comparar

Se puede elegir entre dos opciones:

- **Glucose**: evolución de la glucosa real frente a la simulada.
- **Insulin**: cantidad de insulina suministrada entre dos ficheros.

Archivos utilizados para comparar glucosa (opcional)

Si no se indican manualmente, se utilizan los valores por defecto o se descargan automáticamente desde Medtronic:

- **Data File**: archivo con los datos reales de glucosa. Si no se especifica, se descarga desde la nube.

Archivos utilizados para comparar insulina

- **Reference input File**: archivo de referencia con los datos de insulina a comparar.
- **Input File**: archivo con los datos de insulina a comparar.

Ejecución y resultados

Tras pulsar el botón **Run**, se ejecuta la comparación seleccionada:

- **Comparación de glucosa:** se realiza una simulación del modelo utilizando los datos reales de carbohidratos e insulina, y se compara con la evolución real de la glucosa registrada por la bomba (archivo `data.json`). Como resultado, se genera un gráfico con ambas curvas (real y simulada), así como las siguientes métricas de evaluación:
  - RMS: raíz del error cuadrático medio.
  - MAE: error absoluto medio.
  - Max Abs Error: error absoluto máximo.
  - Bias: media de los errores.
- **Comparación de insulina:** se analizan dos archivos distintos con datos de insulina (uno de referencia y uno a comparar). Se genera una visualización de las dosis administradas a lo largo del tiempo, diferenciando entre:
  - Insulina en bolos de comida: cantidad total de insulina administrada asociada a ingestas.
  - Insulina de corrección: total de unidades administradas para corregir hiperglucemias.
  - Insulina basal: cantidad total de insulina basal suministrada durante el periodo.

A.6. Conclusiones

La herramienta *DiabetesTool* ha sido diseñada como una plataforma versátil y modular para el análisis y simulación de la evolución de la glucosa en personas con diabetes tipo 1. Su interfaz gráfica intuitiva, junto con la integración de datos reales procedentes de Medtronic, permite tanto realizar simulaciones dinámicas y estáticas como ajustar parámetros fisiológicos y validar modelos personalizados.

Gracias a sus cinco modos de ejecución, la herramienta cubre un amplio rango de funcionalidades, desde la exploración de escenarios hasta la comparación con datos reales y la estimación automática de parámetros. Esto la convierte en una herramienta útil tanto en entornos de investigación como en contextos clínicos simulados o de desarrollo.

Apéndice B

Manual de usuario de la bomba de insulina

B.1. Introducción

Como parte del Proyecto de Fin de Grado, se ha desarrollado una herramienta que simula el funcionamiento de una bomba de insulina, ejecutada en una Raspberry Pi. Esta bomba ejecuta en tiempo real el algoritmo de control implementado, y permite al usuario (o paciente) interactuar con el sistema a través de una interfaz gráfica de usuario (GUI) sencilla e intuitiva.

B.2. Requisitos del sistema

Para ejecutar correctamente la bomba de insulina simulada, se requiere un entorno basado en los siguientes componentes:

- Un modelo de **Raspberry Pi** compatible con entorno gráfico, similar al modelo `Raspberry Pi B+`.
- Una **tarjeta microSD** con al menos 16 GB de capacidad, utilizada para alojar la imagen del sistema operativo y los archivos del proyecto.
- Un sistema operativo tipo **Raspberry Pi OS Bookworm 32-bit** (por ejemplo, versión `2024-07-04`), o cualquier versión equivalente compatible con Python y GUI.
- Pantalla táctil de 5 pulgadas y miniteclado conectados a la Raspberry Pi (por HDMI o USB).
- Python 3.9 o superior.

Además, deben instalarse los siguientes paquetes de Python:

- `Pillow` (para el manejo de imágenes, usado a través de `PIL`)
- `matplotlib`
- `tkinter` (puede requerir instalación adicional en algunas distribuciones)

Los siguientes módulos se utilizan pero pertenecen a la librería estándar de Python, por lo que no requieren instalación mediante `pip`:

- `datetime`
- `json`
- `logging`
- `sys`
- `threading`
- `time`

Se recomienda tener conexión a red local para el uso compartido de archivos mediante Samba, especialmente si se desea recibir datos desde un PC externo.

B.3. Ejecución

Para iniciar el sistema, basta con encender la Raspberry Pi. Una vez el sistema haya arrancado, se debe abrir una terminal y ejecutar el archivo principal del programa. Desde el directorio donde se encuentra `main.py`, el comando es:

```
python3 main.py
```

Tras la ejecución, se abrirá automáticamente la interfaz gráfica, donde el usuario podrá visualizar el estado actual de la glucosa e interactuar con la bomba simulada.

B.4. Interfaz de usuario

A continuación se describe la interfaz gráfica de usuario (GUI) de la bomba de insulina simulada, que consta de dos paneles principales: el panel izquierdo muestra el estado actual del sistema y el panel derecho contiene controles y opciones de configuración.

B.4.1. Panel izquierdo: estado actual

En la parte izquierda de la pantalla se muestra el estado actual del sistema. En concreto:

- Se visualiza la glucosa actual en **mg/dL**, integrada dentro del símbolo de la insignia de Medtronic.

- Justo debajo, se muestra el tiempo transcurrido (en minutos) desde la última lectura de glucosa registrada.
- A la izquierda de esa información se muestra la **basal actual** en U/h que se está administrando.
- A la derecha se muestra la cantidad de **insulina activa** (en U) presente en el cuerpo, procedente de bolos o correcciones.
- En la parte superior de la interfaz se visualiza la **hora del dispositivo**, que puede no corresponder con la hora actual real.

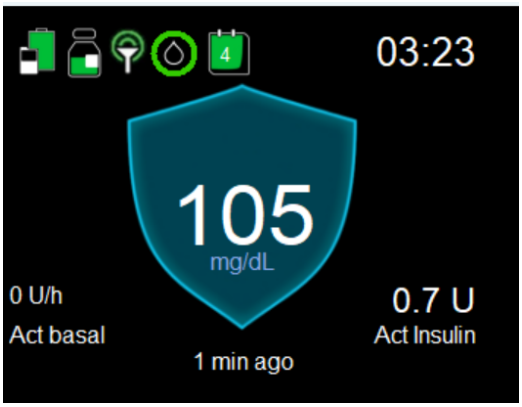


Figura B.1: Panel izquierdo de la interfaz de usuario de la bomba de insulina simulada.

B.4.2. Panel derecho: controles y configuración

De arriba a abajo, el panel derecho de la interfaz incluye los siguientes elementos:

- **Botón Synchronize Time**: al pulsarlo, la hora del dispositivo se sincroniza con la fecha y hora del último dato de glucosa registrado. Esta función es útil en simulaciones que utilizan datos en el pasado.
- **Caja de fecha y hora del dispositivo**: muestra el reloj actual del sistema, con precisión hasta segundos.
- **Selector de velocidad (Speed)**: permite modificar la velocidad a la que avanza el tiempo del sistema. Comienza en `x1` y puede aumentarse, lo cual es útil para ejecutar simulaciones aceleradas.
- **Botón Gráfico**: al pulsarlo, se muestra una gráfica con la evolución reciente de la glucosa.
- **Botón Comidas**: al seleccionarlo, se abre un panel para registrar una nueva ingesta. Este panel incluye:
  - Valor actual de glucosa.
  - Fila para introducir los **carbohidratos** (en gramos). Al escribir un valor, se calcula automáticamente la insulina necesaria.

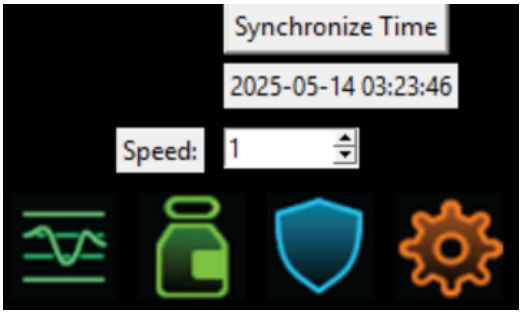


Figura B.2: Panel derecho de la interfaz de usuario de la bomba de insulina simulada.

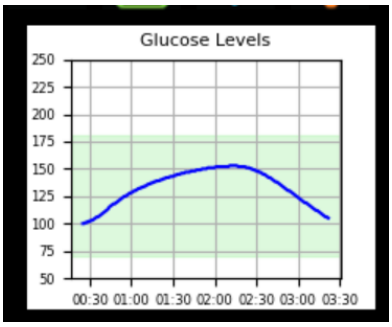


Figura B.3: Gráfico de evolución de glucosa en la bomba.

- Fila para introducir las **UGPs** (Unidades Grasa-Proteína). Al introducir un valor, se calcula la insulina correspondiente, aunque esta se administra de forma diferida.
- Fila de **ajuste automático**, que modifica la dosis en función del valor actual de glucosa y el objetivo. Si la glucosa es mayor al objetivo, se añade insulina extra. Si es menor, se resta insulina para evitar hipoglucemias.
- Fila con la **suma total de insulina** que se propone administrar.
- Botón **Deliver Bolus**, que al pulsarse añade un nuevo registro al archivo `inputs.json` con el bolo y la hora correspondiente.

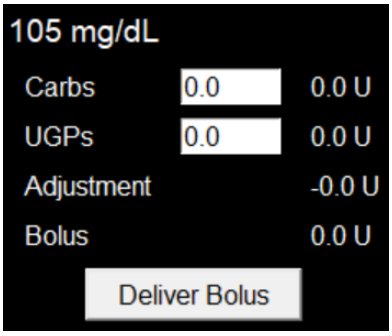


Figura B.4: Panel de comidas en la bomba de insulina.

- **Botón Events**: permite gestionar eventos que afectan al control de glucosa. Se incluyen:
  - **Exercise**: permite introducir múltiples eventos con campos `Start`, `Duration`, e `Intensity`.
  - **Alcohol ingestion**: permite definir eventos con `Start`, `Duration`, y `Amount`.

- **Mood**: incluye campos `Start`, `Duration`, y `Type` (por ejemplo, estrés, calma, etc.).
- **Illness**: incluye campos `Start`, `Duration`, y `Type` (enfermedad leve, fiebre, etc.).
- **Reservoir Changes**: permite configurar eventos relacionados con el cambio del reservorio de insulina. Se pueden indicar:
  - `Start`: fecha de cambio.
  - `Periodic`: si se activa, indica que el cambio se realiza periódicamente.
  - `Days`: número de días entre cambios (si es periódico).

  Todos los eventos pueden añadirse o eliminarse según necesidad, y se puede guardar la configuración mediante el botón **Save**.

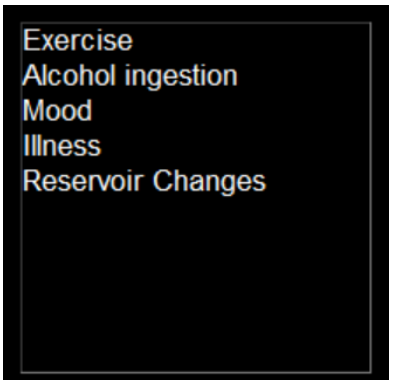


Figura B.5: Panel de eventos en la bomba de insulina.

- **Botón Config**: permite ajustar los parámetros de configuración del algoritmo de la bomba. Las opciones disponibles son:
  - **BG Target**: objetivo de glucosa (mg/dL).
  - **Carb Ratio**: cantidad de gramos de carbohidratos por unidad de insulina.
  - **Insulin Sensitivity Factor (ISF)**: número de mg/dL que se reduce por cada unidad de insulina.
  - **Reservoir Insulin Effect**: duración del efecto máximo de la insulina del reservorio (típicamente 2.5 días).
  - **Basal Max**: cantidad máxima de insulina basal que puede administrarse (U/h).

  Para BG Target, Carb Ratio e ISF, pueden configurarse múltiples tramos horarios utilizando columnas `From`, `To` y `Value`, y modificarse con los botones **Add** y **Remove**.

B.5. Uso general

Una vez configurado el sistema, la bomba ejecuta automáticamente el algoritmo que determina la administración de insulina basal, calcula bolos sugeridos, y responde a eventos registrados. Todos los cambios quedan reflejados en los archivos de configuración o en el archivo `inputs.json`, que también puede ser usado por la herramienta `DiabetesTool` para simulaciones y comparaciones.

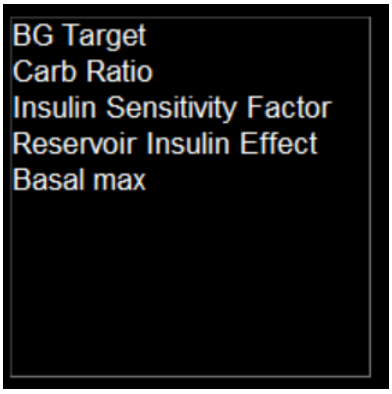


Figura B.6: Panel de configuración de la bomba de insulina.

B.6. Conclusiones

La bomba de insulina simulada desarrollada para este proyecto permite replicar el comportamiento de un sistema automatizado de administración de insulina, incorporando tanto funcionalidades de control en tiempo real como una interfaz accesible para el usuario. Ejecutada sobre una Raspberry Pi, esta herramienta permite validar el algoritmo propuesto en un entorno físico sencillo y económico.

Apéndice C

Preparación del dispositivo Raspberry Pi

C.1. Introducción

Para ejecutar el algoritmo de la bomba de insulina, se ha utilizado una Raspberry Pi Model B+ como plataforma hardware. Este dispositivo se seleccionó por su bajo coste, su tamaño reducido y su compatibilidad con herramientas de desarrollo en entorno Linux.

A continuación se detallan los pasos claves a seguir para su puesta en funcionamiento.

C.2. Instalación del sistema operativo

Se ha instalado la versión oficial del sistema operativo proporcionado por Raspberry Pi Foundation: **Raspberry Pi OS with desktop** basada en *Debian Bookworm*, con fecha de lanzamiento **2 de julio de 2024**, kernel versión **6.6.31**, Debian version 12 y arquitectura **32-bit**. La imagen utilizada fue:

- `2024-07-04-raspios-bookworm-armhf-full.img`
- Tamaño: 1.2 G

El procedimiento seguido fue:

1. Descarga de la imagen del sistema operativo desde el sitio oficial: https://www.raspberrypi.com/software/operating-systems/.
2. Utilización del programa **Win32 Disk Imager** en un PC con Windows para grabar la imagen en una tarjeta MicroSD, la cual será insertada posteriormente en la Raspberry Pi.
3. Inserción de la tarjeta MicroSD en la Raspberry Pi y conectar el dispositivo a la alimentación.
4. Conexión de una **pantalla táctil de 5 pulgadas** y de un **mini teclado táctil** para la interacción con el sistema.

Una vez iniciada, la Raspberry Pi ya cuenta con múltiples herramientas preinstaladas para desarrollo software (Python, editores, librerías, etc.), por lo que no fue necesaria la instalación de entornos adicionales. El algoritmo de la bomba fue desarrollado en un PC y transferido a la Raspberry Pi una vez finalizado.

C.3. Compartición de datos mediante Samba

Como parte de la arquitectura del sistema, el PC actúa como simulador o recolector de datos de glucosa (reales o simulados), mientras que la Raspberry Pi ejecuta el algoritmo de control. Para permitir la comunicación entre ambos dispositivos, se implementó un sistema de **carpeta compartida** utilizando Samba en la red local.

Samba es una implementación libre del protocolo de archivos compartidos de Microsoft Windows (antiguamente llamado SMB, renombrado posteriormente a CIFS) para sistemas de tipo UNIX. De esta forma, es posible que computadoras con GNU/Linux, Mac OS X o Unix en general se vean como servidores o actúen como clientes en redes de Windows. [41].

Se han seguido los pasos detallados en [42], y que se resume en las siguientes secciones:

C.3.1. Instalación de Samba en Raspberry Pi

Desde la terminal de la Raspberry Pi, se ejecuta el siguiente comando para instalar Samba:

```
sudo apt update && sudo apt install samba samba-common -y
```

C.3.2. Configuración del archivo `smb.conf`

A continuación, se edita el archivo de configuración de Samba:

```
sudo nano /etc/samba/smb.conf
```

Al final del archivo se añade el siguiente bloque para compartir la carpeta **/home/paloma** (paloma es el nombre de la carpeta del usuario del dispositivo):

```
[home]
   comment = Carpeta de inicio de la Raspberry Pi
   path = /home/paloma
   browseable = Yes
```

```
5     writeable = Yes
6     only guest = no
7     create mask = 0777
8     directory mask = 0777
9     public = no
```

C.3.3. Creación de usuario y contraseña de Samba

Para permitir el acceso a la carpeta desde Windows, se establece la contraseña del usuario `paloma` para Samba:

```
1 sudo smbpasswd -a paloma
```

C.3.4. Reinicio del servicio

Finalmente, se reinicia el servicio para aplicar los cambios:

```
1 sudo service smbd restart
```

C.3.5. Acceso desde Windows

Para acceder a la carpeta compartida desde un PC con Windows:

1. Se identifica la dirección IP de la Raspberry Pi usando el comando:

   ```
   1 ip addr
   ```

   En la Figura C.1 se muestra la dirección IP de la Raspberry Pi.

2. En el PC, en el explorador de archivos de Windows, se introduce la ruta de red del paso anterior en la barra de direcciones.

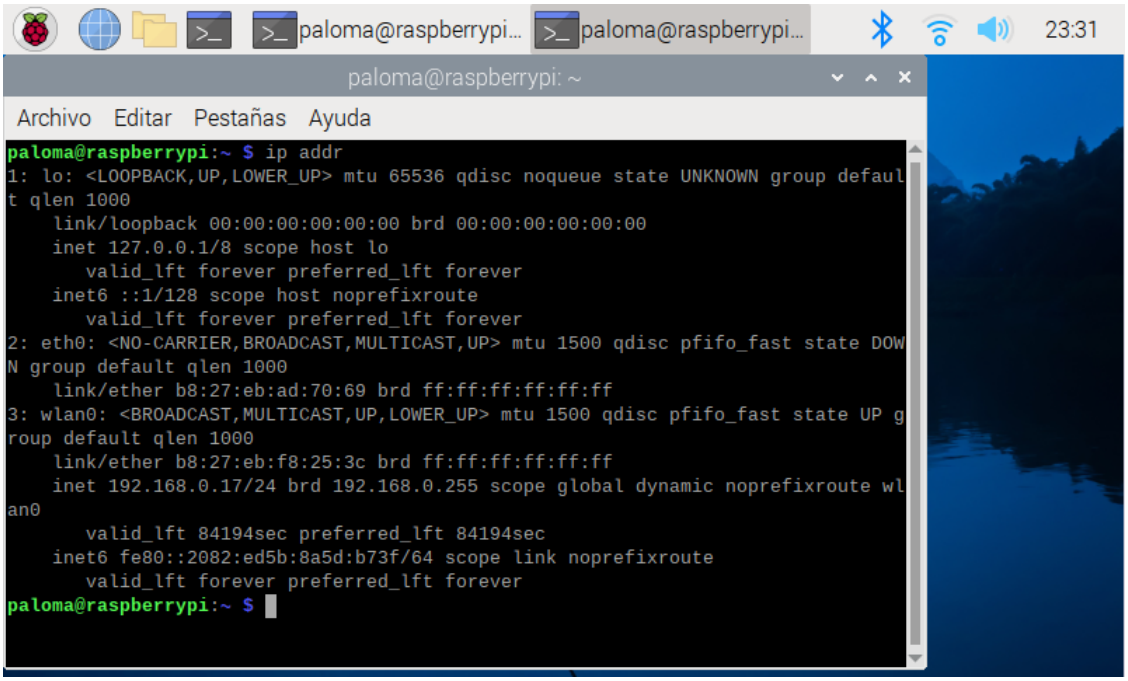


Figura C.1: comando ip addr: direacción 192.168.0.17 de la red wlan0.

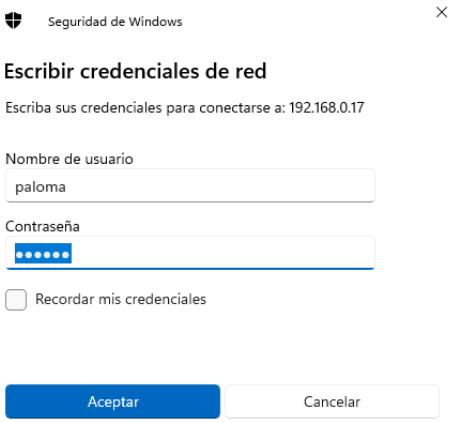


Figura C.2: Credenciales para conectar a la carpeta compartida.

3. Se introducen las credenciales del usuario `paloma` con la contraseña previamente configurada, como se muestra en la Figura C.2.

   En la Figura C.3 se muestra la carpeta compartida desde el PC, donde se pueden ver los archivos de la Raspberry Pi.

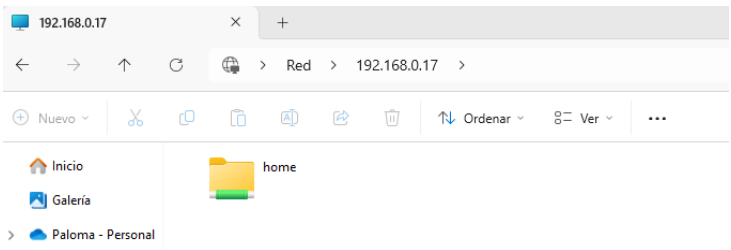


Figura C.3: Carpeta home de la Raspberry Pi.

Este sistema permite que tanto la Raspberry Pi como el PC accedan a una misma carpeta de datos, actuando como mecanismo de entrada/salida para el algoritmo de la bomba de insulina, tal y como se describe en el Capítulo 5.

Apéndice D

Código fuente de DiabetesTool

D.1. Estructura

Se incluye el código de `DiabetesTool` con comentarios en inglés. La organización del proyecto se basa en una estructura modular que facilita la reutilización y extensión del software. A continuación se describe la jerarquía de carpetas principal:

- `main.py`: Script principal que permite ejecutar la aplicación con interfaz gráfica.
- `requirements.txt`: Lista de dependencias necesarias para la ejecución.
- `app/`: Lógica de la interfaz gráfica (Tkinter).
  - `app.py`: Inicialización general de la aplicación.
  - `run_programs.py`: Lógica de ejecución de los distintos modos.
  - `input_section.py`: Gestión de inputs (rutas, fechas, modos).
  - `logging_handler.py`: Configuración de logs y mensajes.
- `carelink/`: Gestión de la descarga de datos desde CareLink.
  - `carelink_client2.py`: Cliente HTTP que interactúa con la API.
  - `json_carelink.py`: Procesamiento del JSON recibido.
  - `data.py`: Descarga y almacenamiento de datos.
  - `pump.py`: Extracción de eventos de insulina desde el JSON.
  - `logindata.json`: Archivo de autenticación generado por el usuario.
- `data/`: Ficheros de entrada y salida.
  - `data/`: Archivos `data.json` y `simple_data.json`.
  - `inputs/`: Archivos con las comidas y bolus (`inputs.json`).
  - `events/`: Archivos con eventos como ejercicio, alcohol, etc.
  - `params/`: Archivos con parámetros personalizados del modelo.
- `inputs/`: Carga y gestión de archivos `.json` de entrada.
  - `input_auxi.py`: Funciones auxiliares para cargar los inputs.
  - `json_data.py`: Lectura y escritura de archivos JSON.
- `model/`: Código del modelo de simulación de glucosa.

  - `cambridge_model.py`: Implementación del modelo de Hovorka.
  - `glucose.py`, `insulin.py`, `meal.py`, `events.py`: módulos del modelo.
- `programs/`: Scripts para ejecución directa de los modos.
  - `simulator_main.py`, `estimator_main.py`, `comparator_main.py`, etc.
- `simulator/`: Núcleo del simulador.
  - `simulator.py`: Avance temporal y simulación principal.
  - `solver.py`: Resolver numérico (Runge-Kutta).
  - `auxi.py`: Utilidades internas.
- `util/`: Funciones auxiliares comunes.
  - `glucose_plot.py`: Generación de gráficos.
  - `residuals.py`: Cálculo de residuos para validación.
  - `auxi.py`: Funciones genéricas de utilidad.

D.2. main.py

```
1 import tkinter as tk
2 from app.app import App
3
4 if __name__ == "__main__":
5     root = tk.Tk()
6     app = App(root)
7     root.mainloop()
```

Código D.1: main.py (DiabetesTool).

D.3. carelink_carepartner_api_login.py

Este script permite al usuario autenticarse en la API de CareLink. El usuario debe introducir su correo electrónico y contraseña, y el script generará un archivo `logindata.json` que contiene el token de acceso necesario para realizar solicitudes a la API.

```
1 ##################################################################
2 #
3 #  Carelink Carepartner API login
4 #
5 #  Description:
6 #
7 #    This program performs the login procedure to the Medtronic
    Carelink Cloud
8 #    service as implemeted in the Carlink Connect app. On successfull
    login it
```

```
9  #    creates a json file with the resulting login data. The file
       contains:
10 #    - access_token
11 #    - refresh_token
12 #    - scope
13 #    - client_id
14 #    - client_secret
15 #    - mag-identifier
16 #
17 #    Code adapted from: https://github.com/ondrej1024/carelink-python-
       client
18 #
19 ######################################################################
20 import base64
21 import hashlib
22 import json
23 import logging as log
24 import os
25 import re
26 from http.client import HTTPConnection
27 import sys
28 from time import sleep
29 import time
30 import requests
31
32 import curlify
33 import OpenSSL
34 from seleniumwire import webdriver
35
36 from util.auxi import random_b64_str, random_android_model,
       random_android_model, random_device_id, random_uuid
37 from carelink.json_carelink import read_token_file, write_datafile,
       logindata_file
38
39
40 # config
41 is_debug = False
42 # discovery_url = 'https://clcloud.minimed.eu/connect/carepartner/v6/
       discover/android/3.1'
43 discovery_url = 'https://clcloud.minimed.eu/connect/carepartner/v11/
       discover/android/3.2'
44
45 rsa_keysize = 2048
46
47 def setup_logging():
48   """Setup logging for the script."""
49   log.info("setting up logging")
50   HTTPConnection.debuglevel = 1
51   log.basicConfig()
52   log.getLogger().setLevel(log.DEBUG)
53   requests_log = log.getLogger("requests.packages.urllib3")
54   requests_log.setLevel(log.DEBUG)
55   requests_log.propagate = True
56
57
58 def create_csr(keypair, cn, ou, dc, o):
59   """Create a Certificate Signing Request (CSR) using the provided
       keypair and subject information."""
```

```
60    req = OpenSSL.crypto.X509Req()
61
62    #order is not checked
63    req.get_subject().CN = cn
64    req.get_subject().OU = ou
65    req.get_subject().DC = dc
66    req.get_subject().O = o
67
68    req.set_pubkey(keypair)
69    req.sign(keypair, 'sha256')
70
71    csr = OpenSSL.crypto.dump_certificate_request(OpenSSL.crypto.
       FILETYPE_PEM, req)
72    return csr
73
74  def reformat_csr(csr):
75    """Reformat the CSR by removing the header and footer, and re-
       encoding it with URL-safe base64."""
76    # remove footer & header, re-encode with url safe base64
77    csr = csr.decode()
78    csr = csr.replace("\n", "")
79    csr = csr.replace("-----BEGIN CERTIFICATE REQUEST-----", "")
80    csr = csr.replace("-----END CERTIFICATE REQUEST-----", "")
81
82    csr_raw = base64.b64decode(csr.encode())
83    csr = base64.urlsafe_b64encode(csr_raw).decode()
84    return csr
85
86  def do_captcha(url, redirect_url):
87    """Open a Firefox instance and handle the captcha login process."""
88    log.info("Opening Firefox instance...")
89    log.info("Warning: you may need to close Firefox if it's already
       running or nothing happens!")
90    try:
91
92      log.basicConfig(level=log.DEBUG)
93      driver = webdriver.Firefox()
94      log.info("Opened Firefox instance")
95      driver.get(url)
96      log.info("Opened url")
97
98      while True:
99        for request in driver.requests:
100         if request.response:
101           if request.response.status_code == 302:
102             if "location" in request.response.headers:
103               location = request.response.headers["location"]
104               if redirect_url in location:
105                 code = re.search(r"code=(.*)&", location).group(1)
106                 state = re.search(r"state=(.*)", location).group(1)
107                 driver.quit()
108                 return (code, state)
109       sleep(0.1)
110   except Exception as e:
111     log.error(f"Failed to open Firefox: {e}")
112     raise
113
114 def resolve_endpoint_config(discovery_url, is_us_region=False):
```

```
115   """Resolve the endpoint configuration from the discovery URL."""
116   discover_resp = json.loads(requests.get(discovery_url).text)
117   sso_url = None
118
119   for c in discover_resp["CP"]:
120     if c['region'].lower() == "us" and is_us_region:
121       sso_url = c['SSOConfiguration']
122     elif c['region'].lower() == "eu" and not is_us_region:
123       sso_url = c['SSOConfiguration']
124
125   if sso_url is None:
126     raise Exception("Could not get SSO config url")
127
128   sso_config = json.loads(requests.get(sso_url).text)
129   api_base_url = f"https://{sso_config['server']['hostname']}:{
      sso_config['server']['port']}/{sso_config['server']['prefix']}"
130   return sso_config, api_base_url
131
132
133 def do_login(endpoint_config):
134   """Perform the login process using the provided endpoint
      configuration."""
135   sso_config, api_base_url = endpoint_config
136   # step 1 initialize
137   log.info(f"initializing client")
138   data = {
139     'client_id': sso_config['oauth']['client']['client_ids'][0]['
      client_id'],
140     "nonce" :  random_uuid()
141   }
142   headers = {
143     'device-id': base64.b64encode(random_device_id().encode()).decode
      () # this is not used elsewhere?
144   }
145   client_init_url = api_base_url + sso_config["mag"]["system_endpoints
      "]["client_credential_init_endpoint_path"]
146   client_init_req = requests.post(client_init_url, data=data, headers=
      headers)
147   client_init_response = json.loads(client_init_req.text)
148
149   # step 2 authorize
150   log.info(f"authorizing client")
151   client_code_verifier = base64.urlsafe_b64encode(os.urandom(40)).
      decode('utf-8')
152   client_code_verifier = re.sub('[^a-zA-Z0-9]+', '',
      client_code_verifier)
153   client_code_challange = hashlib.sha256(client_code_verifier.encode('
      utf-8')).digest()
154   client_code_challange = base64.urlsafe_b64encode(
      client_code_challange).decode('utf-8')
155   client_code_challange = client_code_challange.replace('=', '')
156
157   client_state = random_b64_str(22)
158   auth_params = {
159     'client_id': client_init_response["client_id"],
160     'response_type' : 'code',
161     'display' : 'social_login',
162     'scope': sso_config["oauth"]["client"]["client_ids"][0]['scope'],
```

```
    'redirect_uri': sso_config["oauth"]["client"]["client_ids"][0]['redirect_uri'],
    'code_challenge' : client_code_challange,
    'code_challenge_method': 'S256',
    'state': client_state
  }
  authorize_url = api_base_url + sso_config["oauth"]["system_endpoints"]["authorization_endpoint_path"]
  providers = json.loads(requests.get(authorize_url, params=auth_params).text) # this will redirect
  captcha_url = providers["providers"][0]["provider"]["auth_url"]

  # step 3 captcha login and consent
  log.info(f"captcha login and consent")
  log.info(f"captcha url: {captcha_url}")
  captcha_code, captcha_sso_state = do_captcha(captcha_url, sso_config["oauth"]["client"]["client_ids"][0]['redirect_uri'])
  log.info(f"sso state after captcha: {captcha_sso_state}")

  # step 4 registraton
  log.info(f"registering device")
  register_device_id = random_device_id()
  client_auth_str = f"{client_init_response['client_id']}:{client_init_response['client_secret']}"

  android_model = random_android_model()
  android_model_safe = re.sub(r"[^a-zA-Z0-9]", "", android_model)
  keypair = OpenSSL.crypto.PKey()
  keypair.generate_key(OpenSSL.crypto.TYPE_RSA, rsa_keysize)
  csr = create_csr(keypair, "socialLogin", register_device_id, android_model_safe, sso_config["oauth"]["client"]["organization"])

  reg_headers = {
    'device-name': base64.b64encode(android_model.encode()).decode(),
    'authorization' : f"Bearer {captcha_code}",
    'cert-format': 'pem',
    'client-authorization': "Basic " + base64.b64encode(client_auth_str.encode()).decode(),
    'create-session': 'true',
    'code-verifier': client_code_verifier,
    'device-id': base64.b64encode(register_device_id.encode()).decode(),
    "redirect-uri": sso_config["oauth"]["client"]["client_ids"][0]['redirect_uri']
  }
  csr = reformat_csr(csr)
  reg_url = api_base_url + sso_config["mag"]["system_endpoints"]["device_register_endpoint_path"]
  reg_req = requests.post(reg_url, headers=reg_headers, data=csr)
  if reg_req.status_code != 200:
    log.info(f"\n\n{curlify.to_curl(reg_req.request)}")
    raise Exception(f'Could not register: {json.loads(reg_req.text)["error_description"]}')

  # step 5 token
  log.info(f"getting token data")
  token_req_url = api_base_url + sso_config["oauth"]["system_endpoints"]["token_endpoint_path"]
```

```
209   token_req_data = {
210     "assertion" : reg_req.headers["id-token"],
211     "client_id" : client_init_response['client_id'],
212     "client_secret" : client_init_response['client_secret'],
213     'scope': sso_config["oauth"]["client"]["client_ids"][0]['scope'],
214     "grant_type" : reg_req.headers["id-token-type"]
215   }
216   token_req = requests.post(token_req_url, headers={"mag-identifier" :
        reg_req.headers["mag-identifier"]}, data=token_req_data)
217   if token_req.status_code != 200:
218     log.info(f"\n\n{curlify.to_curl(token_req.request)}")
219     raise Exception("Could not get token data")
220
221   token_data = json.loads(token_req.text)
222   log.info(f"got token data from server")
223
224   token_data["client_id"] = token_req_data["client_id"]
225   token_data["client_secret"] = token_req_data["client_secret"]
226   del token_data["expires_in"]
227   del token_data["token_type"]
228   token_data["mag-identifier"] = reg_req.headers["mag-identifier"]
229
230   write_datafile(token_data, logindata_file)
231   return token_data
232
233 def generate_token():
234   """Generate a token for the Carelink API."""
235   log.info(f"Generate token data")
236   #sys.stderr = sys.__stderr__
237   if is_debug:
238     log.info(f"debug mode enabled")
239     setup_logging()
240
241   log.info(f"Reading token data file")
242   token_data = read_token_file(logindata_file)
243
244   if token_data == None:
245     log.info(f"performing login...")
246     log.info(f"Requesting endpoint config")
247     endpoint_config = resolve_endpoint_config(discovery_url)
248     log.info(f"Endpoint config: {endpoint_config}")
249     token_data = do_login(endpoint_config)
250   else:
251     log.info(f"token data file already exists")
252
253 def main():
254   """Main function to generate a token for the Carelink API."""
255   log.info(f"Generate token data")
256   sys.stderr = sys.__stderr__
257   if is_debug:
258     log.info(f"debug mode enabled")
259     setup_logging()
260
261   log.info(f"Reading token data file")
262   token_data = read_token_file(logindata_file)
263
264   if token_data == None:
265     log.info(f"Requesting endpoint config")
```

```
266     endpoint_config = resolve_endpoint_config(discovery_url)
267     log.info(f"Endpoint config: {endpoint_config}")
268     token_data = do_login(endpoint_config)
269   else:
270     log.info(f"token data file already exists")
271
272
273 if __name__ == "__main__":
274     main()
275
276
```

Código D.2: carelink_carepartner_api_login.py (DiabetesTool).

D.4. app

D.4.1. app.py

```
1  import sys
2  import tkinter as tk
3  from tkinter import ttk
4  import logging as log
5  from app.logging_handler import TextHandler
6  from app.input_section   import show_files_panel, show_inputs,
       show_dinamic_simulator_inputs
7  from inputs.json_data import load_json_data
8  import os
9
10 sys.stderr = open(os.devnull, 'w')  # Redirect stderr to avoid
       printing unwanted errors on the console
11 log.getLogger('matplotlib').setLevel(log.ERROR)  # Suppress matplotlib
       debug logs
12
13 log.basicConfig(format='[%(asctime)s:%(levelname)s] %(message)s',
       datefmt='%Y-%m-%d %H:%M:%S', level=log.INFO)
14
15 class App:
16     def __init__(self, root):
17         self.root = root
18         self.root.title("Diabetes Tool")
19         self.root.geometry("1200x800")
20         self.root.state('zoomed')
21         self.root.protocol("WM_DELETE_WINDOW", self.on_closing)
22
23         self._setup_frames()
24         self._setup_logging()
25         self.file_inputs = "data/inputs/inputs.json"
26         self.create_top_buttons()
27         show_dinamic_simulator_inputs(self)
28         show_files_panel(self)
29         self.data=load_json_data()
30
31     def _setup_frames(self):
```

```
32          """Create layout frames for the GUI."""
33          self.top_frame    = ttk.Frame(self.root)
34          self.left_frame   = ttk.Frame(self.root)
35          self.right_frame  = ttk.Frame(self.root)
36          self.center_frame = ttk.Frame(self.root)
37
38          # Arrange the frames
39          self.top_frame.pack(   side=tk.TOP,                fill=tk.X
     )
40          self.left_frame.pack(  side=tk.LEFT,               fill=tk.Y
     )
41          self.right_frame.pack( side=tk.RIGHT,              fill=tk.Y
     )
42          self.center_frame.pack(side=tk.LEFT,  expand=True, fill=tk.
    BOTH)
43
44          # Paned window for terminal and plot
45          self.paned_window = ttk.PanedWindow(self.center_frame, orient=
    tk.VERTICAL)
46          self.paned_window.pack(expand=True, fill=tk.BOTH)
47
48          # Terminal and plot areas
49          self.terminal_frame = ttk.Frame(   self.paned_window, height
    =400)
50          self.plot_frame     = ttk.Frame(   self.paned_window, height
    =400)
51          self.paned_window.add(self.terminal_frame, weight=1)
52          self.paned_window.add(self.plot_frame,     weight=1)
53
54          # Create a notebook for tabs
55          self.notebook = ttk.Notebook(self.plot_frame, height=400)
56          self.notebook.pack(fill=tk.BOTH, expand=True)
57
58      def _setup_logging(self):
59          """Set up logging to output to terminal frame."""
60          self.terminal_output = tk.Text(self.terminal_frame, wrap=tk.
    WORD, state='disabled', background="black", foreground="white")
61          self.terminal_output.pack(expand=True, fill=tk.BOTH, padx=10,
    pady=10)
62          text_handler = TextHandler(self.terminal_output)
63          log.getLogger().addHandler(text_handler)
64          log.getLogger().setLevel(log.INFO)
65
66      def create_top_buttons(self):
67          """Create buttons for top menu options."""
68          ttk.Button(self.top_frame, text="Dinamic Simulator",  command=
    lambda: show_inputs(self,"DSimulator") ).pack(side=tk.LEFT,  padx
    =5, pady=5)
69          ttk.Button(self.top_frame, text="Static Simulator",   command=
    lambda: show_inputs(self,"SSimulator") ).pack(side=tk.LEFT,  padx
    =5, pady=5)
70          ttk.Button(self.top_frame, text="Estimator",  command=lambda:
    show_inputs(self,"Estimator") ).pack(side=tk.LEFT,  padx=5, pady
    =5)
71          ttk.Button(self.top_frame, text="Downloader", command=lambda:
    show_inputs(self,"Downloader")).pack(side=tk.LEFT,  padx=5, pady
    =5)
```

```
72          ttk.Button(self.top_frame, text="Comparator", command=lambda:
    show_inputs(self,"Comparator")).pack(side=tk.LEFT,  padx=5, pady
    =5)
73          ttk.Button(self.top_frame, text="Events",     command=lambda:
    show_inputs(self,"Events")    ).pack(side=tk.RIGHT, padx=5, pady
    =5)
74          ttk.Button(self.top_frame, text="Parameters", command=lambda:
    show_inputs(self,"Parameters")).pack(side=tk.RIGHT, padx=5, pady
    =5)
75          ttk.Button(self.top_frame, text="Inputs",     command=lambda:
    show_inputs(self,"Inputs"    )).pack(side=tk.RIGHT, padx=5, pady
    =5)
76          ttk.Button(self.top_frame, text="Files",      command=lambda:
    show_inputs(self,"Files")     ).pack(side=tk.RIGHT, padx=5, pady
    =5)
77
78      def on_closing(self):
79          """Method called when the window is closed."""
80          log.info("Closing the application...")
81          self.root.quit()
82          self.root.destroy()
83          sys.exit()
84
85      def clear_left_frame(self):
86          """Remove all widgets from the left frame."""
87          for widget in self.left_frame.winfo_children():
88              widget.destroy()
89      def clear_right_frame(self):
90          """Remove all widgets from the right frame."""
91          for widget in self.right_frame.winfo_children():
92              widget.destroy()
93
94      def create_plot_area(self):
95          """Clear and recreate the plot area frame."""
96          for widget in self.plot_frame.winfo_children():
97              widget.destroy()
98          # Repack the plot_frame (ensure it fills the space)
99          self.plot_frame.pack(fill=tk.BOTH, expand=True)
100         self.paned_window.add(self.plot_frame, weight = 1)
101
102     def create_notebook_area(self):
103         """Clear the plot area and initialize a new tabbed notebook.
    """
104         for widget in self.plot_frame.winfo_children():
105             widget.destroy()
106         # Create the notebook in the plot_frame
107         self.notebook = ttk.Notebook(self.plot_frame, height=600)
108         self.notebook.pack(fill=tk.BOTH, expand=True)
```

Código D.3: app.py (DiabetesTool).

D.4.2. input_section.py

```
1 from datetime import datetime, timedelta
2 import logging as log
```

```
3  import os
4  import tkinter as tk
5  from tkinter import ttk
6  from app.run_programs import run_comparator, run_downloader,
       run_estimator, run_simulator
7  from inputs.input_auxi import DEFAULT_EVENT, DEFAULT_PARAMS,
       FAULT_INPUTS, INPUTS, create_style
8  from inputs.json_data import load_json_data, load_params,
       save_json_data, load_events_from_json
9  from util.auxi import get_current_time
10 from inputs.input_auxi import KEY_MAP, MEAL_COLUMN, BOLU_COLUMN,
       BASA_COLUMN, EXER_COLUMN, ALCO_COLUMN, RESE_COLUMN, MOOD_COLUMN,
       ILLN_COLUMN, EVENTS
11 import pyperclip # type: ignore
12
13
14 def show_inputs(app, name):
15     """Show inputs for the selected module."""
16     if   name == "DSimulator":
17         show_dinamic_simulator_inputs(app)
18     elif   name == "SSimulator":
19         show_static_simulator_inputs(app)
20     elif name == "Estimator":
21         show_estimator_inputs(app)
22     elif name == "Downloader":
23         show_downloader_inputs(app)
24     elif name == "Comparator":
25         show_comparator_inputs(app)
26     elif name == "Events":
27         show_events_inputs(app)
28     elif name == "Parameters":
29         show_parameters_inputs(app)
30     elif name == "Inputs":
31         show_inputs_inputs(app)
32     elif name == "Files":
33         show_files_panel(app)
34
35
36 def show_files_panel(app):
37     """Displays the file management panel with treeview and option to
       add new input files."""
38
39     app.clear_right_frame()
40
41     # Frame for input field and button
42     add_file_frame = ttk.Frame(app.right_frame)
43     add_file_frame.pack(fill=tk.X, padx=10, pady=5)
44
45     # Variable for filename
46     app.new_filename_var = tk.StringVar()
47     ttk.Label(add_file_frame, text="data/").pack(side=tk.LEFT,padx=10,
       pady=5)
48
49
50     # Entry field for filename
51     ttk.Entry(add_file_frame, textvariable=app.new_filename_var, width
       =30).pack(side=tk.LEFT, padx=5)
52
```

```
53      # Button to add a file
54      ttk.Button(add_file_frame, text="Add Input", command=lambda:
        add_new_input_file(app)).pack(side=tk.LEFT, padx=5)
55
56      # Treeview to explore files
57      app.tree = ttk.Treeview(app.right_frame)
58      app.tree.pack(expand=True, fill="both")
59
60      # Define columns and headers
61      app.tree['columns'] = ('Name')
62      app.tree.column('#0',     width=69,     anchor="w")
63      app.tree.heading('#0',    text='Path')
64      app.tree.heading('Name', text='Name')
65
66      # Populate Treeview with JSON files in data and inputs
67      populate_file_tree(app)
68
69      # Bind Treeview selection event to a method
70      app.tree.bind("<<TreeviewSelect>>", lambda event: on_tree_select(
        app, event))
71
72
73  def add_new_input_file(app):
74      """Creates a new JSON file in the 'inputs' folder with the name
        provided in the input field."""
75
76      # Get and clean the filename
77      filename = app.new_filename_var.get().strip()
78
79      # Validate filename is not empty
80      if not filename:
81          log.warning("Error", "Please enter a filename.")
82          return
83
84      # Ensure filename ends with .json
85      if not filename.endswith('.json'):
86          filename += '.json'
87
88      file_path = os.path.join('data', filename)
89
90      # Check if file already exists
91      if os.path.exists(file_path):
92          log.warning("Error", f"The file '{filename}' already exists.")
93          return
94
95      # Create new empty JSON file
96      try:
97          # Define default structures depending on folder
98          if 'inputs' in file_path:
99              default_content = FAULT_INPUTS
100         elif 'params' in file_path:
101             default_content = DEFAULT_PARAMS
102         elif 'events' in file_path:
103             default_content = DEFAULT_EVENT
104         else:
105             default_content = {}
106         save_json_data(default_content, file_path)
107         app.new_filename_var.set('')
```

```
108     except Exception as e:
109         log.warning("Error", f"An error occurred: {e}")
110 
111 
112 def populate_file_tree(app):
113     """Populates the Treeview with JSON files from 'data' and 'inputs'."""
114     app.tree.delete(*app.tree.get_children())  # Clear treeview before populating
115 
116     for folder in ['data', 'inputs', 'events', 'params']:
117         node = app.tree.insert('', 'end', text=folder, values=(folder, 'Folder'))
118         path = os.path.join('data', folder).replace('\\', '/')
119         for file in os.listdir(path):
120             if file.endswith(".json"):
121                 app.tree.insert(node, 'end', text=os.path.join(path, file).replace('\\', '/'), values=(file, 'JSON File'))
122 
123 
124 def on_tree_select(app, event):
125     """Handles the selection event in the Treeview."""
126     selected_item = app.tree.selection()
127     if selected_item:
128         item_text = app.tree.item(selected_item, "text")
129         if os.path.isfile(item_text):
130             pyperclip.copy(item_text)
131             log.info(f"File path copied to clipboard: {item_text}")
132 
133 
134 def add_entry_row(label_text, var, default_value, frame):
135     """ Helper function to pack labels and entries in the same row """
136     var.set(default_value)
137     ttk.Label(frame, text=label_text).pack(anchor="w", padx=5, pady=(2, 1))
138     entry = ttk.Entry(frame, textvariable=var, width=20)
139     entry.pack(anchor="w", padx=5, pady=(0, 5))
140     return entry
141 
142 def show_dinamic_simulator_inputs(app):
143     """Show inputs for dynamic simulator"""
144     rts = True # Real time system True
145     show_simulator_inputs(app, rts)
146 
147 def show_static_simulator_inputs(app):
148     """Show inputs for the static simulator"""
149     rts = False
150     show_simulator_inputs(app, rts)
151 
152 def show_simulator_inputs(app,rts):
153     """Show inputs for the simulator"""
154     app.rts = rts
155     app.clear_left_frame()
156 
157     app.create_plot_area()
158 
159     # Frame to hold inputs in rows
160     input_frame = ttk.Frame(app.left_frame)
```

```
161     input_frame.pack(fill=tk.X, padx=5, pady=5)
162 
163     # Define default values and labels
164     app.sim_start_time = tk.StringVar()
165     app.sim_duration   = tk.StringVar()
166     app.sim_parameter  = tk.StringVar()
167     app.sim_vx         = tk.StringVar()
168     app.sim_vm         = tk.StringVar()
169     app.sim_iterations = tk.StringVar()
170 
171     add_entry_row("Start Time",     app.sim_start_time,
    get_current_time(), input_frame)
172     add_entry_row("Duration (min)", app.sim_duration,   "300",
          input_frame)
173 
174 
175     # File paths for Input, Events, and Params
176     app.sim_pump_file   = tk.StringVar()
177     app.sim_input_file  = tk.StringVar()
178     app.sim_events_file = tk.StringVar()
179     app.sim_params_file = tk.StringVar()
180     app.sim_data_file   = tk.StringVar()
181 
182     if rts:
183         ttk.Separator(input_frame, orient="horizontal").pack(fill="x",
     padx=5, pady=5)
184         ttk.Label(input_frame, text="Glucose from:").pack(padx=5, pady
    =5)
185 
186         # Add complementary buttons (Simulation and Pump)
187         app.glucose_source = tk.StringVar(value="Simulation")  #
    Default selected option
188 
189         sim_button = ttk.Radiobutton(
190             input_frame, text="Simulation", variable=app.
    glucose_source, value="Simulation",
191             command=lambda: toggle_entry(app)
192         )
193         pumpRT_button = ttk.Radiobutton(
194             input_frame, text="Pump in real time", variable=app.
    glucose_source, value="PumpRT",
195             command=lambda: toggle_entry(app)
196         )
197         pumpOD_button = ttk.Radiobutton(
198             input_frame, text="Pump old data", variable=app.
    glucose_source, value="PumpOD",
199             command=lambda: toggle_entry(app)
200         )
201 
202         sim_button.pack(   anchor=tk.W, padx=10, pady=2)
203         pumpRT_button.pack(anchor=tk.W, padx=10, pady=2)
204         pumpOD_button.pack(anchor=tk.W, padx=10, pady=2)
205 
206         # Add Speed entry
207         ttk.Separator(input_frame, orient="horizontal").pack(fill="x",
     padx=5, pady=5)
208         ttk.Label(input_frame, text="Speed").pack(padx=5, pady=5)
209 
```

```
210         speed_frame = ttk.Frame(input_frame)
211         speed_frame.pack(fill=tk.X, padx=10, pady=5)
212 
213         ttk.Label(speed_frame, text="x").pack(side=tk.LEFT, padx=5)
214 
215         app.sim_speed = tk.StringVar(value="1")
216         app.speed_entry = ttk.Entry(speed_frame, textvariable=app.
    sim_speed, width=10)
217         app.speed_entry.pack(side=tk.LEFT, padx=5)
218 
219 
220     ttk.Separator(input_frame, orient="horizontal").pack(fill="x",
    padx=5, pady=5)
221     ttk.Label(input_frame, text="Files (Optional)").pack(padx=5, pady
    =5)
222 
223     if rts:
224         app.sim_pump_file_entry = add_entry_row("Pump data",
              app.sim_pump_file,   "", input_frame)
225     app.sim_input_file_entry    = add_entry_row("Inputs (inputs/inputs
    .json)", app.sim_input_file,  "", input_frame)
226     app.sim_events_file_entry   = add_entry_row("Events (events/events
    .json)", app.sim_events_file, "", input_frame)
227     app.sim_params_file_entry   = add_entry_row("Params (params/params
    .json)", app.sim_params_file, "", input_frame)
228 
229     if rts:
230         add_entry_row("Output (data/data.json)", app.sim_data_file, ""
    , input_frame)
231         # Initially disable the speed entry if Pump is selected
232         toggle_entry(app)
233     else:
234         # Separator line for visual clarity
235         ttk.Separator(input_frame, orient="horizontal").pack(fill="x",
     padx=5, pady=5)
236         ttk.Label(input_frame, text="Multiple simulations (Optional)")
    .pack(padx=5, pady=5)
237 
238         add_entry_row("Parameter to Vary",     app.sim_parameter,  "",
      input_frame)
239         add_entry_row("Maximum Value",         app.sim_vx,         "",
      input_frame)
240         add_entry_row("Minimum Value",         app.sim_vm,         "",
      input_frame)
241         add_entry_row("Number of Simulations", app.sim_iterations, "",
      input_frame)
242 
243     # Run button
244     ttk.Button(app.left_frame, text="Run", command=lambda:
    run_simulator(app)).pack(padx=5, pady=10)
245 
246 def toggle_entry(app):
247     """Toggle the availability of entry fields depending on selected
    input source."""
248     if app.glucose_source.get() == "PumpRT":
249         app.speed_entry.state(["disabled"])
250     else:
251         app.speed_entry.state(["!disabled"])
```

```
252
253     if app.glucose_source.get() == "Simulation":
254         app.sim_pump_file_entry.state([   "disabled"])
255         app.sim_input_file_entry.state( ["!disabled"])
256         app.sim_events_file_entry.state(["!disabled"])
257         app.sim_params_file_entry.state(["!disabled"])
258     else:
259         app.sim_input_file_entry.state( ["disabled"])
260         app.sim_events_file_entry.state(["disabled"])
261         app.sim_params_file_entry.state(["disabled"])
262         if app.glucose_source.get() == "PumpOD":
263             app.sim_pump_file_entry.state([ "!disabled"])
264         else:
265             app.sim_pump_file_entry.state([ "disabled"])
266
267 def toggle_entry_compare(app):
268     """Toggle the availability of entry fields depending on selected
    input source."""
269     if app.compare.get() == "Glucose":
270         app.comp_ref_ins_file_entry.state(["disabled"])
271         app.comp_ins_file_entry.state(["disabled"])
272         app.comp_input_file_entry.state(["!disabled"])
273     else:
274         app.comp_ref_ins_file_entry.state(["!disabled"])
275         app.comp_ins_file_entry.state(["!disabled"])
276         app.comp_input_file_entry.state(["disabled"])
277
278
279 def show_estimator_inputs(app):
280     """Show inputs for the estimator"""
281     app.clear_left_frame()
282     app.create_notebook_area()
283
284     # Frame to hold inputs in rows
285     input_frame = ttk.Frame(app.left_frame)
286     input_frame.pack(fill=tk.X, padx=5, pady=5)
287
288
289     # Define default values and labels
290     app.est_start_time = tk.StringVar()
291     app.est_duration   = tk.StringVar()
292     app.est_parameter  = tk.StringVar()
293     app.est_vx         = tk.StringVar()
294     app.est_vm         = tk.StringVar()
295     app.est_iterations = tk.StringVar()
296     yesterday = (datetime.now() - timedelta(days=1)).strftime("%Y-%m-%
    dT%H:%M:%S")
297
298     add_entry_row("Start Date ",            app.est_start_time,
    yesterday, input_frame)
299     add_entry_row("Duration (min)",         app.est_duration,   "300",
        input_frame)
300     add_entry_row("Parameter to Estimate",  app.est_parameter , "SIT",
        input_frame)
301     add_entry_row("Maximum Value (vx)",     app.est_vx,         "0.04"
    ,   input_frame)
302     add_entry_row("Minimum Value (vm)",     app.est_vm,         "0.02"
    ,   input_frame)
```

```
303     add_entry_row("Number of Iterations",   app.est_iterations, "5",
            input_frame)
304
305     ttk.Separator(input_frame, orient="horizontal").pack(fill="x",
        padx=5, pady=5)
306     ttk.Label(input_frame, text="Files (Optional)").pack(padx=5, pady
        =5)
307
308     # File paths (data file, input, events, params)
309     app.est_data_file   = tk.StringVar()
310     app.est_input_file  = tk.StringVar()
311     app.est_events_file = tk.StringVar()
312     app.est_params_file = tk.StringVar()
313
314     add_entry_row("Data   (Downloaded)",          app.est_data_file,
        "", input_frame)
315     add_entry_row("Inputs (Downloaded)",          app.est_input_file,
        "", input_frame)
316     add_entry_row("Events (events/events.json)", app.est_events_file,
        "", input_frame)
317     add_entry_row("Params (params/params.json)", app.est_params_file,
        "", input_frame)
318
319     # Run button
320     ttk.Button(app.left_frame, text="Run", command=lambda:
        run_estimator(app)).pack(padx=5, pady=10)
321
322
323 def show_comparator_inputs(app):
324     """Show inputs for the comparator"""
325     app.clear_left_frame()
326
327     app.create_plot_area()
328
329     # Define StringVar with default value (current date and time)
330     yesterday = (datetime.now() - timedelta(days=1)).strftime("%Y-%m-%
        dT%H:%M:%S")
331
332     app.comp_start_time    = tk.StringVar()
333     app.comp_duration      = tk.StringVar()
334     app.comp_input_file    = tk.StringVar()
335     app.comp_ins_file      = tk.StringVar()
336     app.comp_ref_ins_file  = tk.StringVar()
337
338
339     add_entry_row("Start Date",           app.comp_start_time,
        yesterday, app.left_frame)
340     add_entry_row("Duration (min)",       app.comp_duration,     "",
            app.left_frame)
341
342
343     ttk.Separator(app.left_frame, orient="horizontal").pack(fill="x",
        padx=5, pady=5)
344     ttk.Label(app.left_frame, text="Compare:").pack(padx=5, pady=5)
345
346     app.compare = tk.StringVar(value="Glucose")
347     glucose_button = ttk.Radiobutton(
```

```
348         app.left_frame, text="Glucose", variable=app.compare, value="
    Glucose",
349         command=lambda: toggle_entry_compare(app)
350     )
351     insulin_button = ttk.Radiobutton(
352         app.left_frame, text="Insulin", variable=app.compare, value="
    Insulin",
353         command=lambda: toggle_entry_compare(app)
354     )
355
356     glucose_button.pack(anchor=tk.W, padx=10, pady=2)
357     insulin_button.pack(anchor=tk.W, padx=10, pady=2)
358
359     ttk.Separator(app.left_frame, orient="horizontal").pack(fill="x",
    padx=5, pady=5)
360     ttk.Label(app.left_frame, text="Glucose").pack(padx=5, pady=5)
361
362     app.comp_input_file_entry = add_entry_row("Data File (Optional)",
    app.comp_input_file,   "",       app.left_frame)
363
364     ttk.Separator(app.left_frame, orient="horizontal").pack(fill="x",
    padx=5, pady=5)
365     ttk.Label(app.left_frame, text="Insulin").pack(padx=5, pady=5)
366
367
368     app.comp_ref_ins_file_entry = add_entry_row("Reference Input File"
    , app.comp_ref_ins_file,   "", app.left_frame)
369     app.comp_ins_file_entry     = add_entry_row("Input File",
      app.comp_ins_file,       "", app.left_frame)
370
371     ttk.Separator(app.left_frame, orient="horizontal").pack(fill="x",
    padx=5, pady=5)
372     toggle_entry_compare(app)
373     ttk.Button(app.left_frame, text="Run", command=lambda:
    run_comparator(app)).pack(padx=10, pady=20)
374
375
376 def show_downloader_inputs(app):
377     """Show inputs for the downloader"""
378     app.clear_left_frame()
379
380     app.create_plot_area()
381
382
383     app.dl_repeat_times  = tk.StringVar()
384     app.dl_wait_time  = tk.StringVar()
385     add_entry_row("Repeat Times",    app.dl_repeat_times, "1", app.
    left_frame)
386     add_entry_row("Wait Time (min)", app.dl_wait_time,    "5", app.
    left_frame)
387
388     # Save Data Checkbox
389     app.dl_save_data = tk.BooleanVar()
390     ttk.Checkbutton(app.left_frame, text="Save Data", variable=app.
    dl_save_data).pack(padx=10, pady=5)
391
392     # Verbose Checkbox
393     app.dl_verbose = tk.BooleanVar()
```

```
394     ttk.Checkbutton(app.left_frame, text="Verbose Mode", variable=app.
    dl_verbose).pack(padx=10, pady=5)
395 
396     # Run Button
397     ttk.Button(app.left_frame, text="Run", command=lambda:
    run_downloader(app)).pack(padx=10, pady=20)
398 
399 
400 def show_events_inputs(app, file="data/events/events.json"):
401     """Display Events input section with exercise, alcohol, reservoir,
     mood, and illness."""
402     app.clear_right_frame()
403     ttk.Label(app.right_frame, text="Manage Events").pack(padx=10,
    pady=5)
404     create_style()
405 
406     app.file_frame = ttk.Frame(app.right_frame)
407     app.file_frame.pack(fill=tk.X, padx=10, pady=5)
408     filename_var = tk.StringVar(value=file)
409     ttk.Entry(app.file_frame, textvariable=filename_var, width=30).
    pack(side=tk.LEFT, padx=5)
410     ttk.Button(app.file_frame, text="Load", command=lambda:
    show_events_inputs(app, filename_var.get())).pack(side=tk.LEFT,
    padx=5)
411 
412     app.data["events"] = load_events_from_json(filename_var.get())
413 
414     event_types = EVENTS
415     for event_type in event_types:
416         if event_type.lower() not in app.data["events"]:
417             app.data["events"][event_type.lower()] = []
418 
419     create_input_section(app, "Exercise",  app.data["events"]['
    exercise'],  EXER_COLUMN, file)
420     create_input_section(app, "Alcohol",   app.data["events"]['alcohol
    '],   ALCO_COLUMN, file)
421     create_input_section(app, "Reservoir", app.data["events"]['
    reservoir'], RESE_COLUMN, file)
422     create_input_section(app, "Mood",      app.data["events"]['mood'],
          MOOD_COLUMN, file)
423     create_input_section(app, "Illness",   app.data["events"]['illness
    '],   ILLN_COLUMN, file)
424 
425 def show_parameters_inputs(app, file="data/params/params.json"):
426     """Show the parameters in the left frame and allow the user to
    modify them."""
427     app.clear_right_frame()
428     ttk.Label(app.right_frame, text="Model Parameters").pack(padx=10,
    pady=10)
429 
430     app.file_frame = ttk.Frame(app.right_frame)
431     app.file_frame.pack(fill=tk.X, padx=10, pady=5)
432     filename_var = tk.StringVar(value=file)
433     ttk.Entry(app.file_frame, textvariable=filename_var, width=30).
    pack(side=tk.LEFT, padx=5)
434     ttk.Button(app.file_frame, text="Load", command=lambda:
    show_parameters_inputs(app, filename_var.get())).pack(side=tk.LEFT
    , padx=5)
```

```
435
436     # Define variables for each parameter
437     app.param_vars = {}
438     default_params = load_params(filename_var.get())
439
440     for param_name, default_value in default_params.items():
441         # Create a frame for each parameter row
442         param_frame = ttk.Frame(app.right_frame)
443         param_frame.pack(padx=5, pady=2, fill='x')
444
445         # Add a label and entry side by side
446         ttk.Label(param_frame, text=param_name, width=20).pack(side="
    left")
447         app.param_vars[param_name] = tk.DoubleVar(value=default_value)
448         ttk.Entry(param_frame, textvariable=app.param_vars[param_name
    ]).pack(side="left", expand=True, fill='x')
449
450         # Add a Run button to save changes
451     ttk.Button(app.right_frame, text="Save", command=lambda:
    save_parameters(app, filename_var.get())).pack(padx=10, pady=20)
452
453 def save_parameters(app, file):
454     """Save the modified parameters to the JSON file."""
455     params_to_save = {param_name: var.get() for param_name, var in app
    .param_vars.items()}
456     save_json_data(params_to_save, file, False)
457
458 def show_inputs_inputs(app, file="data/inputs/inputs.json"):
459     """Displays inputs management section with tables for meals, bolus
    , and basal insulin."""
460     app.clear_right_frame()
461     ttk.Label(app.right_frame, text="Manage Inputs").pack(padx=10,
    pady=5)
462     create_style()
463
464     app.file_frame = ttk.Frame(app.right_frame)
465     app.file_frame.pack(fill=tk.X, padx=10, pady=5)
466     filename_var = tk.StringVar(value=file)
467     ttk.Entry(app.file_frame, textvariable=filename_var, width=30).
    pack(side=tk.LEFT, padx=5)
468     ttk.Button(app.file_frame, text="Load", command=lambda:
    show_inputs_inputs(app, filename_var.get())).pack(side=tk.LEFT,
    padx=5)
469
470     app.data = load_json_data(filename_var.get())
471
472     # Initialize the keys if they are not in App.data
473     input_types = INPUTS
474     for input_type in input_types:
475         if input_type not in app.data:
476             app.data[input_type] = []
477
478     # Create input sections
479     create_input_section(app, "Meals", app.data['meals'], MEAL_COLUMN,
     file)
480     create_input_section(app, "Bolus", app.data['bolus'], BOLU_COLUMN,
     file)
```

```
481     create_input_section(app, "Basal", app.data['basal'], BASA_COLUMN,
        file)
482
483
484
485 def create_input_section(app, name, records, columns, file):
486     """Create individual input sections for Events or Inputs."""
487     section_frame = ttk.Frame(app.right_frame)
488     section_frame.pack(fill=tk.X, padx=10, pady=5)
489
490     ttk.Label( section_frame, text=name
                ).pack(side=tk.LEFT)
491     ttk.Button(section_frame, text="Add",    command=lambda:
        add_record(     app, name, file)).pack(side=tk.LEFT)
492     ttk.Button(section_frame, text="Remove", command=lambda:
        remove_selected(app, name, file)).pack(side=tk.LEFT)
493     # Treeview for records
494     if name == "Reservoir":
495         tree = ttk.Treeview(app.right_frame, columns=columns, show="
        headings", height=1)
496     elif name in EVENTS:
497         tree = ttk.Treeview(app.right_frame, columns=columns, show="
        headings", height=2)
498     else:
499         tree = ttk.Treeview(app.right_frame, columns=columns, show="
        headings", height=4)
500     tree.pack(padx=10, pady=5, fill=tk.X)
501     for col in columns:
502         tree.heading(col, text=col)
503         if col == "Start":
504             tree.column(col, anchor=tk.CENTER, width=70)
505         else:
506             tree.column(col, anchor=tk.CENTER, width=30)
507
508     setattr(app, f"{name.lower()}_tree", tree)
509
510     # Show data in the treeview
511     populate_tree(tree, records)
512
513     # Event to start editing when selecting a record
514     tree.bind("<Double-1>", lambda event, tr=tree: edit_cell(app,
        event, tr, name, columns, file))
515
516 def populate_tree(tree, records):
517     """Populate a Treeview with records."""
518     for idx, record in enumerate(records):
519         values = (idx, *record.values())
520         tree.insert("", "end", values=values)
521
522 def edit_cell(app, event, tree, name, columns, file):
523     # Identify the item and column clicked
524     item_id   = tree.identify_row(   event.y)
525     column_id = tree.identify_column(event.x)
526
527     # Create an Entry widget to edit the cell
528     col = int(column_id[1:]) - 1
529     cell_value = tree.item(item_id, 'values')[col]
530     entry = tk.Entry(tree)
```

```
531     entry.insert(0, cell_value)
532     entry.place(x=event.x, y=event.y)
533     entry.focus_set()
534 
535     # Save value when pressing Enter
536     def on_return(event):
537         save_edited_value(app, event, tree, item_id, col, name,
    columns, file)
538 
539     # Bind Enter key to save change
540     entry.bind("<Return>", on_return)
541 
542 def save_edited_value(app, event, tree, item_id, col, name, columns,
    file):
543     """Save edited value from Entry widget to Treeview and update data
    ."""
544     entry = event.widget
545     new_value = entry.get()
546     entry.destroy()
547     column_id = tree.identify_column(event.x)
548 
549     column_index = int(column_id[1:]) - 1
550 
551     if tree["columns"][column_index] != "Start":
552         try:
553             new_value = float(new_value)
554         except ValueError:
555             log.warning("Invalid input. Expected a number.")
556             entry.destroy()
557             return
558 
559     # Update the Treeview with the new value
560     current_values = list(tree.item(item_id, 'values'))
561     current_values[col] = new_value
562     tree.item(item_id, values=current_values)
563 
564     # Save the change to the JSON file
565     record_index = int(current_values[0])
566 
567     if name in EVENTS:
568         key = KEY_MAP["Events"][name.lower()][col - 1]
569         app.data["events"][name.lower()][record_index][key] = (
570             int(new_value) if key in ["duration", "intensity"]
571             else float(new_value) if key in ["quantity", "repeat_every
    "]
572             else new_value
573         )
574         save_json_data(app.data["events"], file)
575     else:
576         key = KEY_MAP[name][col - 1]
577         app.data[name.lower()][record_index][key] = (
578             int(new_value) if key in ["duration", "UGPs"]
579             else float(new_value) if key in ["i", "iir"]
580             else new_value
581         )
582         save_json_data(app.data, file)
583 
584 def add_record(app, name, file):
```

```
585     """Add a new record to the corresponding dataset with default
    values."""
586     time = datetime.now() + timedelta(days=1)
587     time = time.isoformat(timespec='seconds')
588     if name in EVENTS:
589         default_record = {
590             "exercise":  {"start": time, "duration": 0, "intensity":
    5},
591             "alcohol":   {"start": time, "duration": 0, "quantity":
    1},
592             "reservoir": {"start": time, "repeat_every": 3},
593             "mood":      {"start": time, "duration": 0, "type": "
    Stress"},
594             "illness":   {"start": time, "duration": 0, "type": "
    Infection"}
595         }
596         new_record = default_record[name.lower()]
597         app.data["events"][name.lower()].append(new_record)
598         save_json_data(app.data["events"], file)
599         show_events_inputs(app, file)
600         return
601     elif name == "Meals":
602         new_record = {"start": time, "duration": 20, "carbs": 15, "
    UGPs": 1}
603     elif name == "Bolus":
604         new_record = {"start": time, "i": 0.5}
605     elif name == "Basal":
606         new_record = {"start": time, "duration": 60, "iir": 0.6}
607
608     app.data[name.lower()].append(new_record)
609     save_json_data(app.data, file)
610
611     show_inputs_inputs(app, file)
612
613 def remove_selected(app, name, file):
614     """Remove the selected record from the corresponding dataset."""
615     tree = getattr(app, f"{name.lower()}_tree")
616     selected_item = tree.selection()
617     if selected_item:
618         record_index = int(tree.item(selected_item, 'values')[0])
619         if name in EVENTS:
620             del app.data["events"][name.lower()][record_index]
621             save_json_data(app.data["events"], file)
622             show_events_inputs(app, file)
623         else:
624             del app.data[name.lower()][record_index]
625             save_json_data(app.data, file)
626             show_inputs_inputs(app, file)
```

Código D.4: input_section.py (DiabetesTool).

D.4.3. logging_handler.py

```
1 import logging as log
2 import tkinter as tk
```

```
3
4  class TextHandler(log.Handler):
5      """A custom logging handler that outputs log messages to a Tkinter
       Text widget."""
6      def __init__(self, text_widget):
7          super().__init__()
8          self.text_widget = text_widget
9
10         # Set the formatter for the handler
11         self.setFormatter(log.Formatter(fmt='[%(asctime)s:%(levelname)
       s] %(message)s',
12                                         datefmt='%Y-%m-%d %H:%M:%S'))
13
14     def emit(self, record):
15         """Overrides the default emit method to insert formatted log
       messages into the Tkinter Text widget."""
16         msg = self.format(record)  # Format the record using the
       handler's formatter
17         self.text_widget.configure(state='normal')
18         self.text_widget.insert(tk.END, msg + '\n')
19         self.text_widget.configure(state='disabled')
20         self.text_widget.see(tk.END)  # Auto-scroll to the end
```

Código D.5: logging_handler.py (DiabetesTool).

D.4.4. run_programs.py

```
1  from datetime import datetime
2  import os
3  import subprocess
4  from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg # type
       : ignore
5  import logging as log
6  from programs.download_main import main as download_main
7  from programs.estimator_main import main as estimator_main
8  from programs.simulator_main import main as simulator_main
9  from programs.comparator_main import main as comparator_main
10 from programs.pump_main import main as pump_main
11 from carelink_carepartner_api_login import generate_token as
       token_main
12 import tkinter as tk
13 from tkinter import ttk
14 import matplotlib.pyplot as plt # type: ignore
15
16 def display_plot(app, figure):
17     """Clear existing plot frame and embed a new matplotlib figure."""
18     for widget in app.plot_frame.winfo_children():
19         widget.destroy()
20     canvas = FigureCanvasTkAgg(figure, master=app.plot_frame)
21     canvas.draw()
22     canvas.get_tk_widget().pack(expand=True, fill=tk.BOTH)
23
24 def create_plot_tab(app, tab_name, fig):
25     """Add a new tab with a matplotlib figure inside the notebook
       widget."""
```

```
26      # Create a new tab frame
27      tab = ttk.Frame(app.notebook)
28      app.notebook.add(tab, text=tab_name)
29
30      # Add the figure to the tab
31      canvas = FigureCanvasTkAgg(fig, master=tab)
32      canvas.draw()
33      canvas.get_tk_widget().pack(expand=True, fill=tk.BOTH)
34
35
36  def run_simulator(app):
37      """Parses the user inputs and runs the simulator with given
        parameters."""
38      # Get the values from the user interface
39      start_time = app.sim_start_time.get()
40      rts = app.rts # Real Time System
41
42      # Convert start_time to datetime
43      start_time = datetime.fromisoformat(start_time)
44      duration   = int(app.sim_duration.get()) if app.sim_duration.get()
        else None
45      speed      = int(app.sim_speed.get())    if app.sim_speed.get()
        else  1
46
47      input_file  = app.sim_input_file.get()  if app.sim_input_file.get
        ()  else None
48      events_file = app.sim_events_file.get() if app.sim_events_file.get
        () else None
49      params_file = app.sim_params_file.get() if app.sim_params_file.get
        () else None
50      pump_file   = app.sim_pump_file.get()   if app.sim_pump_file.get()
          else None
51
52      # Real Time System
53      if rts:
54
55          data_file = app.sim_data_file.get() if app.sim_data_file.get()
        else None
56
57          for widget in app.plot_frame.winfo_children():
58              widget.destroy()
59
60          fig = plt.figure()  # Create a figure to use with canvas
61          canvas = FigureCanvasTkAgg(fig, master=app.plot_frame)
62          canvas.get_tk_widget().pack(expand=True, fill=tk.BOTH)
63
64          # Simulation
65          if app.glucose_source.get() == "Simulation":
66              simulator_main(start_time=start_time, rts=rts, duration =
        duration, canvas = canvas,
67                             input_file=input_file, events_file=
        events_file, params_file=params_file, data_file=data_file,
68                             speed = speed)
69          # Pump real data or pupm data from file
70          else:
71              pump_main(start_time=start_time, duration = duration,
        canvas = canvas,
```

```
72                          input_file=input_file, events_file=events_file,
    params_file=params_file,
73                          data_file=data_file, pump_file=pump_file, speed
    = speed)
74
75      else:
76
77          parameter = app.sim_parameter.get()
78          # Not variable parameter
79          if parameter == '':
80              vx = None
81              vm = None
82              iterations = 1
83          # Multiple simulations varying parameter
84          else:
85              vx = float(app.sim_vx.get()) # Maximum value of parameter
86              vm = float(app.sim_vm.get()) # Minimum value of parameter
87              iterations = int(app.sim_iterations.get()) # Number of
    iterations
88
89          app.create_notebook_area()
90
91          fig1, fig2, fig3 = simulator_main(start_time=start_time, rts=
    rts, duration=duration, parameter=parameter,vx=vx,vm=vm,iterations
    =iterations,
92                                            input_file=input_file,
    events_file=events_file, params_file=params_file)
93
94          create_plot_tab(app,"Glucose", fig1)
95          create_plot_tab(app,"State vars", fig2)
96          create_plot_tab(app,"Intermediate vars", fig3)
97
98  def run_estimator(app):
99      """Parses the user inputs and runs the estimator with given
    parameters."""
100
101     # Get the values from the user interface
102     start_date = app.est_start_time.get()
103     start_date = datetime.fromisoformat(start_date)  # Convert to a
    datetime object
104
105     # Get other parameter values
106     duration = int(app.est_duration.get()) if app.est_duration.get()
    else None
107     parameter = app.est_parameter.get()
108     vx = float(app.est_vx.get()) # Maximum value of parameter
109     vm = float(app.est_vm.get()) # Minimum value of parameter
110     iterations = int(app.est_iterations.get()) # Number of iterations
111     data_file   = app.est_data_file.get()   if app.est_data_file.get()
        else None
112     input_file  = app.est_input_file.get()  if app.est_input_file.get
    ()  else None
113     events_file = app.est_events_file.get() if app.est_events_file.get
    () else None
114     params_file = app.est_params_file.get() if app.est_params_file.get
    () else None
115
116     # Run the estimator main function with direct arguments
```

```
117     fig1, fig2 = estimator_main(start_date=start_date,duration=
        duration,parameter=parameter,vx=vx,vm=vm,iterations=iterations,
118                                 file=data_file, input_file=input_file,
         events_file=events_file, params_file=params_file)
119     create_plot_tab(app,"Res", fig1)
120     create_plot_tab(app,"Glucose", fig2)
121
122 def run_comparator(app):
123     """Parses the user inputs and runs the comparator with given
        parameters."""
124     start_time = app.comp_start_time.get()
125     start_time = datetime.fromisoformat(start_time)
126     duration   = int(app.comp_duration.get()) if app.comp_duration.get
        () else None
127     compare = app.compare.get()
128     input_file = app.comp_input_file.get()
129     ref_ins_file = app.comp_ref_ins_file.get()
130     ins_file = app.comp_ins_file.get()
131
132     # Compare glucose evolution
133     if compare == "Glucose":
134         fig = comparator_main(start_time=start_time,duration=duration,
        input_file=input_file)
135     # Compare insulin distribution
136     elif compare == "Insulin":
137         fig = comparator_main(start_time=start_time,duration=duration,
        ref_ins_file=ref_ins_file, ins_file=ins_file, compare_insulin=True
        )
138     display_plot(app,fig)
139
140
141 def run_downloader(app):
142     """Parses the user inputs and runs the downloader with given
        parameters."""
143     repeat_times = int(app.dl_repeat_times.get())
144     wait_times   = int(app.dl_wait_time.get())
145     save_data    = app.dl_save_data.get() == True
146     verbose      = app.dl_verbose.get() == True
147     fig = download_main(repeat_times=repeat_times, wait_times=
        wait_times, save_data=save_data, verbose=verbose)
148     display_plot(app,fig)
```

Código D.6: run_programs.py (DiabetesTool).

D.5. carelink

D.5.1. carelink_client2.py

```
1 ##################################################################
2 #
3 #  Carelink Client 2 library
4 #
5 #  Description:
```

```
#
#    This library implements a client for the Medtronic Carelink API
#    as used by the official Carelink Connect Android app.
#
#    Code adapted from: https://github.com/ondrej1024/carelink-python-client
#    whose author is Ondrej Wisniewski
#
########################################################################



import json
import requests
import time
import base64
import os
import logging as log
from datetime import datetime
from util.auxi import FORMAT


# Version string
VERSION = "1.1"

# Constants
DEFAULT_FILENAME="carelink/logindata.json"
# CARELINK_CONFIG_URL = 'https://clcloud.minimed.eu/connect/carepartner/v6/discover/android/3.1'
CARELINK_CONFIG_URL = 'https://clcloud.minimed.eu/connect/carepartner/v11/discover/android/3.2'
AUTH_ERROR_CODES = [401,403]
COMMON_HEADERS = {
                  "Accept": "application/json",
                  "Content-Type": "application/json",
                  "User-Agent": "Dalvik/2.1.0 (Linux; U; Android 10; Nexus 5X Build/QQ3A.200805.001)",
                 }

# Logging config
log.basicConfig(format=FORMAT, datefmt='%Y-%m-%d %H:%M:%S', level=log.INFO)


#############################################################
# Class CareLinkClient
#############################################################
class CareLinkClient(object):

   def __init__(self, tokenFile=DEFAULT_FILENAME):

      self.__version = VERSION

      # Authorization
      self.__tokenFile = tokenFile
      self.__tokenData = None
      self.__accessTokenPayload = None

```

```
59          # API config
60          self.__config = None
61
62          # User info
63          self.__username = None
64          self.__user = None
65          self.__patient = None
66          self.__country = None
67
68          # API status
69          self.__last_api_status = None
70
71      ############################################################
72      # Class internal functions
73      ############################################################
74
75      ############################################################
76      # Read token file
77      ############################################################
78      def _read_token_file(self, filename):
79          log.info("_read_token_file()")
80          token_data = None
81          log.info("   filename: %s" % filename)
82          if os.path.isfile(filename):
83              log.info("   file found")
84              try:
85                  token_data = json.loads(open(filename, "r").read())
86              except json.JSONDecodeError:
87                  log.error("ERROR: failed parsing token file %s" % filename
    )
88
89              if token_data is not None:
90                  required_fields = ["access_token", "refresh_token", "scope
    ", "client_id", "client_secret", "mag-identifier"]
91                  for f in required_fields:
92                      if f not in token_data:
93                          log.error("ERROR: field %s is missing from token
    file" % f)
94          else:
95              log.error("ERROR: token file %s not found" % filename)
96          return token_data
97
98      ############################################################
99      # Write token file
100     ############################################################
101     def _write_token_file(self, obj, filename):
102         log.info("_write_token_file()")
103         with open(filename, 'w') as f:
104             json.dump(obj, f, indent=4)
105
106     ############################################################
107     # Get Carelink API config
108     ############################################################
109     def _get_config(self, discovery_url, country):
110         log.info("_get_config()")
111         resp = requests.get(discovery_url)
112         log.debug("   status: %d" % resp.status_code)
113         data = resp.json()
```

```
114         region = None
115         config = None
116 
117         for c in data["supportedCountries"]:
118             try:
119                 region = c[country.upper()]["region"]
120                 break
121             except KeyError:
122                 pass
123         if region is None:
124             raise Exception("ERROR: country code %s is not supported" %
    country)
125         log.debug("   region: %s" % region)
126 
127         for c in data["CP"]:
128             if c["region"] == region:
129                 config = c
130                 break
131         if config is None:
132             raise Exception("ERROR: failed to get config base urls for
    region %s" % region)
133 
134         resp = requests.get(config["SSOConfiguration"])
135         log.debug("   status: %d" % resp.status_code)
136         sso_config = resp.json()
137         sso_base_url = f"https://{sso_config['server']['hostname']}:{
    sso_config['server']['port']}/{sso_config['server']['prefix']}"
138         token_url = sso_base_url + sso_config["oauth"]["system_endpoints
    "]["token_endpoint_path"]
139         c["token_url"] = token_url
140         return config
141 
142     ############################################################
143     # Get user data
144     ############################################################
145     def _get_user(self, config, token_data):
146         log.info("_get_user()")
147         url = config["baseUrlCareLink"] + "/users/me"
148         headers = COMMON_HEADERS
149         headers["mag-identifier"] = token_data["mag-identifier"]
150         headers["Authorization"] = "Bearer " + token_data["access_token"
    ]
151         self.__last_api_status = None
152         resp = requests.get(url=url,headers=headers)
153         self.__last_api_status = resp.status_code
154         log.debug("   status: %d" % resp.status_code)
155         try:
156             user = resp.json()
157         except:
158             user = None
159         return user
160 
161     ############################################################
162     # Get patient data
163     ############################################################
164     def _get_patient(self, config, token_data):
165         log.info("_get_patient()")
166         url = config["baseUrlCareLink"] + "/links/patients"
```

```
167         headers = COMMON_HEADERS
168         headers["mag-identifier"] = token_data["mag-identifier"]
169         headers["Authorization"] = "Bearer " + token_data["access_token"
    ]
170         self.__last_api_status = None
171         resp = requests.get(url=url,headers=headers)
172         self.__last_api_status = resp.status_code
173         log.debug("   status: %d" % resp.status_code)
174         try:
175             patient = resp.json()[0]
176         except:
177             patient = None
178         return patient
179 
180     ############################################################
181     # Get periodic pump and sensor data
182     ############################################################
183     def _get_data(self, config, token_data, username, role, patientid):
184         log.info("_get_data()")
185         url = config["baseUrlCumulus"] + "/display/message"
186         headers = COMMON_HEADERS
187         headers["mag-identifier"] = token_data["mag-identifier"]
188         headers["Authorization"] = "Bearer " + token_data["access_token"
    ]
189         data = {}
190         data["username"] = username
191         log.debug(patientid)
192         if role in ["CARE_PARTNER","CARE_PARTNER_OUS"]:
193             data["role"] = "carepartner"
194             data["patientId"] = patientid
195         else:
196             data["role"] = "patient"
197         log.debug("url: %s" % url)
198         log.debug("headers: %s" % json.dumps(headers))
199         log.debug("data: %s" % json.dumps(data))
200 
201         self.__last_api_status = None
202         log.debug("POST")
203         resp = requests.post(url=url,headers=headers,data=json.dumps(
    data))
204         log.debug("data: resp")
205         self.__last_api_status = resp.status_code
206         log.debug("   status: %d" % resp.status_code)
207         try:
208             my_data = resp.json()
209         except:
210             my_data = None
211         return my_data
212 
213     ############################################################
214     # Do token data refresh
215     ############################################################
216     def _do_refresh(self, config, token_data):
217         log.info("_do_refresh()")
218         token_url = config["token_url"]
219         data = {
220             "refresh_token": token_data["refresh_token"],
221             "client_id":     token_data["client_id"],
```

```
222             "client_secret": token_data["client_secret"],
223             "grant_type":    "refresh_token"
224             }
225         headers = {
226             "mag-identifier": token_data["mag-identifier"]
227             }
228         resp = requests.post(url=token_url, headers=headers, data=data)
229         log.debug("   status: %d" % resp.status_code)
230         if resp.status_code != 200:
231             raise Exception("ERROR: failed to refresh token")
232         new_data = resp.json()
233         token_data["access_token"] = new_data["access_token"]
234         token_data["refresh_token"] = new_data["refresh_token"]
235         return token_data
236 
237     ##############################################################
238     # Get access token payload
239     ##############################################################
240     def _get_access_token_payload(self, token_data):
241         log.info("_get_access_token_payload()")
242         try:
243             token = token_data["access_token"]
244         except:
245             log.debug("   no access token found")
246             return None
247         try:
248             # Decode json web token payload
249             payload_b64 = token.split('.')[1]
250             payload_b64_bytes = payload_b64.encode()
251             missing_padding = (4 - len(payload_b64_bytes) % 4) % 4
252             if missing_padding:
253                 payload_b64_bytes += b'=' * missing_padding
254             payload_bytes = base64.b64decode(payload_b64_bytes)
255             payload = payload_bytes.decode()
256             payload_json = json.loads(payload)
257             #log.debug(payload_json)
258         except:
259             log.info("   malformed access token")
260             return None
261         return payload_json
262 
263     ##############################################################
264     # Check access token validity
265     ##############################################################
266     def _is_token_valid(self, access_token_payload):
267         log.info("_is_token_valid()")
268         try:
269             # Get expiration time stamp
270             token_validto = access_token_payload["exp"]
271         except:
272             log.info("   missing data in access token")
273             return False
274 
275         # Check expiration time stamp
276         tdiff = token_validto - time.time()
277         if tdiff < 0:
278             log.info("   access token has expired %ds ago" % abs(tdiff))
279             return False
```

```
280         if tdiff < 600:
281             log.info("   access token is about to expire in %ds" % abs(
    tdiff))
282             return False
283 
284         # Token is valid
285         auth_token_validto = datetime.utcfromtimestamp(token_validto).
    strftime('%a %b %d %H:%M:%S UTC %Y')
286         log.info("   access token expires in %ds (%s)" % (tdiff,
    auth_token_validto))
287         return True
288 
289     ############################################################
290     # Init static data
291     ############################################################
292     def _init(self):
293         self.__tokenData = self._read_token_file(self.__tokenFile)
294         if self.__tokenData is None:
295             return False
296         self.__accessTokenPayload = self._get_access_token_payload(self.
    __tokenData)
297         if self.__accessTokenPayload is None:
298             return False
299         try:
300             self.__country = self.__accessTokenPayload["token_details"]["
    country"]
301             self.__config = self._get_config(CARELINK_CONFIG_URL, self.
    __country)
302             self.__username = self.__accessTokenPayload["token_details"][
    "preferred_username"]
303             self.__user = self._get_user(self.__config, self.__tokenData)
304             if self.__user["role"] in ["CARE_PARTNER","CARE_PARTNER_OUS"
    ]:
305                 self.__patient = self._get_patient(self.__config, self.
    __tokenData)
306         except Exception as e:
307             log.error(e)
308             if self.__last_api_status in AUTH_ERROR_CODES:
309                 try:
310                     self.__tokenData = self._do_refresh(self.__config, self
    .__tokenData)
311                     self.__accessTokenPayload = self.
    _get_access_token_payload(self.__tokenData)
312                     self._write_token_file(self.__tokenData, self.
    __tokenFile)
313                 except Exception as e:
314                     log.error(e)
315             return False
316         return True
317 
318 
319     ############################################################
320     # Class public functions
321     ############################################################
322 
323     ############################################################
324     # Init object
325     ############################################################
```

```
326     def init(self):
327         # First try
328         if self._init() == False:
329             # Second try (after token refresh)
330             if self._init() == False:
331                 # Failed permanently
332                 log.error("ERROR: unable to initialize")
333                 return False
334         return True
335 
336     ############################################################
337     # Print user info
338     ############################################################
339     def printUserInfo(self):
340         log.info("User Info:")
341         log.info("    user:     %s (%s %s)" % (self.__username, self.
    __user["firstName"], self.__user["lastName"]))
342         log.info("    role:     %s" % self.__user["role"])
343         log.info("    country:  %s" % self.__country)
344         if self.__patient is not None:
345             log.info("    patient:  %s (%s %s)" % (self.__patient["
    username"],self.__patient["firstName"],self.__patient["lastName"])
    )
346 
347     ############################################################
348     # Get recent periodic pump data
349     ############################################################
350     def getRecentData(self):
351         # Check if access token is valid
352         if not self._is_token_valid(self.__accessTokenPayload):
353             self.__tokenData = self._do_refresh(self.__config, self.
    __tokenData)
354             self.__accessTokenPayload = self._get_access_token_payload(
    self.__tokenData)
355             self._write_token_file(self.__tokenData, self.__tokenFile)
356             if not self._is_token_valid(self.__accessTokenPayload):
357                 log.error("ERROR: unable to get valid access token")
358                 return None
359 
360         if self.__patient is not None:
361             patientId = self.__patient["username"]
362         else:
363             patientId = None
364 
365         # Get data: first try
366         data = self._get_data(self.__config,
367                               self.__tokenData,
368                               self.__username,
369                               self.__user["role"],
370                               patientId)
371 
372         # Check API response
373         if self.__last_api_status in AUTH_ERROR_CODES:
374             # Try to refresh token
375             self.__tokenData = self._do_refresh(self.__config, self.
    __tokenData)
376             self.__accessTokenPayload = self._get_access_token_payload(
    self.__tokenData)
```

```
377             self._write_token_file(self.__tokenData, self.__tokenFile)
378
379             # Get data: second try
380             data = self._get_data(self.__config,
381                                   self.__tokenData,
382                                   self.__username,
383                                   self.__user["role"],
384                                   patientId)
385             # Check API response
386             if self.__last_api_status in AUTH_ERROR_CODES:
387                 # Failed permanently
388                 log.error("ERROR: unable to get data")
389                 return None
390         # return data
391         return data.get("patientData", {})
392
393     ############################################################
394     # Get last API response code
395     ############################################################
396     def getLastResponseCode(self):
397         return self.__last_api_status
398
399     ############################################################
400     # Get Client library version
401     ############################################################
402     def getClientVersion(self):
403         return self.__version
```

Código D.7: carelink_client2.py (DiabetesTool).

D.5.2. data.py

```
1
2  from carelink.carelink_client2 import CareLinkClient
3  from carelink_carepartner_api_login import generate_token
4  import datetime
5  import logging as log
6  import time
7  from carelink.json_carelink import write_json, simplify_json,
       append_simple_json, load_json, delete_json
8  from dateutil.parser import parse as parse_datetime # type: ignore
9  from inputs.json_data import save_data_to_json
10 from util.auxi import FORMAT, MIN_H, make_naive_param
11
12 # Logging settings
13 log.basicConfig(format=FORMAT, datefmt='%Y-%m-%d %H:%M:%S', level=log.
       INFO)
14
15 def init_client(verbose=False, generate = False):
16     """Initialize Carelink client."""
17     log.info("Initializing Carelink client...")
18     if verbose:
19         log.getLogger().setLevel(log.DEBUG)
20
21     client = CareLinkClient()
```

```
22      log.debug("Created customer")
23
24      if not client.init() and generate:
25          log.info("Generating Token...")
26          generate_token()
27
28      if client.init():
29          client.printUserInfo()
30          return client
31      else:
32          log.error("Error: Failure to initialize the client (answer
    code: %s)" % client.getLastResponseCode())
33          return None
34
35  def download_data(client, save_data=False, repeat=1, wait=5):
36      """Download recent customer data and keep them in a JSON file if
    specified."""
37      for i in range(repeat):
38          log.debug("Starting download, attempt: %d" % (i + 1))
39          try:
40              recent_data = client.getRecentData()
41              if recent_data is not None and client.getLastResponseCode
    () == 200:
42                  last_sg = recent_data["lastSG"]["sg"]
43                  log.info(f"Last SG: {last_sg}")
44                  if i==0:
45                      filename = "data/data/data" + "-" + datetime.
    datetime.now().strftime("%Y%m%d_%H%M%S") + ".json"
46                      if write_json(recent_data, filename):
47                          log.debug("Successful data")
48                      filename_simple = "data/data/simple_data" + "-" +
    datetime.datetime.now().strftime("%Y%m%d_%H%M%S") + ".json"
49                      if simplify_json(filename, filename_simple):
50                          log.debug("Simplified data successful")
51
52                      dw_filename = "data/inputs/downloaded_inputs" + "-
    " + datetime.datetime.now().strftime("%Y%m%d_%H%M%S") + ".json"
53
54                      simple_to_inputs(filename_simple, dw_filename)
55
56                  else:
57                      if append_simple_json(filename_simple, recent_data
    ):
58                          log.debug("Success -appendant data")
59              else:
60                  log.error("Error: failure obtaining the data (answer
    code %d)" % client.getLastResponseCode())
61                  break
62          except Exception as e:
63              log.error(e)
64              break
65
66          if i < repeat - 1:
67              log.debug("Waiting %d minutes before the next download " %
     wait)
68              time.sleep(wait * 60)
69
70      if not save_data:
```

```
71          delete_json(filename)
72
73      return filename_simple, dw_filename
74
75
76  def simple_to_inputs(filename_simple, dw_filename):
77
78      data = load_json(filename_simple)
79      meals, basal, bolus = parse_markers(data)
80
81      # Convert dates into Meals, Bolus and Basal to Naive
82      make_naive_param(meals,  'start')
83      make_naive_param(bolus,  'start')
84      make_naive_param(basal,  'start')
85
86      # GSave the data paired in a JSON file with the Inputs format
87      inputs_data = {
88          "meals": meals,
89          "bolus": bolus,
90          "basal": basal
91      }
92      save_data_to_json(inputs_data, dw_filename)
93      return
94
95  def parse_markers(data):
96      """Parses the markers from the JSON data to generate meals, basal,
        and bolus lists."""
97      meals = parse_meals_markers(data.get("markers", []))
98      basal = parse_basal_markers(data.get("markers", []))
99      bolus = parse_bolus_markers(data.get("markers", []))
100
101     return meals, basal, bolus
102
103 def parse_meals_markers(markers):
104     """Parses MEAL markers to generate the meals list."""
105     meals = []
106     for marker in markers:
107         if marker['type'] == 'MEAL':
108             amount = marker.get('amount', 0)
109             meal = {
110                 'start': parse_datetime(marker['dateTime']),
111                 'duration': 5,  # We consider a fixed duration for
        meals, adjustable if there is more information
112                 'carbs': amount,
113                 'UGPs':  0
114             }
115             meals.append(meal)
116     return meals
117
118 def parse_basal_markers(markers):
119     """Parses AUTO_BASAL_DELIVERY markers to generate the basal list.
        """
120     basal = []
121     for marker in markers:
122         if marker['type'] == 'AUTO_BASAL_DELIVERY':
123             duration = 5 # minutes
124             iir = float(marker.get('bolusAmount', 0)) / duration *
        MIN_H # IIR in U/h
```

```
125             basal_entry = {
126                 'start': parse_datetime(marker['dateTime']),
127                 'duration': duration,
128                 'iir': iir
129             }
130             basal.append(basal_entry)
131     return basal
132 
133 def parse_bolus_markers(markers):
134     """Parses INSULIN markers to generate the bolus list."""
135     bolus = []
136     for marker in markers:
137         if marker['type'] == 'INSULIN':
138             bolus_entry = {
139                 'start': parse_datetime(marker['dateTime']),
140                 'i': marker.get('deliveredFastAmount', 0)
141             }
142             bolus.append(bolus_entry)
143     return bolus
144 
145 def parse_results_sgs(sgs):
146     """Generates results from SG readings in the JSON."""
147     results = []
148     for sg in sgs:
149         time = parse_datetime(sg['datetime'])
150         glucose_level = sg['sg']
151         results.append({
152             'time': time,
153             'glucose': glucose_level
154         })
155     return results
```

Código D.8: data.py (DiabetesTool).

D.5.3. json_carelink.py

```
1 
2 import json
3 import os
4 from datetime import datetime
5 import logging as log
6 
7 logindata_file = 'carelink/logindata.json'
8 
9 def write_json(jsonobj, filename):
10     """ Wrute a JSON object to a file. """
11     try:
12         with open(filename, "w") as f:
13             json.dump(jsonobj, f, indent=3)
14     except Exception as e:
15         log.error("Error: Failure when saving %s (%s) " % (filename,
       str(e)))
16         return False
17     else:
18         return True
```

```
19
20
21  def simplify_json(filename, filename_simple):
22      """ Simplify the JSON file by extracting relevant fields and
        removing unnecessary data."""
23      try:
24          # Load the original JSON file
25          with open(filename, 'r') as f:
26              data = json.load(f)
27
28          # Create a new simplified dictionary
29          simple_data = {
30              "sLastSensorTime": data.get("lastSG", {}).get("timestamp")
        ,
31              "lastSG": {
32                  "sg": data.get("lastSG", {}).get("sg"),
33                  "datetime": data.get("lastSG", {}).get("timestamp")
34              },
35              "activeInsulin": {
36                  "amount": data.get("activeInsulin", {}).get("amount"),
37                  "datetime": data.get("activeInsulin", {}).get("
        datetime")
38              },
39              "sgs": [
40                  {
41                      "sg": sg.get("sg"),
42                      "datetime": sg.get("timestamp")
43                  } for sg in data.get("sgs", [])
44              ],
45              "markers": []
46          }
47
48          # Process markers
49          for marker in data.get("markers", []):
50              if marker.get("type") in ["AUTO_MODE_STATUS", "
        LOW_GLUCOSE_SUSPENDED", "BG_READING"]:
51                  continue
52
53              item = {
54                  "type": marker.get("type"),
55                  "dateTime": marker.get("timestamp") or marker.get("
        dateTime")
56              }
57
58              # Extract specific fields of dataValues
59              data_values = marker.get("data", {}).get("dataValues", {})
60              for field in ["amount", "activationType", "
        deliveredFastAmount", "bolusAmount"]:
61                  value = marker.get(field, data_values.get(field))
62                  if field == "deliveredFastAmount" and isinstance(value
        , str):
63                      try:
64                          value = float(value)
65                      except ValueError:
66                          pass
67                  if value is not None:
68                      item[field] = value
69
```

```
70              simple_data["markers"].append(item)
71
72          # Save the simplified file
73          with open(filename_simple, 'w') as f_simple:
74              json.dump(simple_data, f_simple, indent=3)
75
76          return True
77
78      except Exception as e:
79          log.error(f"Error simplifying JSON: {e}")
80          return False
81
82
83  def append_simple_json(filename_simple, recent_data):
84      """ Append new data to the existing simplified JSON file. """
85      try:
86          # Read the existing simplified file
87          with open(filename_simple, 'r') as f:
88              simple_data = json.load(f)
89
90          # Update sLastsensortime, lastSG and activeInsulin
91          simple_data["sLastSensorTime"] = recent_data.get("
    sLastSensorTime", simple_data.get("sLastSensorTime"))
92
93          if "lastSG" in recent_data:
94              simple_data["lastSG"] = {
95                  "sg": recent_data["lastSG"].get("sg", simple_data["
    lastSG"].get("sg")),
96                  "datetime": recent_data["lastSG"].get("datetime",
    simple_data["lastSG"].get("datetime"))
97              }
98
99          if "activeInsulin" in recent_data:
100             simple_data["activeInsulin"] = {
101                 "amount": recent_data["activeInsulin"].get("amount",
    simple_data["activeInsulin"].get("amount")),
102                 "datetime": recent_data["activeInsulin"].get("datetime
    ", simple_data["activeInsulin"].get("datetime"))
103             }
104
105         # Add new sgs, if datetime is posterior to the last sg
106         last_sg_time = datetime.fromisoformat(simple_data["sgs"][-1]["
    datetime"]) if simple_data["sgs"] else None
107         new_sgs = [
108             {
109                 "sg": sg.get("sg"),
110                 "datetime": sg.get("datetime")
111             } for sg in recent_data.get("sgs", [])
112             if last_sg_time is None or datetime.fromisoformat(sg["
    datetime"]) > last_sg_time
113         ]
114         simple_data["sgs"].extend(new_sgs)
115
116         # Add new Markers (only type MEAL, INSULIN,
    AUTO_BASAL_DELIVERY)
117         last_marker_time = datetime.fromisoformat(simple_data["markers
    "][-1]["dateTime"]) if simple_data["markers"] else None
118         new_markers = [
```

```
119             {
120                 "type": marker.get("type"),
121                 "dateTime": marker.get("dateTime"),
122                 "amount": marker.get("amount", None),
123                 "activationType": marker.get("activationType", None),
124                 "deliveredFastAmount": marker.get("deliveredFastAmount
    ", None),
125                 "bolusAmount": marker.get("bolusAmount", None)
126             } for marker in recent_data.get("markers", [])
127             if marker.get("type") in ["MEAL", "INSULIN", "
    AUTO_BASAL_DELIVERY"] and
128                 (last_marker_time is None or datetime.fromisoformat(
    marker["dateTime"]) > last_marker_time)
129         ]
130 
131         # Clean None and add the new markers
132         for marker in new_markers:
133             keys_to_remove = [k for k, v in marker.items() if v is
    None]
134             for key in keys_to_remove:
135                 del marker[key]
136 
137         simple_data["markers"].extend(new_markers)
138 
139         # Save the updated file
140         with open(filename_simple, 'w') as f:
141             json.dump(simple_data, f, indent=3)
142 
143         return True
144 
145     except Exception as e:
146         log.error(f"Error: Failure by adding new data to the
    Simplified JSON file ({str(e)})")
147         return False
148 
149 def load_json(filename):
150     with open(filename, 'r') as file:
151         data = json.load(file)
152     return data
153 
154 def delete_json(file, verbose=False):
155     """ Elimina el archivo JSON especificado. """
156     try:
157         if os.path.exists(file):
158             os.remove(file)
159             if verbose:
160                 log.info(f"File '{file}' properly eliminated.")
161             return True
162         else:
163             log.error(f"File '{file}' does not exist.")
164             return False
165     except Exception as e:
166         log.error(f"Error eliminating the file '{file}': {str(e)}")
167         return False
168 
169 def read_token_file(file):
170     """ Reads the token file and returns the token data. """
171     token_data = None
```

```
172     if os.path.isfile(file):
173         try:
174             token_data = json.loads(open(file, "r").read())
175         except json.JSONDecodeError:
176             log.info("failed parsing json")
177 
178         if token_data is not None:
179             required_fields = ["access_token", "refresh_token", "scope
    ", "client_id", "client_secret", "mag-identifier"]
180             for i in required_fields:
181                 if i not in token_data:
182                     log.info(f"field {i} is missing from data file")
183                     return None
184     return token_data
185 
186 def write_datafile(obj, filename):
187     """ Writes the data file. """
188     log.info("wrote data file")
189     with open(filename, 'w') as f:
190         json.dump(obj, f, indent=4)
```

Código D.9: json_carelink.py (DiabetesTool).

D.5.4. pump.py

```
1  from datetime import timedelta
2  import threading
3  import logging as log
4  import time
5  from carelink.data import download_data, init_client, parse_markers
6  from carelink.json_carelink import load_json, write_json
7  from dateutil.parser import parse as parse_datetime  # type: ignore
8  from inputs.json_data import load_data_from_json
9  from util.auxi import FORMAT, MIN_H, S_MIN, make_naive,
       momentary_activeInsulin
10 from util.glucose_plot import animate_glucose_and_inputs,
       animate_glucose_init, carb_intake_function, insulin_bolus_bars,
       insulin_intake_function
11 from simulator.auxi    import meal_gp_intake
12 
13 # Logging configuration
14 log.basicConfig(format=FORMAT, datefmt='%Y-%m-%d %H:%M:%S', level=log.
       INFO)
15 
16 MIN_PUMP_STEP = 5
17 SEC_PUMP_SLEEP = MIN_PUMP_STEP * S_MIN
18 MAX_HOURS_PLOT = 6
19 RTS_RECORDS = int(MAX_HOURS_PLOT * MIN_H / MIN_PUMP_STEP)
20 
21 
22 class Pump:
23     def __init__(self, duration=120, input_file='data/inputs/inputs.
       json',
24                  events_file='data/events/events.json', params_file='
       data/params/params.json', data_file='data/data/data.json',
```

```
    pump_file=None, speed = 1):

        self.duration = duration  # in minutes
        self.input_file = input_file
        self.events_file = events_file
        self.params_file = params_file
        self.data_file = data_file
        self.pump_file = pump_file
        self.speed = speed
        self.pump_results = []



    def run_pump_data_in_thread(self, start_time, plot=False, canvas=
    None):
        """Start the pump data process in a separate thread."""
        threading.Thread(
            target=self.run_pump_data,  # Target method to run
            args=(start_time, plot, canvas),  # Arguments for the
    method
            daemon=True  # Ensure the thread stops when the main
    program exits
        ).start()

    def run_pump_data(self, start_time, plot=False, canvas=None):
        """Main function to run the pump data simulation."""
        fig, ax1, glucose_line, insulin_line, carb_line, times,
    glucose, ti, insulin, tc, chs = animate_glucose_init()

        if canvas:
            canvas.figure = fig
            canvas.draw()
            canvas.flush_events()

        if self.pump_file is None:
            client = init_client()
            save_data = False
            repeat = 1
            wait = 0

        t = start_time
        t_old = t - timedelta(hours=MAX_HOURS_PLOT)


        # Load initial data
        if self.pump_file is None:
            filename_simple, dw_filename  = download_data(client,
    save_data=save_data, repeat=repeat, wait=wait)
        else:
            filename_simple = self.pump_file
        data                                = load_json(filename_simple)
        meals, ibasal, ibolus               = parse_markers(data)

        # Preload historical SG data and times
        self.preload_data(data, meals, ibasal, t_old, t, times,
    glucose, ti, insulin, tc, chs)

        tmax = start_time + timedelta(minutes=self.duration)
```

```
76
77          self.pre_store_results(data, t)
78
79          # Main simulation loop
80          time_wait = 0
81          while t <= tmax:
82              step_start_time = time.time()
83
84              # Download data if no pump file is provided
85              if self.pump_file is None:
86                  filename_simple, dw_filename = download_data(client,
    save_data=save_data, repeat=repeat, wait=wait)
87                  data                          = load_json(
    filename_simple)
88                  meals, ibasal, ibolus         = parse_markers(data)
89
90              self.store_results(data, times, glucose, ti, insulin, tc,
    chs, t, ibasal, meals)
91              self.trim_records(       times, glucose, ti, insulin, tc,
    chs                   )
92              self.print_results()
93              insulin_bolus_bars(ax1, ibolus)
94              animate_glucose_and_inputs(fig, ax1, glucose_line,
    insulin_line, carb_line, times, glucose, ti, insulin, tc, chs,
    canvas)
95
96              if canvas:
97                  canvas.draw()
98                  canvas.flush_events()
99
100             elapsed_real_time = time.time() - step_start_time
101             time_wait += MIN_PUMP_STEP*S_MIN/self.speed-
    elapsed_real_time
102             time.sleep(max(0,time_wait))
103             if time_wait < 0:
104                 log.warning(f'Simulation may run slower than expected
    {time_wait}')
105             else:
106                 time_wait = 0
107             t += timedelta(minutes=MIN_PUMP_STEP)
108
109         return self.pump_results
110
111     def print_results(self):
112         """Print the results to the output file"""
113         write_json(self.pump_results,self.data_file)
114
115     def preload_data(self, data, meals_old, ibasal_old, t_old, t,
    times, glucose, ti, insulin, tc, chs):
116         """Load and filter SG data for the initial plot."""
117         sg_data = []
118         if "lastSG" in data and "datetime" in data["lastSG"]:
119             last_sg_time = make_naive(parse_datetime(data["lastSG"]["
    datetime"]))
120             if t_old <= last_sg_time <= t:
121                 sg_data.append((last_sg_time, data["lastSG"]["sg"]))
122
123         if "sgs" in data:
```

```
124             for record in data["sgs"]:
125                 sg_time = make_naive(parse_datetime(record["datetime"
    ]))
126                 if t_old <= sg_time <= t:
127                     sg_data.append((sg_time, record["sg"]))
128 
129         sg_data.sort(key=lambda x: x[0])
130         for sg_time, sg_value in sg_data:
131             times.append(sg_time)
132             glucose.append(sg_value)
133 
134         # Generate data for insulin and carbs every interval
135         current_time = t_old
136         while current_time <= t:
137             # Append insulin and carbs calculated using the provided
    functions
138             insulin.append(insulin_intake_function(current_time,
    ibasal_old))
139             chs.append(carb_intake_function(current_time, meals_old))
140 
141             # Append the time to ti and tc for consistency
142             ti.append(current_time)
143             tc.append(current_time)
144 
145             # Increment time by the interval
146             current_time += timedelta(minutes=1)
147 
148 
149     def retrieve_sg(self, data, t):
150         """Retrieve the closest SG value to time t from the data."""
151         def compute_active_insulin(data, t):
152             """Compute the active insulin at time t."""
153             insulin_duration = 2.0  # Default insulin duration in
    hours
154             _, _, ibolus     = parse_markers(data)
155             return momentary_activeInsulin(insulin_duration, t, ibolus
    )
156 
157         # Initialize default values
158         closest_time = None
159         closest_glucose = None
160         log.info(f"Time: {t}")
161 
162         # Convert 'lastSG' time to datetime and compare with t
163         if "lastSG" in data and "datetime" in data["lastSG"]:
164             last_sg_time = make_naive(parse_datetime(data["lastSG"]["
    datetime"]))
165             log.info(f"last_sg_time: {last_sg_time}")
166 
167             if last_sg_time <= t:
168                 closest_time    = last_sg_time
169                 closest_glucose = data["lastSG"]["sg"]
170                 activeInsulin   = data["activeInsulin"]["amount"]
171                 return {
172                     "time": closest_time,
173                     "Gp": closest_glucose,
174                     "activeInsulin": activeInsulin,
175                 }
```

```
176         else:
177             log.info("No last sg:")
178             log.info(data)
179 
180         # Check the 'sgs' list and look for the closest time that is
    earlier or equal to t
181         if "sgs" in data:
182             for record in data["sgs"]:
183                 sg_time = make_naive(parse_datetime(record["datetime"
    ]))
184 
185                 if sg_time <= t and (closest_time is None or sg_time >
     closest_time):
186                     closest_time = sg_time
187                     closest_glucose = record["sg"]
188                     activeInsulin   = compute_active_insulin(data,
    sg_time)
189         else:
190             log.info("No  sgs:")
191             log.info(data)
192 
193         # Return the found time and glucose value, or default values
    if nothing is found
194         if closest_time is not None:
195             return {
196                 "time": closest_time,
197                 "Gp": closest_glucose,
198                 "activeInsulin": activeInsulin,
199             }
200         else:
201             # If no valid value is found, return t with a default Gp
    value
202             return {
203                 "time": t,
204                 "Gp": 100,  # Default value
205             }
206 
207     def pre_store_results(self, data, t):
208         """Store historical glucose data before time t in pump results
    ."""
209         if "sgs" in data:
210             for record in data["sgs"]:
211                 sg_time = make_naive(parse_datetime(record["datetime"
    ]))
212                 if sg_time < t:
213                     self.pump_results.append({
214                         "time": sg_time.isoformat(),
215                         "glucose": record["sg"],
216                         "activeInsulin": 0.0,
217                     })
218 
219     def store_results(self, data, times, glucose, ti, insulin, tc, chs
    , t, ibasal, meals):
220         """Store the results of the simulation."""
221         output = self.retrieve_sg(data, t)
222         log.info(output)
223         result = {
224             "time":          output['time'].isoformat(),
```

```
225             "glucose":        output['Gp'],
226             "activeInsulin": output['activeInsulin'],
227         }
228         self.pump_results.append(result)
229 
230         times.append(  output["time"])
231         ti.append(     output["time"])
232         tc.append(     output["time"])
233         glucose.append(output["Gp"]  )
234         insulin.append(insulin_intake_function(t, ibasal))
235         chs.append(carb_intake_function(t, meals))
236 
237     def trim_records(self, times, glucose, ti, insulin, tc, chs):
238         """Trim the records to keep only the last RTS_RECORDS entries.
    """
239         times[:]    = times[  -RTS_RECORDS:]
240         #ti[:]       = ti[     -RTS_RECORDS:]
241         #tc[:]       = tc[     -RTS_RECORDS:]
242         glucose[:] = glucose[-RTS_RECORDS:]
243         #insulin[:] = insulin[-RTS_RECORDS:]
244         #chs[:]      = chs[    -RTS_RECORDS:]
```

Código D.10: pump.py (DiabetesTool).

D.6. inputs

D.6.1. input_auxi.py

```
1 
2 from tkinter import  ttk
3 
4 MEAL_COLUMN =  ["Index", "Start", "Duration", "Carbs", "UGPs"]
5 BOLU_COLUMN =  ["Index", "Start", "I (Units)"]
6 BASA_COLUMN =  ["Index", "Start", "Duration", "IIR (U/h)"]
7 EXER_COLUMN =  ["Index", "Start", "Duration", "Intensity"]
8 ALCO_COLUMN =  ["Index", "Start", "Duration", "Quantity"]
9 RESE_COLUMN =  ["Index", "Start", "Repeat Every"]
10 MOOD_COLUMN =  ["Index", "Start", "Duration", "Type"]
11 ILLN_COLUMN =  ["Index", "Start", "Duration", "Type"]
12 EVENTS      =  ["Alcohol", "Exercise", "Mood", "Illness", "Reservoir"]
13 INPUTS      =  ["meals", "bolus", "basal"]
14 
15 KEY_MAP = {
16     "Meals": ["start", "duration", "carbs", "UGPs"],
17     "Bolus": ["start", "i"],
18     "Basal": ["start", "duration", "iir"],
19     "Events": {
20         "exercise": ["start", "duration", "intensity"],
21         "alcohol": ["start", "duration", "quantity"],
22         "reservoir": ["start", "repeat_every"],
23         "mood": ["start", "duration", "type"],
24         "illness": ["start", "duration", "type"]
25     }
```

```
}

DEFAULT_EVENT = {
    "alcohol": [],
    "exercise": [],
    "mood": [],
    "illness": [],
    "reservoir": []
}
DEFAULT_PARAMS = {
"BW": 60.0,           # kg (Body Weight)
"Gpeq": 110.0,        # mg/dl (Plasma Glucose Equilibrium)
"k12": 0.07714,       # min^-1 (Transfer rate from compartment 1 to 2)
"ka1": 0.0066,        # min^-1 (Absorption rate constant 1)
"ka2": 0.06,          # min^-1 (Absorption rate constant 2)
"ka3": 0.3,           # min^-1 (Absorption rate constant 3)
"ke": 0.1592,         # min^-1 (Insulin elimination rate)
"VI": 0.1364,         # l/kg (Insulin distribution volume)
"VG": 0.1164,         # l/kg (Glucose distribution volume)
"AG": 0.9,            # - (Glucose bioavailability, unitless)
"tmaxG": 20.0,        # min (Time to maximum glucose absorption)
"SIT": 0.04269,       # min^-1/(mU/l) (Insulin sensitivity in tissue)
"SID": 0.0004342,     # min^-1/(mU/l) (Insulin sensitivity in liver for
    EGP suppression)
"SIE": 0.007857,      # min^-1/(mU/l) (Insulin sensitivity in liver for
    glucose effectiveness)
"EGP0": 0.0241,       # mmol/kg/min (Endogenous glucose production at
   baseline)
"F01": 0.0107,        # mmol/kg/min (Non-insulin-dependent glucose
   utilization)
"tmaxI": 54.0,        # min (Time to maximum insulin absorption)
"i_delay": 15.0,      # min (Insulin effect delay)
"m_delay": -5.0       # min (Meal effect delay)
}
FAULT_INPUTS = {
    "meals": [],
    "bolus": [],
    "basal": []
}

def create_style():
    # Configurar estilos de filas alternas
    style = ttk.Style()
    style.configure("Treeview", rowheight=25)
    style.map("Treeview", background=[("selected", "blue")])
    style.configure("Treeview.Heading", font=("Arial", 8, "bold"))

    style.configure("evenrow", background="lightblue")
    style.configure("oddrow", background="lightgreen")
```

Código D.11: input_auxi.py (DiabetesTool).

D.6.2. json_data.py

```

```

```
import json
from datetime import datetime
from tkinter import messagebox
from util.auxi import make_naive

import logging as log
def load_data_from_json(file_path, meals_=None, ibolus_=None, ibasal_=None):
    """ Loads meals, insulin boluses, and basal insulin from a JSON file, appending them to existing lists and sorting them by time.
    """
    if meals_ is None:
        meals = []
    else:
        meals = meals_.copy()
    if ibolus_ is None:
        ibolus = []
    else:
        ibolus = ibolus_.copy()
    if ibasal_ is None:
        ibasal = []
    else:
        ibasal = ibasal_.copy()

    with open(file_path, 'r') as file:
        data = json.load(file)

    meals.extend(
        {**meal, 'start': datetime.fromisoformat(meal['start'])}
        for meal in data['meals']
    )
    ibolus.extend(
        {**bolus, 'start': datetime.fromisoformat(bolus['start'])}
        for bolus in data['bolus']
    )
    ibasal.extend(
        {**basal, 'start': datetime.fromisoformat(basal['start'])}
        for basal in data['basal']
    )

    meals.sort( key=lambda x: make_naive(x['start']))
    ibolus.sort(key=lambda x: make_naive(x['start']))
    ibasal.sort(key=lambda x: make_naive(x['start']))

    return meals, ibolus, ibasal

def load_params(params_file):
    """Load parameters from the specified JSON file."""
    try:
        with open(params_file, 'r') as file:
            params = json.load(file)
    except Exception as e:
        log.info(f"Error loading parameters: {e}")
        params = {}
    return params


def load_json_data(file_path='data/inputs/inputs.json'):
```

```
    """Load JSON data from the specified file path. If the file does not exist, return an empty structure."""
    try:
        with open(file_path, 'r') as file:
            return json.load(file)
    except FileNotFoundError:
        return {"meals": [], "bolus": [], "basal": []}

def save_json_data(data, file_path='data/inputs/inputs.json', msg = False):
    """Save JSON data to the specified file path. If the file does not exist, create it."""
    with open(file_path, 'w') as file:
        json.dump(data, file, indent=4)
    if msg:
        messagebox.showinfo("Saved", "Successful saved data")

def save_data_to_json(data, filename, verbose = False):
    """Save data to a JSON file."""

    def datetime_converter(o):
        if isinstance(o, datetime):
            return o.isoformat()
        raise TypeError(f"Object of type {o.__class__.__name__} is not JSON serializable")

    try:
        with open(filename, "w") as f:
            json.dump(data, f, indent=3, default=datetime_converter)
        if verbose:
            log.info(f"Successfully saved data in {filename}.")
        return True
    except Exception as e:
        log.error(f"Error: Failure to save the data in {filename}. Error: {str(e)}")
        return False

def load_events_from_json(filepath):
    """Load events data from a JSON file."""
    try:
        with open(filepath, 'r') as file:
            return json.load(file)
    except FileNotFoundError:
        return {
            "exercise": [],
            "alcohol": [],
            "reservoir_change": [],
            "mood": [],
            "illness": []
        }
```

Código D.12: json_data.py (DiabetesTool).

D.7. model

D.7.1. ambridge_model.py

```
from datetime import datetime
from model.events import parameters_w_events
import model.meal as m
import model.glucose as g
import model.insulin as i
from inputs.json_data import load_params, save_json_data

class Cambridge:
    def __init__(self, params_file='data/params/params.json',
                 events_file='data/events/events.json'):
        # Load parameters from the JSON file
        self.load_params(params_file)
        self.start_time = datetime.now()

        self.events_file = events_file
        self.params_file = params_file

        # Model state
        self.state = {
            "Q1": 0,  # Glucose in the accessible compartment (mmol)
            "Q2": 0,  # Glucose in the non-accessible compartment (
    mmol)
            "S1": 0,  # Insulin in compartment 1 (mU)
            "S2": 0,  # Insulin in compartment 2 (mU)
            "I": 0,   # Insulin in plasma (mU/l)
            "x1": 0,  # Insulin action on glucose transport (1/min)
            "x2": 0,  # Insulin action on glucose disposal (1/min)
            "x3": 0,  # Insulin action on endogenous glucose
    production (unitless)
            "D1": 0,  # Glucose in compartment 1 (mmol)
            "D2": 0   # Glucose in compartment 2 (mmol)
        }

    def set_events_file(self, events_file):
        """Set the events file path."""
        self.events_file = events_file

    def set_params_file(self, params_file):
        """Set the parameters file path."""
        self.params_file = params_file

    def load_params(self, params_file):
        """Load parameters from the specified JSON file."""
        params = load_params(params_file)

        # Set parameters, using defaults if not specified in the JSON
    file
        self.BW = params.get('BW', 60)
        self.Gpeq = params.get('Gpeq', 140)
        self.k12 = params.get('k12', 0.07714)
        self.ka1 = params.get('ka1', 0.006)
        self.ka2 = params.get('ka2', 0.06)
```

```
50          self.ka3 = params.get('ka3', 0.3)
51          self.ke = params.get('ke', 0.1592)
52          self.VI = params.get('VI', 0.1164)
53          self.VG = params.get('VG', 0.1164)
54          self.AG = params.get('AG', 0.785)
55          self.tmaxG = params.get('tmaxG', 79)
56          self.SIT = params.get('SIT', 0.02269)
57          self.SID = params.get('SID', 0.0002342)
58          self.SIE = params.get('SIE', 0.017857)
59          self.EGP0 = params.get('EGP0', 0.0141)
60          self.F01 = params.get('F01', 0.0107)
61          self.tmaxI = params.get('tmaxI', 54)
62          self.i_delay = params.get('i_delay', 30)
63          self.m_delay = params.get('m_delay', -5)
64
65      def save_params(self, params_file='data/params/params.json'):
66          """Save the current parameters to the specified JSON file."""
67          params = {
68              'BW': self.BW,
69              'Gpeq': self.Gpeq,
70              'k12': self.k12,
71              'ka1': self.ka1,
72              'ka2': self.ka2,
73              'ka3': self.ka3,
74              'ke': self.ke,
75              'VI': self.VI,
76              'VG': self.VG,
77              'AG': self.AG,
78              'tmaxG': self.tmaxG,
79              'SIT': self.SIT,
80              'SID': self.SID,
81              'SIE': self.SIE,
82              'EGP0': self.EGP0,
83              'F01': self.F01,
84              'tmaxI': self.tmaxI,
85              'i_delay': self.i_delay,
86              'm_delay': self.m_delay
87          }
88          save_json_data(params, params_file, msg = False)
89
90
91      def evaluate_parameter_values_at(self, t, file_events):
92          """ Evaluate the parameter values at a specific time t,
    considering events. """
93
94          # Initialize parameters with default values
95          params = {
96              "BW": self.BW,
97              "ka1": self.ka1,
98              "ka2": self.ka2,
99              "ka3": self.ka3,
100             "VG": self.VG,
101             "F01": self.F01,
102             "EGP0": self.EGP0,
103             "tmaxI": self.tmaxI,
104             "VI": self.VI,
105             "ke": self.ke,
106             "tmaxG": self.tmaxG,
```

```
            "SIT": self.SIT,
            "SID": self.SID,
            "SIE": self.SIE,
            "k12": self.k12,
            "Gpeq": self.Gpeq,
            "AG": self.AG,
        }

        return parameters_w_events(params, t, file_events)

    def get_delays(self):
        """ Return the insulin and meal delays. """
        return self.i_delay, self.m_delay

    def compute_steady_state(self, u, t):
        """
        Computes the steady state of the glucose-insulin model.

        Args:
        - u: A dictionary containing the model inputs, such as insulin infusion rate (iir).
        - t: Current time to evaluate the model parameters.

        Returns:
        - A dictionary containing the steady-state values for the model variables.
        """
        # Evaluate model parameters at time t
        p = self.evaluate_parameter_values_at(t, self.events_file)
        # Compute steady-state glucose concentration (Gpeq) in mmol/l
        Gpeq = p["Gpeq"] * g.mg_dl_to_mmol_l  # Convert mg/dl to mmol/l
        # Compute non-insulin-dependent glucose flux at steady state (F01eq)
        F01eq = g.nid_glucose_flux_c(p["F01"], p["BW"], Gpeq)
        # Compute insulin amount at steady state (Seq) in mU
        Seq = i.insulin_from_iir(u["iir"], p["tmaxI"])
        # Compute plasma insulin concentration at steady state (Ieq) in mU/l
        Ieq = i.plasma_insulin_from_S(Seq, p["tmaxI"], p["VI"], p["BW"], p["ke"])
        # Compute remote effect of insulin on glucose transport, disposal, and endogenous production in 1/min
        x1eq, x2eq, x3eq = i.insulin_action(p["SIT"], p["SID"], p["SIE"], Ieq)
        # Compute masses of native glucose at steady state in accessible and non-accessible compartments (mmol)
        Q1eq, Q2eq = g.glucose_masses_compartments(F01eq, p["EGP0"], p["BW"], p["k12"], x1eq, x2eq, x3eq)

        # Return a dictionary with all steady-state variable values
        return {
            "Q1": Q1eq,  # Glucose in the accessible compartment
            "Q2": Q2eq,  # Glucose in the non-accessible compartment
            "S1": Seq,   # Insulin in compartment 1
            "S2": Seq,   # Insulin in compartment 2
            "I" : Ieq,   # Plasma insulin concentration
            "x1": x1eq,  # Insulin action on glucose transport
```

```
155             "x2": x2eq,  # Insulin action on glucose disposal
156             "x3": x3eq,  # Insulin action on endogenous glucose
    production
157             "D1": 0,     # Glucose in the gut (compartment 1)
158             "D2": 0      # Glucose in the gut (compartment 2)
159         }
160
161
162     def compute_IIR(self, targetBG, t):
163         """ Compute the insulin infusion rate (IIR) to reach a target
    blood glucose level. """
164         p = self.evaluate_parameter_values_at(t, self.events_file)
165         targetBG = targetBG * g.mg_dl_to_mmol_l # Equilibrium blood
    glucose levels in mmol/l
166         return i.compute_iir(p,targetBG) # U/h
167
168
169     def compute_derivatives(self, x, u, t):
170         """
171         Computes the derivatives of the model variables based on the
    current state,
172         carbohydrate intake and insulin input, and the current time.
173
174         Args:
175         - x: A dictionary containing the current state of the model
    variables.
176         - u: A dictionary containing model inputs, such as
    carbohydrate intake and insulin infusion rate.
177         - t: Current time to evaluate model parameters.
178
179         Returns:
180         - A dictionary containing the derivatives of the model
    variables.
181         """
182         # Evaluate model parameters at time t
183         p = self.evaluate_parameter_values_at(t, self.events_file)
184
185         # Inputs
186         M   = m.meal_from_carb(u.get("carbs", 0))                    #
     Meal intake in mmol/min
187         IIR = u.get("iir", 0) * i.U_h_to_mU_min                      #
     Insulin infusion rate converted to mU/min
188         G   = g.glucose_from_volume(x["Q1"], p["VG"], p["BW"])       #
     Plasma glucose concentration in mmol/l
189
190         F01c = g.nid_glucose_flux_c(p["F01"], p["BW"], G)           #
    Non-insulin-dependent glucose flux in mmol/min
191         FR   = g.renal_flux(G)                                       #
    Renal glucose excretion in mmol/min
192         UG   = m.gut_absorption_rate(x["D2"], p["tmaxG"])           #
    Gut glucose absorption rate in mmol/min
193
194         # Insulin action coefficients, units: 1/min^2/(mU/l)
195         kb1, kb2, kb3 = i.insulin_action_coeffs(p["SIT"], p["SID"], p[
    "SIE"], p["ka1"], p["ka2"], p["ka3"])
196
197         # Compute and return vector of derivatives dx/dt
198         return {
```

```
199             # Insulin subsystem
200             "S1": i.S1_derivative(x["S1"], p["tmaxI"], IIR),
                # Derivative of insulin in compartment 1
201             "S2": i.S2_derivative(x["S1"], x["S2"], p["tmaxI"]),
                # Derivative of insulin in compartment 2
202             "I":  i.I_derivative(x["I"], x["S2"], p["tmaxI"], p["VI"],
     p["BW"], p["ke"]),  # Derivative of plasma insulin concentration
203
204             # Glucose subsystem
205             "Q1": g.Q1_derivative(x["Q1"], x["Q2"], F01c, p["k12"], FR
    , UG, p["EGP0"], p["BW"], x["x1"], x["x3"]), # Derivative of
    glucose in the accessible compartment
206             "Q2": g.Q2_derivative(x["Q1"], x["Q2"], p["k12"], x["x1"],
     x["x2"]),                                    # Derivative of
    glucose in the non-accessible compartment
207             "x1": g.x_derivative(x["x1"], p["ka1"], kb1, x["I"]),
                                                  # Derivative of
    insulin action on transport
208             "x2": g.x_derivative(x["x2"], p["ka2"], kb2, x["I"]),
                                                  # Derivative of
    insulin action on disposal
209             "x3": g.x_derivative(x["x3"], p["ka3"], kb3, x["I"]),
                                                  # Derivative of
    insulin action on endogenous glucose production
210
211             # Meal subsystem
212             "D1": m.D1_derivative(x["D1"], p["tmaxG"], M, p["AG"]),  #
     Derivative of gut glucose in compartment 1
213             "D2": m.D2_derivative(x["D1"], x["D2"], p["tmaxG"])      #
     Derivative of gut glucose in compartment 2
214         }
215
216     def compute_output(self, x, t, activeInsulin):
217         """
218         Computes the plasma glucose concentration (Gp) from the
    current state.
219
220         Args:
221         - x: A dictionary containing the current states of the model
    variables.
222         - t: Current time to evaluate the model parameters.
223         - activeInsulin: Amount of active insulin (typically computed
    externally).
224
225         Returns:
226         - A dictionary containing the plasma glucose concentration.
227         """
228         p = self.evaluate_parameter_values_at(t, self.events_file)
229
230         # Compute plasma glucose concentration (mg/dl) from Q1
231         Gp = g.glucose_from_volume(x["Q1"], p["VG"], p["BW"]) * g.
    mmol_l_to_mg_dl
232
233         return {
234             "Gp": Gp,
235             "activeInsulin": activeInsulin,
236         }
```

Código D.13: cambridge_model.py (DiabetesTool).

D.7.2. events.py

```

import math
from datetime import datetime, timedelta
from inputs.json_data import load_params
from util.auxi import MIN_DAY, MIN_H, S_MIN


def parameters_w_events(params, t, file_events):
    """ Apply the effects of events on the parameters at time t"""
    events = load_params(file_events)

    if events is None:
        return params

    # Calculate how each event affects according to the proximity in time to t
    for event_type, event_list in events.items():
        for event in event_list:
            event_time = datetime.fromisoformat(event["start"])
            # If the event is in the affected time range
            if event_time <= t:

                # Exercise effect
                if event_type == "exercise":
                    params = exercise(params, t, event, event_time)

                # Alcohol effect
                elif event_type == "alcohol":
                    params = alcohol(params, t, event, event_time)

                # Mood effect
                elif event_type == "mood":
                    params = mood(params, t, event, event_time)

                # Illness effect
                elif event_type == "illness":
                    params = illness(params, t, event, event_time)

    return params

def smooth_transition(start_factor, end_factor, elapsed, duration):
    """Apply a soft transition between two values using a sigmoid function."""
    progress = elapsed / duration
    smooth_factor = 1 / (1 + math.exp(-10 * (progress - 0.5)))  # sigmoid adjusted for soft transition
    return start_factor + (end_factor - start_factor) * smooth_factor

```

```
46 def adjust_sensitivity(params, intensity, elapsed, duration, increase=
   True):
47     """Adjust the sensitivity gently according to intensity and
   duration."""
48     progress = elapsed / duration
49     smooth_factor = 1 / (1 + math.exp(-10 * (progress - 0.5)))  #
   sigmoid transition
50     factor = intensity * 0.0001 * (smooth_factor if increase else 1 -
   smooth_factor)
51     params["SIT"] += factor
52     params["SID"] += factor
53 
54 def exercise(params, t, event, event_time):
55     """Apply the effect of exercise on the parameters at time t"""
56     duration = event.get("duration", 0)
57     intensity = event.get("intensity", 5)
58     end_time = event_time + timedelta(minutes=duration)
59     effect_time = end_time + timedelta(minutes=30)
60     sensitivity_duration = int(duration * intensity * 1.2)
61 
62     def handle_sensitivity(start, end, increase):
63         """Handle the sensitivity adjustment based on the elapsed time
   ."""
64         elapsed = (t - start).total_seconds() / S_MIN
65         duration = (end - start).total_seconds() / S_MIN
66         adjust_sensitivity(params, intensity, elapsed, duration,
   increase=increase)
67 
68     if intensity > 5:
69         factor = 0.4*intensity
70         # EGP0 linear increase in the first 5 minutes
71         if event_time <= t <= event_time + timedelta(minutes=5):
72             elapsed = (t - event_time).total_seconds() / S_MIN
73             params["EGP0"] *= smooth_transition(1, factor, elapsed, 5)
74 
75         # EGP0 linear decrease in the 10 minutes after exercise
76         elif event_time + timedelta(minutes=5) < t <= end_time :
77             params["EGP0"] *= factor
78 
79         # EGP0 linear decrease in the 10 minutes after exercise
80         elif end_time < t <= end_time + timedelta(minutes=10):
81             elapsed = (t - end_time).total_seconds() / S_MIN
82             params["EGP0"] *= smooth_transition(factor, 1, elapsed,
   10)
83 
84     # Sensitivity adjustments
85     if effect_time <= t < effect_time + timedelta(minutes=5):
86         handle_sensitivity(effect_time, effect_time + timedelta(
   minutes=5), increase=True)
87     elif effect_time + timedelta(minutes=5) <= t <= end_time +
   timedelta(minutes=sensitivity_duration):
88         adjust_sensitivity(params, intensity, 1, 1)  # Maximum
   sensitivity
89     elif end_time + timedelta(minutes=sensitivity_duration) < t <=
   end_time + timedelta(minutes=sensitivity_duration + 10):
90         handle_sensitivity(end_time + timedelta(minutes=
   sensitivity_duration), end_time + timedelta(minutes=
   sensitivity_duration + 10), increase=False)
```

```
91
92      return params
93
94
95
96  def alcohol(params, t, event, event_time):
97      """Apply the effect of alcohol on the parameters at time t"""
98      quantity = event.get("quantity", 1)
99      alcohol_duration = int(1 * MIN_DAY if quantity >= 6 else quantity
    * 4 * MIN_H)
100     # Define the reduction factor based on the amount of alcohol
101     reduction_factor = 0.7 if quantity >= 6 else (1 - 0.05 * quantity)
102     start_time = event_time
103     end_time = start_time + timedelta(minutes=alcohol_duration)
104     # Linear growth in the first 5 minutes to the reduction value
105     if start_time <= t <= start_time + timedelta(minutes=15):
106         elapsed = (t - start_time).total_seconds() / S_MIN
107         params["EGP0"] *= smooth_transition(1, reduction_factor,
    elapsed, 15)
108
109     # Reduction maintained during the main effect
110     elif start_time + timedelta(minutes=15) <= t <= end_time:
111         params["EGP0"] *= reduction_factor
112
113     # Linear decrease to restore the original value in the 10 minutes
    after the effect
114     elif end_time < t <= end_time + timedelta(minutes=25):
115         elapsed = (t - end_time).total_seconds() / S_MIN
116         params["EGP0"] *= smooth_transition(reduction_factor, 1,
    elapsed, 25)
117
118     return params
119
120 def mood(params, t, event, event_time):
121     """Apply the effect of mood on the parameters at time t"""
122     duration = event.get("duration", 0)
123     mood_type = event.get("type", "Stress")
124
125     start_time = event_time
126     end_time = start_time + timedelta(minutes=duration)
127
128     # Adjustment factors according to the type of mood
129     if mood_type == "Stress":
130         egp_factor = 1.8
131         sie_factor = 0.9
132     elif mood_type == "Relaxation":
133         sit_factor = sie_factor = 1.1
134
135     # Initial adjustment (5 minutes of linear increase)
136     if start_time <= t <= start_time + timedelta(minutes=5):
137         elapsed = (t - start_time).total_seconds() / S_MIN
138         if mood_type == "Stress":
139             params["EGP0"] *= smooth_transition(1, egp_factor, elapsed
    , 5)
140             params["SIE"]  *= smooth_transition(1, sie_factor, elapsed
    , 5)
141         elif mood_type == "Relaxation":
```

```
142             params["SIT"]  *= smooth_transition(1, sit_factor, elapsed
    , 5)
143             params["SIE"]  *= smooth_transition(1, sie_factor, elapsed
    , 5)
144
145     # Maintain the effect during the duration of the event
146     elif start_time + timedelta(minutes=5) < t <= end_time:
147         if mood_type == "Stress":
148             params["EGP0"] *= egp_factor
149             params["SIE"]  *= sie_factor
150         elif mood_type == "Relaxation":
151             params["SIT"]  *= sit_factor
152             params["SIE"]  *= sie_factor
153
154     # Restore the original value in the 10 minutes after the event
155     elif end_time < t <= end_time + timedelta(minutes=10):
156         elapsed = (t - end_time).total_seconds() / S_MIN
157         if mood_type == "Stress":
158             params["EGP0"] *= smooth_transition(egp_factor, 1, elapsed
    , 10)
159             params["SIE"]  *= smooth_transition(sie_factor, 1, elapsed
    , 10)
160         elif mood_type == "Relaxation":
161             params["SIT"]  *= smooth_transition(sit_factor, 1, elapsed
    , 10)
162             params["SIE"]  *= smooth_transition(sie_factor, 1, elapsed
    , 10)
163
164     return params
165
166
167 def illness(params, t, event, event_time):
168     """Apply the effect of illness on the parameters at time t"""
169     duration = event.get("duration", 0)
170     illness_type = event.get("type", "Infection")
171
172     start_time = event_time
173     end_time = start_time + timedelta(minutes=duration)
174
175     # Adjustment factors according to the type of disease
176     if illness_type == "Infection":
177         egp_factor = 1.3
178         sit_sid_factor = 0.85
179     elif illness_type == "Fever":
180         egp_factor = 1.5
181         sit_sid_factor = 0.8
182     elif illness_type == "Dehydration":
183         vg_factor = 0.95
184
185     # Initial increase (5 minutes of linear increase)
186     if start_time <= t <= start_time + timedelta(minutes=5):
187         elapsed = (t - start_time).total_seconds() / S_MIN
188         if illness_type == "Infection" or illness_type == "Fever":
189             params["EGP0"] *= smooth_transition(1, egp_factor,
    elapsed, 5)
190             params["SIT"]  *= smooth_transition(1, sit_sid_factor,
    elapsed, 5)
```

```
191             params["SID"]  *= smooth_transition(1, sit_sid_factor,
    elapsed, 5)
192         elif illness_type == "Dehydration":
193             params["VG"]   *= smooth_transition(1, vg_factor, elapsed,
     5)
194
195     # Maintain the effect during the duration of the event
196     elif start_time + timedelta(minutes=5) < t <= end_time:
197         if illness_type == "Infection" or illness_type == "Fever":
198             params["EGP0"] *= egp_factor
199             params["SIT"]  *= sit_sid_factor
200             params["SID"]  *= sit_sid_factor
201         elif illness_type == "Dehydration":
202             params["VG"]   *= vg_factor
203
204     # Restore the original value in the 10 minutes after the event
205     elif end_time < t <= end_time + timedelta(minutes=10):
206         elapsed = (t - end_time).total_seconds() / S_MIN
207         if illness_type == "Infection" or illness_type == "Fever":
208             params["EGP0"] *= smooth_transition(egp_factor,     1,
    elapsed, 10)
209             params["SIT"]  *= smooth_transition(sit_sid_factor, 1,
    elapsed, 10)
210             params["SID"]  *= smooth_transition(sit_sid_factor, 1,
    elapsed, 10)
211         elif illness_type == "Dehydration":
212             params["VG"]   *= smooth_transition(vg_factor,      1,
    elapsed, 10)
213
214     return params
```

Código D.14: events.py (DiabetesTool).

D.7.3. glucose.py

```
1
2   mmol_l_to_mg_dl = 18.016  # Conversion factor from mmol/l to mg/dl
3   mg_dl_to_mmol_l = 1 / mmol_l_to_mg_dl # Conversion factor from mg/dl
        to mmol/l
4
5
6   def nid_glucose_flux_c(F01, BW, G): # mmol/min
7       """
8       Total non-insulin-dependent glucose flux corrected for the ambient
        glucose concentration(mmol/min)
9       Eq. (2) R Hovorka et al 2004
10      """
11      return F01 * BW * min(G / 4.5, 1)
12
13  def glucose_masses_compartments(F01, EGP0, BW, k12, x1, x2, x3): #
        mmol
14      """
15          Computes steady-state glucose masses in compartments Q1 and Q2
        based on insulin action and fluxes.
16
```

```
        "x1": "insulin action on glucose transport",            1/min
        "x2": "insulin action on glucose disposal",             1/min
        "x3": "insulin action on endogenous glucose production",1
        "k12": "transfer rate from the non-accessible to the accessible compartment", 1/min
        "EGP0": "endogenous glucose production (EGP) extrapolated to the zero insulin concentration", mmol/kg/min
        "F01": "non-insulin-dependent glucose flux, mmol/min
        "Q1": "mass in accessible compartment (plasma)",
        "Q2": "mass in non-accessible compartment",
    """
    glucose = glucose_w_insulin_F01(F01, EGP0, BW, x3)
    if x2 == 0:
        x2 = 1e-10  # Assign a very small value
    q2 = glucose / x2
    if q2 <= 0:
        raise ValueError("Q2eq must be greater than 0")

    transported_glucose = glucose_out_Q2(q2, k12, x2)
    if x1 == 0:
        x1 = 1e-10  # Assign a very small value
    q1 = transported_glucose / x1

    return q1, q2

def glucose_from_volume(Q1, VG, BW): # mmol/l
    """ Calculates plasma glucose concentration from total mass in accessible compartment. """
    gdv_l = VG * BW
    g = Q1 / gdv_l
    return g

def renal_flux(G): # mmol /min
     """ Models renal glucose excretion when plasma glucose exceeds 9 mmol/L """
     return max(0, 0.003 * (G - 9))

def x_derivative(x,ka,kb,I):
     """ Computes the derivative of the insulin action variable x (x1, x2, or x3) """
     return I*kb - x*ka

def Q1_derivative(Q1, Q2, F01c, k12, renal_flux, gut_abs, EGP0, BW, x1, x3): # mmol/min
    """ Computes the rate of change of glucose in the accessible compartment (Q1) """
    glucose = glucose_w_insulin_F01(F01c, EGP0, BW, x3)
    glucose_from_Q2 = k12 * Q2
    glucose_out_Q1 = x1 * Q1
    return  glucose_from_Q2 - glucose_out_Q1 + glucose - renal_flux + gut_abs

def Q2_derivative(Q1, Q2, k12, x1, x2): # mmol/min
    """ """
    glucose_out = glucose_out_Q2(Q2, k12, x2)
    glucose_from_Q1 = x1*Q1
    return glucose_from_Q1 - glucose_out

```

```
67 def glucose_w_insulin_F01(F01, EGP0, BW, x3): # mmol / min
68     """ Computes the rate of change of glucose in the non-accessible
       compartment (Q2) """
69     egp0 = EGP0 * BW
70     glucose_w_insulin = egp0 * (1 - x3)
71     glucose = glucose_w_insulin - F01
72     return glucose
73 
74 def glucose_out_Q2(Q2, k12, x2): # mmol / min
75     """ Computes endogenous glucose production (EGP) suppressed by
       insulin, minus non-insulin-dependent utilization """
76     transf_rate = k12 + x2
77     glucose_outQ2 = Q2 * transf_rate # mmol/min
78     return glucose_outQ2
```

Código D.15: glucose.py (DiabetesTool).

D.7.4. nsulin.py

```
1  import math
2  from model.glucose import nid_glucose_flux_c
3  
4  U_h_to_mU_min = 1000 / 60
5  mU_min_to_U_h = 1 / U_h_to_mU_min
6  
7  def insulin_from_iir(iir, tmaxI):  # mU
8      """
9      Infulin from insulin infusion rate in mU
10     Args:
11     - iir: Infuline infusion rate in U/h
12     - tmaxI: time-to-maximum of absorption of subcutaneously injected
       short-acting insulin in min
13     Returns:
14     - S: Insulin in mU
15     """
16     return (iir or 0) * U_h_to_mU_min * tmaxI
17 
18 def iir_from_insulin(S, tmaxI): # U / h
19     """ Converts insulin amount into infusion rate based on absorption
       time """
20     return S / tmaxI * mU_min_to_U_h
21 
22 def insulin_from_VI(VI, BW, ke, tmaxI, I): # mU
23     """ Calculates insulin amount from plasma insulin concentration
       using volume and clearance """
24     vi_l = VI* BW                # Distribution volume of the
       accessible compartment
25     insulin = I * vi_l           # Plasma insulin in mU
26     s = tmaxI * insulin * ke
27     return s
28 
29 def plasma_insulin_from_S(S, tmaxI, VI, BW, ke): # mU / l
30     """ Calculates plasma insulin concentration from total insulin (S)
       using volume and elimination. """
31     idv_l = VI * BW              # insulin distribution volume in l
```

```
32      iv_l  = idv_l * tmaxI * ke  # insulin volume in l considering the
    Insulin elimination from plasma and time-to-maximum of absorption
    of subcut. injected short-acting insulin
33      return S / iv_l
34
35  def insulin_action(SIT, SID, SIE, I):
36      """
37      Computes insulin action on transport (x1), disposal (x2), and
    suppression of EGP (x3).
38      Units: x1 and x2 in 1/min, x3 unitless.
39
40      "x1": "insulin action on glucose transport",            1/min
41      "x2": "insulin action on glucose disposal",             1/min
42      "x3": "insulin action on endogenous glucose production",1
43      "SIT": "insulin sensitivity of distribution/transport", l/min/mU
44      "SID": "insulin sensitivity of disposal",               l/min/mU
45      "SIE": "insulin sensitivity of EGP"                     l    /mU
46      """
47      return SIT*I, SID*I,SIE*I
48
49  def insulin_action_coeffs(SIT, SID, SIE, ka1, ka2, ka3):
50      """ Calculates sensitivity coefficients for insulin action
    differential equations """
51      return SIT*ka1, SID*ka2, SIE*ka3
52
53  def compute_iir(params, Gpeq):  # mU / h
54      """
55      Computes the insulin infusion rate required to maintain steady
    glucose (Gpeq).
56      Based on Hovorka model equilibrium equations.
57      """
58      # Compute glucose mass in the accessible compartment (Q1eq)
59      Q1eq = Gpeq * params["VG"] * params["BW"]
60      # Compute steady-state non-insulin-dependent glucose flux (F01eq)
61      F01eq = nid_glucose_flux_c(params["F01"], params["BW"], Gpeq)
62      # Compute steady-state endogenous glucose production (EGP0)
63      egp0 = params["EGP0"] * params["BW"]
64      # Solve the quadratic equation to find plasma insulin at
    equilibrium (Ieq)
65      Ieq = plasma_insulin_from_model(Q1eq, params["SIT"], params["SID"
    ], params["SIE"], egp0, F01eq, params["k12"])
66      # Compute steady-state insulin amount (Seq) in mU
67      Seq = insulin_from_VI(params["VI"], params["BW"], params["ke"],
    params["tmaxI"], Ieq)
68      # Return adjusted IIR in U/h
69      return iir_from_insulin(Seq,  params["tmaxI"])
70
71  def plasma_insulin_from_model(Q1, SIT, SID, SIE, EGP0, F01, k12):
72      """ Solves a quadratic equation to compute steady-state plasma
    insulin based on model inputs """
73      # Coefficients for the quadratic equation
74      # aI^2 + bI + c = 0
75      a = -Q1 * SIT * SID - EGP0 * SID * SIE
76      b = -F01 * SID - k12 * Q1 * SIT + k12 * SIT * Q1 + EGP0 * SID -
    EGP0 * k12 * SIE
77      c = -F01 * k12 + EGP0 * k12
78      I = (-b - math.sqrt(b * b - 4 * a * c)) / (2 * a)
79      return I
```

```
80
81 def S1_derivative(S1, tmaxI, iir): # mU/min
82     """ Computes rate of change of insulin in subcutaneous compartment
       1 """
83     s_in  = iir
84     s_out = S1 / tmaxI
85     return s_in - s_out
86
87 def S2_derivative(S1, S2, tmaxI):# mU/min
88     """ Computes rate of change of insulin in subcutaneous compartment
       2 """
89     s_in  = S1 / tmaxI
90     s_out = S2 / tmaxI
91     return s_in - s_out
92
93 def I_derivative(I, S2, tmaxI, VI, BW, ke): # mU/min/l
94     """ Computes rate of change of plasma insulin concentration """
95     volume = VI * BW
96     i_in  = S2 / tmaxI / volume
97     i_out = I * ke
98     return i_in - i_out
```

Código D.16: insulin.py (DiabetesTool).

D.7.5. meal.py

```
1 mmol_to_mg = 180.16
2 mg_to_mmol = 1 / mmol_to_mg
3
4 def meal_from_carb(carbs):  # mmol/min
5     """Converts carbohydrate intake (g) to mmol/min."""
6     carbs_mg_min = carbs * 1000  # Carbs from g to mg
7     carbs_mmol_min = carbs_mg_min * mg_to_mmol
8     return carbs_mmol_min
9
10 def gut_absorption_rate(D, tmaxG):  # mmol/min
11     """Computes glucose absorption rate from gut compartment."""
12     return D / tmaxG
13
14 def D1_derivative(D1, tmaxG, meal, AG):  # mmol/min
15     """Computes the rate of change of glucose in gut compartment 1."""
16     return meal * AG - gut_absorption_rate(D1, tmaxG)
17
18 def D2_derivative(D1, D2, tmaxG):  # mmol/min
19     """Computes the rate of change of glucose in gut compartment 2."""
20     return gut_absorption_rate(D1, tmaxG) - gut_absorption_rate(D2,
       tmaxG)
```

Código D.17: meal.py (DiabetesTool).

D.8. programs

D.8.1. comparator_main.py

```
import logging as log
from datetime import datetime, timedelta
from carelink.data import init_client, download_data, simple_to_inputs
from inputs.json_data import  load_data_from_json
from carelink.json_carelink import delete_json, load_json
from model.cambridge_model import Cambridge
from simulator.simulator import Simulator
from simulator.solver import SolverRK1_2
from util.glucose_plot import plot_compare_ins, plot_compare
from util.auxi import filter_datetime, make_naive, get_closest_sg
import argparse
from util.auxi import FORMAT, filter_all
from util.residuals import evaluate_comparison_metrics

# Logging settings
log.basicConfig(format=FORMAT, datefmt='%Y-%m-%d %H:%M:%S', level=log.INFO)

def main(start_time=None,duration=None,input_file=None, ref_ins_file=None, ins_file=None, compare_insulin=False, plot=False):
    """ Main function to run the simulator and compare results with Carelink data."""

    def dw_data():
        """Download data from Carelink and return the filenames."""
        # Initialize Carelink client and download real data
        client = init_client(verbose=False)
        if not client:
            log.error("Carelink client could not be initialized.")
            return
        #Download recent data and simplify them
        filename_simple, dw_filename = download_data(client, save_data=False, repeat=1, wait=0)
        if not filename_simple or not dw_filename:
            log.error("Couldn't download recent data.")
            return
        return filename_simple, dw_filename

    download = False

    if start_time is None:
        # Command line analysis
        parser = argparse.ArgumentParser()
        parser.add_argument('--start-date', '-s', type=str, help='Format start date YYYY-MM-DDTHH:MM:SS', required=False)
        parser.add_argument('--file', '-f', type=str, help='Simplified JSON file name', required=False)
        args = parser.parse_args()

        if args.file:
            filename_simple = args.file
            log.info(f"Using file data {filename_simple}.")
```

```
47
48          else:
49              filename_simple, dw_filename = dw_data()
50              download = True
51
52      else:
53          start_time = make_naive(start_time)
54          if input_file == "" and ref_ins_file is None:
55              filename_simple, dw_filename = dw_data()
56              download = True
57          else:
58              if compare_insulin:
59                  if ref_ins_file =="":
60                      filename_simple, dw_filename = dw_data()
61                      download = True
62                  else:
63                      filename_simple = ref_ins_file
64              else:
65                  filename_simple = input_file
66
67      if filename_simple is None:
68          log.error("The token could not be obtained.")
69          return
70
71      # Read simplified data
72      data = load_json(filename_simple)
73
74      # Set the simulation from the first data of SG
75      first_sg_time = datetime.fromisoformat(data['sgs'][0]['datetime'])
76
77      # Get the start date
78      if start_time is None:
79          if args.start_date:
80              start_time =  make_naive(datetime.fromisoformat(args.
    start_date))
81          else:
82              # Default date based on the first SGS record
83              start_time =  make_naive(first_sg_time)
84
85      # Calculate the duration in minutes from start_time until now
86      if duration is None or duration == "":
87          end_time = make_naive(datetime.fromisoformat(data['sgs'][-1]['
    datetime']))
88          duration = int((end_time - start_time).total_seconds() / 60)
89      else:
90          end_time = make_naive(start_time + timedelta(minutes=int(
    duration)))
91
92      # Parsesar meals, insulin and basal
93      if download:
94          meals, bolus, basal = load_data_from_json(dw_filename)
95          i_file = dw_filename
96      else:
97          i_file = 'data/inputs/comparator_inputs-temp.json'
98          simple_to_inputs(filename_simple, i_file)
99          # Load input data for simulation
100         meals, bolus, basal = load_data_from_json(i_file)
101
```

```
102     if compare_insulin:
103 
104         def insulin_comparison(basal_ref, bolus_ref, basal_compare,
    bolus_compare):
105             # --- Meal Boluses ---
106             bolus_ref_used = set()
107             log.info("=== Meal Boluses Comparison ===")
108             for meal in [b for b in bolus_compare if b.get("type") ==
    "Meal"]:
109                 closest_ref = None
110                 min_diff = timedelta.max
111                 for i, ref in enumerate(bolus_ref):
112                     if i in bolus_ref_used:
113                         continue
114                     diff = abs(ref["start"] - meal["start"])
115                     if diff < min_diff:
116                         min_diff = diff
117                         closest_ref = ref_index = i
118                 if closest_ref is not None:
119                     bolus_ref_used.add(ref_index)
120                     log.info(f"Meal at {meal['start']} -- Compare: {
    meal['i']} U vs Ref: {bolus_ref[ref_index]['i']} U (delta t = {
    min_diff})")
121                 else:
122                     log.info(f"Meal at {meal['start']} -- Compare: {
    meal['i']} U, no matching Ref found.")
123 
124             # --- Correction Boluses ---
125             compare_corr = sum(b['i'] for b in bolus_compare if b.get(
    "type") == "Correction")
126             ref_corr = sum(b['i'] for i, b in enumerate(bolus_ref) if
    i not in bolus_ref_used)
127 
128             log.info("=== Correction Boluses Comparison ===")
129             log.info(f"Total Correction -- Compare: {compare_corr:.2f}
     U vs Ref: {ref_corr:.2f} U")
130 
131             # --- Basal ---
132             total_basal_ref = sum(b['iir'] * b['duration'] / 60.0 for
    b in basal_ref)
133             total_basal_compare = sum(b['iir'] * b['duration'] / 60.0
    for b in basal_compare)
134             log.info("=== Basal Comparison ===")
135             log.info(f"Total Basal      -- Compare: {
    total_basal_compare:.2f} U vs Ref: {total_basal_ref:.2f} U")
136 
137 
138         _, bolus_compare, basal_compare = load_data_from_json(ins_file
    )
139         basal_ref = filter_datetime(start_time, end_time, basal,'start
    ')
140         bolus_ref = filter_datetime(start_time, end_time, bolus,'start
    ')
141 
142         fig = plot_compare_ins(filename_simple, basal_compare,
    bolus_compare, start_time, end_time, plot=False)
143 
```

```
144             insulin_comparison(basal_ref, bolus_ref, basal_compare,
    bolus_compare)
145
146         else:
147
148             # Create the Cambridge model
149             model = Cambridge()
150             # Solver
151             solver = SolverRK1_2()
152
153             init_g = get_closest_sg(data['sgs'], start_time)
154             # Create the simulator and configure meals
155             simulator = Simulator(model, solver=solver, duration=duration,
     time_step=1, i_delay = model.i_delay, m_delay = model.m_delay,
    initial_g = init_g)
156
157             simulator.set_input_file(i_file)
158
159             # Execute the simulation
160             results = simulator.run_simulation(start_time, rts=False,
    print_output=False)
161
162             delete_json(i_file)
163             results, meals, basal, bolus = filter_all(results, meals,
    basal, bolus, start_time, end_time)
164             fig = plot_compare(results, meals, basal, bolus,
    filename_simple, start_time, end_time, plot)
165             evaluate_comparison_metrics(results, filename_simple,
    start_time, end_time)
166
167         return fig
168
169 if __name__ == "__main__":
170     main()
```

Código D.18: comparator_main.py (DiabetesTool).

D.8.2. download_main.py

```
1  import argparse
2  import logging as log
3  from carelink.data import init_client, download_data
4  from util.auxi import FORMAT
5  from util.glucose_plot import plot_data
6
7  VERSION = "1.0"
8
9  # Logging settings
10 log.basicConfig(format=FORMAT, datefmt='%Y-%m-%d %H:%M:%S', level=log.
       INFO)
11
12 def main(repeat_times=None, wait_times=None, save_data=None, verbose=
       None, plot = False):
13     """ Main function to download data from Carelink and plot it. """
14
```

```
    if repeat_times is None or wait_times is None or save_data is None
    or verbose is None:
      parser = argparse.ArgumentParser()
      parser.add_argument('--repeat', '-r', type=int, help='Request
   repetitions', required=False)
      parser.add_argument('--wait', '-w', type=int, help='Wait minutes
    between repeated calls', required=False)
      parser.add_argument('--data', '-d', help='Save recent data',
   action='store_true')
      parser.add_argument('--verbose', '-v', help='Detailed mode (
   verbose)', action='store_true')
      args = parser.parse_args()
      repeat = 1 if args.repeat is None else args.repeat
      wait = 5 if args.wait is None else args.wait
      save_data = args.data
      verbose = args.verbose
    else:
      repeat = repeat_times
      wait = wait_times
      save_data = save_data
      verbose = verbose

    # Initialize Carelink client
    client = init_client(verbose)

    # If the client was correctly initialized, download data
    if client:
        filename_simple, dw_file = download_data(client, save_data=
   save_data, repeat=repeat, wait=wait)
        fig = plot_data(filename_simple, plot)
        return fig

if __name__ == "__main__":
    main()
```

Código D.19: download_main.py (DiabetesTool).

D.8.3. estimator_main.py

```
import logging as log
from datetime import datetime, timedelta
from carelink.data import init_client, download_data, simple_to_inputs
from inputs.json_data import load_data_from_json
from carelink.json_carelink import delete_json, load_json
from model.cambridge_model import Cambridge
from simulator.solver import SolverRK1_2
from util.glucose_plot import plot_compare
import argparse
from util.residuals import evaluate_comparison_metrics,
   optimize_parameter, plot_residuals
from util.auxi import make_naive
from util.auxi import FORMAT, filter_all


# Logging settings
```

```
16 log.basicConfig(format=FORMAT, datefmt='%Y-%m-%d %H:%M:%S', level=log.
   INFO)
17
18 def main(start_date=None,duration=None,parameter=None,vx=None,vm=None,
   iterations=None,
19         file=None, input_file=None, events_file=None, params_file=
   None, plot = False):
20     """ Main function to run the simulator, compare results with
   Carelink data and estimate parameters."""
21
22     if start_date is None or duration is None or parameter is None or
   vx is None or vm is None or iterations is None or iterations is
   None:
23         parser = argparse.ArgumentParser()
24         parser.add_argument('--start_date', '-s', type=str, help='
   Format start date YYYY-MM-DDTHH:MM:SS', required=True)
25         parser.add_argument('--duration', '-d', type=int, help='
   Duration in minutes', required=False)
26         parser.add_argument('--end_date', '-e', type=str, help='Final
   Date in Format YYYY-MM-DDTHH:MM:SS', required=False)
27         parser.add_argument('--parameter', '-p', type=str, help='Name
   of the parameter to be estimated', required=True)
28         parser.add_argument('--vx','-vx', type=float, help='Maximum
   parameter value', required=True)
29         parser.add_argument('--vm', '-vm', type=float, help='Minimum
   parameter value', required=True)
30         parser.add_argument('--iterations', '-i', type=int, help='
   Maximum number of iterations', required=True)
31         parser.add_argument('--dfile', '-df', type=str, help='Data
   JSON Simplified file name', required=False)
32         parser.add_argument('--ifile', '-if', type=str, help='Input
   Json file name', required=False)
33         parser.add_argument('--efile', '-ef', type=str, help='Events
   Json file name ', required=False)
34         parser.add_argument('--pfile', '-pf', type=str, help='Params
   Json file name ', required=False)
35         args = parser.parse_args()
36
37
38         if args.efile:
39             e_file = args.efile
40             log.info(f"Using file inputs {e_file}.")
41         else:
42             e_file = 'data/events/events.json'
43         if args.pfile:
44             p_file = args.pfile
45             log.info(f"Using file inputs {p_file}.")
46         else:
47             p_file = 'data/params/params.json'
48
49         if args.dfile and args.ifile:
50             filename_simple = args.dfile
51             i_file = args.ifile
52             log.info(f"Using file data {filename_simple}.")
53             log.info(f"Using file inputs {i_file}.")
54             i_from_data = False
55         else:
56
```

```
57              if args.dfile:
58                  filename_simple = args.dfile
59                  log.info(f"Using file data {filename_simple}.")
60                  i_file = 'data/inputs/comparator_inputs-temp.json'
61                  i_from_data = True
62              else:
63
64                  # Initialize Carelink client and download real data
65                  client = init_client(verbose=False)
66                  if not client:
67                      log.error("Carelink client could not be
    initialized.")
68                      return
69                  filename_simple, dw_filename = download_data(client,
    save_data=False, repeat=1, wait=0)
70                  if not filename_simple:
71                      log.error("Recent data could not be downloaded.")
72                      return
73
74                  if args.ifile:
75                      i_file = args.ifile
76                      log.info(f"Using file inputs {i_file}.")
77                      i_from_data = False
78                  else:
79                      i_from_data = True
80                      i_file = dw_filename
81
82
83          start_time = make_naive(datetime.fromisoformat(args.start_date
    ))
84          parameter=args.parameter
85          vx=args.vx
86          vm=args.vm
87          iterations=args.iterations
88
89          if args.duration:
90              duration = args.duration
91              end_time = start_time + timedelta(minutes=duration)
92          else:
93              end_time = make_naive(datetime.fromisoformat(args.end_date
    )) if args.end_date else make_naive(datetime.now())
94              duration = int((end_time - start_time).total_seconds() /
    60)
95
96       else:
97          start_time= make_naive(start_date)
98          duration=duration
99          end_time = start_time + timedelta(minutes=duration)
100         parameter=parameter
101         vx=vx
102         vm=vm
103         iterations=iterations
104
105         if events_file is not None and events_file != "":
106             e_file = events_file
107             log.info(f"Using file inputs {e_file}.")
108         else:
109             e_file = 'data/events/events.json'
```

```
110         if params_file is not None and params_file != "":
111             p_file = params_file
112             log.info(f"Using file inputs {p_file}.")
113         else:
114             p_file = 'data/params/params.json'
115 
116         if (file is not None and file != "") and (input_file is not
    None and input_file != ""):
117             filename_simple = file
118             i_file = input_file
119             log.info(f"Using file data {filename_simple}.")
120             log.info(f"Using file inputs {i_file}.")
121             i_from_data = False
122         else:
123 
124             if file is not None and file != "":
125                 filename_simple = file
126                 log.info(f"Using file data {filename_simple}.")
127                 i_file = 'data/inputs/estimator_inputs-temp.json'
128                 i_from_data = True
129             else:
130                 client = init_client(verbose=False)
131                 if not client:
132                     log.error("Carelink client could not be
    initialized.")
133                     return
134                 filename_simple, dw_filename = download_data(client,
    save_data=False, repeat=1, wait=0)
135                 if not filename_simple:
136                     log.error("Recent data could not be downloaded.")
137                     return
138                 if input_file is not None and input_file != "":
139                     i_file = input_file
140                     log.info(f"Using file inputs {i_file}.")
141                     i_from_data = False
142                 else:
143                     i_file = dw_filename
144                     i_from_data = True
145 
146 
147     # Read simplified data
148     data = load_json(filename_simple)
149 
150     # Parsesar foods, insulin and basal
151     if i_from_data:
152         i_file = 'data/inputs/estimator_inputs-temp.json'
153         simple_to_inputs(filename_simple, i_file)
154         delete = True
155     else:
156         delete = False
157 
158     # Load input data for simulation
159     meals, bolus, basal = load_data_from_json(i_file)
160 
161 
162     # Create the Cambridge model
163     model = Cambridge(params_file=p_file, events_file=e_file)
164     # Solver
```

```
165     solver = SolverRK1_2()
166 
167     # Optimize the parameter
168     best_value, best_residuals, best_results, best_times =
    optimize_parameter(
169         start_time = start_time,
170         model=model,
171         solver=solver,
172         parameter=parameter,
173         duration = duration,
174         vx=vx,
175         vm=vm,
176         max_iter=iterations,
177         real_data=data['sgs'],
178         i_delay = model.i_delay, m_delay = model.m_delay,
179         i_file = i_file, e_file=e_file
180     )
181 
182     log.info(f"The best value of {parameter} is: {best_value}")
183 
184     best_results, meals, basal, bolus = filter_all(best_results, meals
    , basal, bolus, start_time, end_time)
185 
186     best_times = [t for t in best_times if start_time <= t <= end_time
    ]
187     best_residuals = best_residuals[:len(best_times)]
188 
189     fig1 = plot_residuals(best_residuals, best_times, plot)
190     fig2 = plot_compare(best_results, meals, basal, bolus,
    filename_simple, start_time, end_time, plot)
191 
192     # Evaluate the comparison metrics
193     evaluate_comparison_metrics(best_results, filename_simple,
    start_time, end_time)
194 
195     if delete:
196         delete_json(i_file)
197 
198     return fig1, fig2
199 
200 if __name__ == "__main__":
201     main()
```

Código D.20: estimator_main.py (DiabetesTool).

D.8.4. pump_main.py

```
1 from datetime import datetime
2 from carelink.pump import Pump
3 from util.auxi import make_naive
4 import logging as log
5 import argparse
6 
7 def main(start_time=None, duration=None, plot = False, canvas = None,
```

```
8           input_file=None, events_file=None, params_file=None,
    data_file=None,
9           pump_file = None, speed = 1):
10      """ Main function to run the pump simulation."""
11      if start_time is None:
12          parser = argparse.ArgumentParser()
13          parser.add_argument('--start_time', '-s', type=str, help='
    Format start date YYYY-MM-DDTHH:MM:SS', required=True)
14          parser.add_argument('--duration', '-d', type=int, help='
    Duration in minutes', required=False)
15          parser.add_argument('--ifile', '-if', type=str, help='Input
    Json file name ', required=False)
16          parser.add_argument('--efile', '-ef', type=str, help='Events
    Json file name ', required=False)
17          parser.add_argument('--pfile', '-pf', type=str, help='Params
    Json file name ', required=False)
18          parser.add_argument('--dfile', '-df', type=str, help='Data
    Json file name ', required=False)
19          parser.add_argument('--pufile', '-puf', type=str, help='Name
    of the Pump Input Data Json file ', required=False)
20
21
22          args = parser.parse_args()
23          start_time = make_naive(datetime.fromisoformat(args.start_time
    ))
24          rts = True
25          if args.duration is None:
26              duration = 60*24
27          if args.pufile:
28              pu_file = args.pufile
29              log.info(f"Using file pump data {pu_file}.")
30          else:
31              pu_file = None
32          if args.ifile:
33              i_file = args.ifile
34              log.info(f"Using file inputs {i_file}.")
35          else:
36              i_file = 'data/inputs/inputs.json'
37          if args.efile:
38              e_file = args.efile
39              log.info(f"Using file inputs {e_file}.")
40          else:
41              e_file = 'data/events/events.json'
42          if args.pfile:
43              p_file = args.pfile
44              log.info(f"Using file inputs {p_file}.")
45          else:
46              p_file = 'data/params/params.json'
47          if args.dfile:
48              d_file = args.dfile
49              log.info(f"Write output data in file {d_file}.")
50          else:
51              d_file = 'data/data/data.json'
52
53      else:
54          start_time = make_naive(start_time)
55          rts        = True
56          if duration is None:
```

```
57              duration = 60*24
58          if pump_file is not None and pump_file != "":
59              pu_file = pump_file
60              log.info(f"Using file pump data {pu_file}.")
61          else:
62              pu_file = None
63          if input_file is not None and input_file != "":
64              i_file = input_file
65              log.info(f"Using file inputs {i_file}.")
66          else:
67              i_file = 'data/inputs/inputs.json'
68          if events_file is not None and events_file != "":
69              e_file = events_file
70              log.info(f"Using file inputs {e_file}.")
71          else:
72              e_file = 'data/events/events.json'
73          if params_file is not None and params_file != "":
74              p_file = params_file
75              log.info(f"Using file inputs {p_file}.")
76          else:
77              p_file = 'data/params/params.json'
78          if data_file is not None and data_file != "":
79              d_file = data_file
80              log.info(f"Write output data in file {d_file}.")
81          else:
82              d_file = 'data/data/data.json'
83
84
85      pump = Pump(duration, input_file=i_file, events_file=e_file,
        params_file=p_file, data_file=d_file, pump_file=pu_file, speed=
        speed)
86      pump.run_pump_data_in_thread(start_time, plot=plot, canvas =
        canvas)
87
88      return
89
90
91  if __name__ == "__main__":
92      main()
```

Código D.21: pump_main.py (DiabetesTool).

D.8.5. simulator_main.py

```
1  from datetime import datetime, timedelta
2  from model.cambridge_model import Cambridge
3  from simulator.simulator import Simulator, simulate_param_i
4  from simulator.solver import SolverRK1_2
5  from util.auxi import filter_all, make_naive, str2bool
6  from util.glucose_plot import carb_intake_function,
       insulin_intake_function, plot_glucose,insulin_bolus_bars,
       plot_parameters_init, plot_state_init, plot_state_variables,
       plot_parameters, plot_init
7  from inputs.json_data import load_data_from_json
8  import logging as log
```

```
import argparse

def main(start_time=None, rts=None, duration=None,parameter=None,vx=None,vm=None,
         iterations=None, plot = False, canvas = None, input_file=None, events_file=None, params_file=None, data_file=None, speed = 1):

     if start_time is None or rts is None:
          parser = argparse.ArgumentParser()
          parser.add_argument('--start_time', '-s', type=str, help='Format start date YYYY-MM-DDTHH:MM:SS', required=True)
          parser.add_argument('--rts', '-r', type=str2bool, help='Real - time mode')
          parser.add_argument('--duration', '-d', type=int, help='Duration in minutes', required=False)
          parser.add_argument('--parameter', '-p', type=str, help='Name of the parameter to be estimated', required=False)
          parser.add_argument('--vx','-vx', type=float, help='Maximum parameter value', required=False)
          parser.add_argument('--vm', '-vm', type=float, help='Minimum parameter value', required=False)
          parser.add_argument('--iterations', '-i', type=int, help='Maximum number of iterations', required=False)
          parser.add_argument('--ifile', '-if', type=str, help='Input Json file name ', required=False)
          parser.add_argument('--efile', '-ef', type=str, help='Events Json file name ', required=False)
          parser.add_argument('--pfile', '-pf', type=str, help='Params Json file name', required=False)
          parser.add_argument('--dfile', '-df', type=str, help='Output Data Json file name ', required=False)

          args = parser.parse_args()
          start_time = make_naive(datetime.fromisoformat(args.start_time))
          rts = args.rts
          if args.duration is None:
               duration = 60*24
          if args.parameter is None:
               parameter = None
               vx = None
               vm = None
               iterations = 1
          if args.ifile:
               i_file = args.ifile
               log.info(f"Using file inputs {i_file}.")
          else:
               i_file = 'data/inputs/inputs.json'
          if args.efile:
               e_file = args.efile
               log.info(f"Using file inputs {e_file}.")
          else:
               e_file = 'data/events/events.json'
          if args.pfile:
               p_file = args.pfile
               log.info(f"Using file inputs {p_file}.")
          else:
               p_file = 'data/params/params.json'
```

```
53          if args.dfile:
54              d_file = args.dfile
55              log.info(f"Write output data in file {d_file}.")
56          else:
57              d_file = 'data/data/data.json'
58
59      else:
60          start_time = make_naive(start_time)
61          rts        = rts
62          if duration is None:
63              duration = 60*24
64          parameter=parameter
65          vx=vx
66          vm=vm
67          if iterations is None:
68              iterations = 1
69          if input_file is not None and input_file != "":
70              i_file = input_file
71              log.info(f"Using file inputs {i_file}.")
72          else:
73              i_file = 'data/inputs/inputs.json'
74          if events_file is not None and events_file != "":
75              e_file = events_file
76              log.info(f"Using file inputs {e_file}.")
77          else:
78              e_file = 'data/events/events.json'
79          if params_file is not None and params_file != "":
80              p_file = params_file
81              log.info(f"Using file inputs {p_file}.")
82          else:
83              p_file = 'data/params/params.json'
84          if data_file is not None and data_file != "":
85              d_file = data_file
86              log.info(f"Write output data in file {d_file}.")
87          else:
88              d_file = 'data/data/data.json'
89
90
91      # Load data from the JSON file
92      meals, bolus, basal = load_data_from_json(i_file)
93
94
95      # Create the Cambridge model
96      model = Cambridge(params_file=p_file, events_file=e_file)
97      # Solver
98      solver = SolverRK1_2()
99
100     # Create the simulator and configure meals
101     simulator = Simulator(model, solver=solver, duration=duration,
    time_step=1,
102                           i_delay = model.i_delay, m_delay = model.
    m_delay,
103                           input_file = i_file, events_file=e_file,
    data_file=d_file)
104
105     fig1, ax1_g, ax2_g, ax3_g = plot_init()
106     fig2, ax2_state           = plot_state_init()
107     fig3, ax3_param           = plot_parameters_init()
```

```
108
109     if rts == True:
110         # Execute simulation in real time
111         results = simulator.run_simulation_in_thread(start_time, rts=
    True, plot=plot, canvas = canvas, speed = speed)
112     else:
113
114         end_time = start_time + timedelta(minutes=duration)
115
116         # Execute one only simulation. Fixed parameters
117         if iterations == 1:
118
119             results = simulator.run_simulation(start_time, rts=False)
120             results, meals, basal, bolus = filter_all(results, meals,
    basal, bolus, start_time, end_time)
121
122             fig1, ax1 = plot_glucose(results, meals, basal, bolus,
    plot, fig1, ax1_g)
123             fig2, ax2 = plot_state_variables(results, plot, fig2,
    ax2_state)
124             fig3, ax3 = plot_parameters(results, plot, fig3, ax3_param
    )
125
126         # Execute multiple simulations. Varying parameters
127         else:
128
129             for i in range(iterations):
130                 results, _ = simulate_param_i(vm,vx,i,iterations,
    parameter,model,solver,duration,
131                                               model.i_delay, model.
    m_delay, 100, i_file,e_file,start_time)
132                 results, meals, basal, bolus = filter_all(results,
    meals, basal, bolus, start_time, end_time)
133                 fig1, ax1 = plot_glucose(results, meals, basal, bolus,
     plot, fig1, ax1_g)
134                 fig2, ax2 = plot_state_variables(results, plot, fig2,
    ax2_state)
135                 fig3, ax3 = plot_parameters(results, plot, fig3,
    ax3_param)
136
137         times = [datetime.fromisoformat(result['time']) for result in
    results]
138         carb_intake = [carb_intake_function(t, meals) for t in times]
139         ax2_g.plot(times, carb_intake, label='Carbohydrate intake (g)'
    , color='green', linestyle='--')
140         insulin_bolus_bars(ax1_g, bolus)
141         insulin_intake = [insulin_intake_function(t, basal) for t in
    times]
142
143         ax3_g.plot(times, insulin_intake, label='Insulin infusion rate
     (IIR)', color='red', linestyle='-.')
144
145     return fig1, fig2, fig3
146
147 if __name__ == "__main__":
148     main()
```

Código D.22: simulator_main.py (DiabetesTool).

D.9. simulator

D.9.1. auxi.py

```python
from datetime import datetime
from inputs.json_data import load_params

def calculate_ugp_duration(ugp):
    """
    Calculates the total digestion duration based on the number of UGPs.
    The duration is 3 hours for 1 UGP and increases by 1 additional hour for each extra UGP,
    up to a maximum of 8 hours.
    """
    base_duration = 3 * 60  # 3 hours in minutes for 1 UGP
    extra_duration = max(0, ugp - 1) * 60  # Add 1 extra hour per additional UGP
    return min(base_duration + extra_duration, 8 * 60)  # Max 8 hours

def meal_gp_intake(meal, t):
    """
    Models the absorption of fats and proteins (UGPs) over the digestion period.
    A linear function is used to distribute UGP absorption across the digestion time.
    """
    ugp_duration = calculate_ugp_duration(meal['UGPs'])  # minutes
    t_elapsed = (t - meal['start']).total_seconds() / 60  # Time in minutes since meal start

    if 0 <= t_elapsed < ugp_duration:
        return meal['UGPs'] * (t_elapsed / ugp_duration) / meal['duration']  # Linear absorption
    else:
        return 0

def insulin_efficacy(t, event_file):
    """ Calculates the insulin efficacy factor based on the time elapsed since the last reservoir change."""
    # Load reservoir change events
    events = load_params(event_file)
    reservoir_events = events.get("reservoir", [])

    # Apply reduction in insulin efficacy based on time elapsed since last reservoir change
    efficacy_factor = 1.0  # Initially no reduction
    for event in reservoir_events:
        # Get reservoir change start time and repetition interval
        start_time = datetime.fromisoformat(event["start"])
        repeat_every = event.get("repeat_every", 0)

        # Determine time since last reservoir change before current time `t`
        hours_since_change = ((t - start_time).total_seconds() / 3600) % (repeat_every * 24) if repeat_every else (t - start_time).
```

```
    total_seconds() / 3600
42          days_since_change = hours_since_change / 24
43
44          # Apply efficacy reduction if more than 2.5 days have passed
    since last change
45          if 2.5 <= days_since_change < 4:
46              # Compute efficacy reduction factor, from 1 to 0 at 4 days
47              efficacy_factor = max(0, 1 - (days_since_change - 2.5) /
    2.0)
48          elif days_since_change >= 4:
49              efficacy_factor = 0.25
50      return efficacy_factor
```

Código D.23: auxi.py (DiabetesTool).

D.9.2. simulator.py

```
1  from datetime import timedelta
2  import threading
3  import time
4  from carelink.json_carelink import write_json
5  from util.auxi          import MIN_H, S_MIN, momentary_activeInsulin
6  from util.glucose_plot import animate_glucose_and_inputs,
       animate_glucose_init
7  from util.glucose_plot import carb_intake_function,
       insulin_intake_function, insulin_bolus_bars
8  from simulator.auxi     import insulin_efficacy, meal_gp_intake
9
10 import logging as log
11 from inputs.json_data import load_data_from_json
12
13 RTS_RECORDS    = 300
14 MIN_RTS_STEP   = 5
15
16 class Simulator:
17     def __init__(self, model, meals=None, solver=None, duration=120,
       time_step=1,
18                  i_delay=30, m_delay=-5, initial_g = 100, input_file =
        'data/inputs/inputs.json',
19                  events_file='data/events/events.json', data_file='
       data/data/data.json'):
20          """
21          Initializes the simulator.
22
23          Args:
24          - model: The Cambridge model for simulating glucose and
       insulin.
25          - meals: List of meals (each with duration and carbohydrate
       amount).
26          - solver: Solver for the differential equations (e.g., RK4,
       not implemented here).
27          - duration: Duration of the simulation in minutes.
28          - time_step: Time step in minutes.
29          """
30          self.model = model
```

```
31          self.meals = meals if meals is not None else []
32          self.solver = solver
33          self.duration = duration  # in minutes
34          self.time_step = time_step  # in minutes
35          self.simulation_results = []
36          self.callback = None
37          self.iir = []
38          self.input_file = input_file
39          self.events_file = events_file
40          self.data_file = data_file
41          self.i_delay = i_delay # in minutes
42          self.m_delay = m_delay # in minutes
43          self.initial_g = initial_g
44          self.insulin_duration = 2.0 # hours
45
46      def set_input_file(self, input_file):
47          """ Set the input file for the simulation."""
48          self.input_file = input_file
49
50      def set_events_file(self, events_file):
51          """ Set the events file for the simulation."""
52          self.events_file = events_file
53
54      def print_results(self):
55          """ Write the results of the simulation in the data file."""
56          write_json(self.simulation_results,self.data_file)
57
58      def set_meals(self, meals):
59          """ Set the meals for the simulation. """
60          min_meal_duration = 1  # Minimum duration of a meal
61          self.meals = [
62              {**meal, 'duration': max(meal.get('duration',
    min_meal_duration), min_meal_duration)}
63              for meal in meals
64          ]
65
66      def get_iir(self):
67          """ Obtain the current insulin infusion rates (IIR). """
68          return self.iir
69
70      def set_iir(self, ibolus, ibasal):
71          """
72          Sets the insulin infusion based on basal insulin and boluses.
73
74          Args:
75          - ibolus: List of insulin boluses. Each bolus must have 'start
    ' and 'i' (insulin units).
76          - ibasal: List of basal infusions. Each entry must have 'start
    ', 'duration', and 'iir' (U/h).
77          """
78          self.iir = []
79
80          # Add the basal insulin with the start, duration and iir
81          for insulin in ibasal:
82              self.iir.append({
83                  'start': insulin['start'],
84                  'duration': insulin['duration'],
85                  'iir': insulin['iir']  # U/h
```

```
86              })
87
88          # Add insulin boluses, converting insulin units (i) to an
    infusion rate (iir)
89          for bolus in ibolus:
90              #  Boluses are assumed to be delivered over 1 minute
91              bolus_duration = 1  # min
92              bolus_iir = bolus['i'] / (bolus_duration / MIN_H)
93              self.iir.append({
94                  'start': bolus['start'],
95                  'duration': bolus_duration,
96                  'iir': bolus_iir  # U/h
97              })
98
99
100     def set_solver(self, solver):
101         """ Set the numerical solver. """
102         self.solver = solver
103
104
105     def _momentary_carb_intake(self, t):
106         """ Calculate the momentary carbohydrate intake at time t."""
107         carb_intake = sum(
108             meal['carbs'] / meal['duration'] if meal['start'] <= (t-
    timedelta(minutes=self.m_delay)) < meal['start'] + timedelta(
    minutes=meal['duration']) else 0
109             for meal in self.meals
110         )
111         ugp_intake = sum(
112             meal_gp_intake(meal, t) if meal['start'] <= (t - timedelta
    (minutes=self.m_delay)) else 0
113             for meal in self.meals
114         )
115         return carb_intake + ugp_intake
116
117     def _momentary_activeInsulin(self, t, ibolus):
118         """ Calculate the momentary active insulin at time t. """
119         return momentary_activeInsulin(self.insulin_duration, t,
    ibolus)
120
121
122     def _momentary_iir_intake(self, t, event_file):
123         """ Calculate the momentary insulin infusion rate (IIR) at
    time t."""
124
125         # Apply insulin efficacy reduction based on time since last
    reservoir change
126         efficiency_factor = insulin_efficacy(t, event_file)
127
128         # Compute insulin amount adjusted by efficacy factor
129         miiri = sum(
130             insulin['iir'] * efficiency_factor if insulin['start'] <=
    (t - timedelta(minutes=self.i_delay)) < insulin['start'] +
    timedelta(minutes=insulin['duration']) else 0
131             for insulin in self.iir
132         )
133         return miiri
134
```

```
135
136     def run_simulation_in_thread(self, start_time, rts = False, plot=
    False, canvas=None, speed = 1):
137         """Start the simulation process in a separate thread."""
138         threading.Thread(
139             target=self.run_simulation,  # Target method to run
140             args=(start_time, rts, plot, canvas, speed),  # Arguments
    for the method
141             daemon=True  # Ensure the thread stops when the main
    program exits
142         ).start()
143
144     def run_simulation(self, start_time, rts = False, plot = False,
    canvas=None, speed = 1, print_output = True):
145         """ Run the simulation."""
146         if rts:
147
148             fig, ax1, glucose_line, insulin_line, carb_line, times,
    glucose, _, insulin, _, chs = animate_glucose_init()
149
150             if canvas:
151                 canvas.figure = fig
152                 canvas.draw()
153                 canvas.flush_events()
154
155         t      = start_time
156         tmax   = start_time + timedelta(minutes=self.duration)
157         activeInsulin = 0
158         meals, ibolus, ibasal = load_data_from_json(self.input_file)
159         self.set_meals(meals)
160         self.set_iir(ibolus, ibasal)
161         carbs = self._momentary_carb_intake(t)
162         iir    = self.model.compute_IIR(self.initial_g, t)
163
164         input  = {"carbs": carbs, "iir": iir}
165         state  = self.model.compute_steady_state(input, t)
166         output = self.model.compute_output(state, t, activeInsulin)
167         tLast  = t - timedelta(minutes=self.time_step)
168
169         count = -1
170         div = MIN_RTS_STEP / self.time_step
171
172         time_wait = 0
173         while t <= tmax:
174             step_start_time = time.time()
175
176             # Real Time
177             if rts:
178                 meals, ibolus, ibasal = load_data_from_json(self.
    input_file)
179                 self.set_meals(meals)
180                 self.set_iir(ibolus, ibasal)
181                 count += 1
182
183             carbs  = self._momentary_carb_intake(t)
184             iir    = self._momentary_iir_intake(t, self.events_file)
185             input  = {"carbs": carbs, "iir": iir}
```

```
186             derivatives = lambda t, state: self.model.
    compute_derivatives(state, input, t)
187             state       = self.solver.solve(derivatives, tLast, state,
     t)
188             tLast = t
189 
190             activeInsulin = self._momentary_activeInsulin(t, ibolus)
191             output = self.model.compute_output(state, t, activeInsulin
    )
192 
193             params = self.model.evaluate_parameter_values_at(t, self.
    events_file)
194 
195             # Store the results
196             self.store_results(output, t, state, params)
197             if print_output:
198                 self.print_results()
199 
200             if rts:
201                 log.info(f"Time: {t}, Glucose: {round(output['Gp'],2)}
     mg/dl, Active insulin: {output['activeInsulin']} U")
202                 log.info(f"Carbohydrates: {carbs} g, IIR: {iir} U/h")
203                 log.info("")
204 
205             if rts:
206                 times.append(t)
207                 glucose.append(output['Gp'])
208                 insulin.append(insulin_intake_function(t,ibasal))
209                 chs.append(carb_intake_function(t,meals))
210                 insulin_bolus_bars(ax1, ibolus)
211                 times   = times[  -RTS_RECORDS:]
212                 glucose = glucose[-RTS_RECORDS:]
213                 insulin = insulin[-RTS_RECORDS:]
214                 chs     = chs[    -RTS_RECORDS:]
215 
216                 # Encourage glucose, insulin and carbohydrates in real
     time
217                 animate_glucose_and_inputs(fig, ax1, glucose_line,
    insulin_line, carb_line, times, glucose, times, insulin, times,
    chs, canvas)
218                 if canvas:
219                     canvas.draw()
220                     canvas.flush_events()
221 
222             if rts:
223                 elapsed_real_time = time.time() - step_start_time
224                 time_wait += self.time_step*S_MIN/speed-
    elapsed_real_time
225                 time.sleep(max(0,time_wait))
226                 if time_wait < 0:
227                     log.warning(f'Simulation may run slower than
    expected {time_wait}')
228                 else:
229                     time_wait = 0
230 
231             # Advance time
232             t += timedelta(minutes=self.time_step)
233 
```

```
234         return self.simulation_results
235
236     def get_simulation_results(self):
237         """ Return the simulation results."""
238         return self.simulation_results
239
240     def store_results(self, output, t, state, params):
241         """ Store the results of the simulation in a list. """
242         result = {
243             "time": t.isoformat(),
244             "glucose": int(output['Gp']),
245             "state": state.copy(),
246             "params": params.copy(),
247             "activeInsulin": output['activeInsulin'],
248         }
249         self.simulation_results.append(result)
250
251
252 def simulate_param_i(vm,vx,i,iterations,parameter,model,solver,
    duration,
253                      i_delay, m_delay, initial_g,i_file,e_file,
    start_time):
254     """ Simulate the model with a specific parameter value."""
255
256     if iterations <= 0:
257         raise ValueError("The number of iterations must be greater
    than 0.")
258     elif iterations == 1:
259         param_value = (vm + vx) / 2
260     else:
261         param_value = vm + (vx - vm) * (i / (iterations - 1))
262
263     if parameter in ['i_delay', 'm_delay']:
264         # Create the simulator with the specific delay parameter set
265         if parameter == 'i_delay':
266             simulator = Simulator(model, solver=solver, duration=
    duration, time_step=1,
267                                   i_delay=param_value, m_delay =
    m_delay, initial_g=initial_g,
268                                   input_file = i_file, events_file=
    e_file)
269         else:
270             simulator = Simulator(model, solver=solver, duration=
    duration, time_step=1,
271                                   i_delay = i_delay, m_delay=
    param_value, initial_g=initial_g,
272                                   input_file = i_file, events_file=
    e_file)
273     else:
274         # Set the new value of the parameter in the model
275         setattr(model, parameter, param_value)
276         # Create the simulator without modifying delays
277         simulator = Simulator(model, solver=solver, duration=duration,
     time_step=1,
278                               i_delay=i_delay, m_delay = m_delay,
    initial_g=initial_g,
279                               input_file = i_file, events_file=e_file)
280
```

```
281     results = simulator.run_simulation(start_time, rts=False)
282     return results, param_value
```

Código D.24: simulator.py (DiabetesTool).

D.9.3. solver.py

```
1  import math
2  from datetime import timedelta
3  from util.auxi import norm, times_scalar, vector_sum
4
5
6  class SolverRK1_2:
7      def __init__(self, default_time_step=1, tol=1e-3):
8          """
9          Initializes the solver with a default time step and relative
       tolerance.
10         """
11         self._time_step = default_time_step  # Current time step in
       minutes
12         self._tol = tol  # Relative tolerance
13
14     def reset(self, default_time_step):
15         """
16         Resets the time step to the default value.
17         """
18         self._time_step = default_time_step
19
20     def solve(self, derivatives, t_init, x_init, t_final):
21         """
22         Solves the differential equation using RK1 (Euler) and RK2 (
       Heun) methods.
23
24         Args:
25         - derivatives: Function that returns dx/dt.
26         - t_init: Initial time.
27         - x_init: Initial state.
28         - t_final: Final time.
29
30         Returns:
31         - x: Final state after integration.
32         """
33         t = t_init
34         x = x_init
35
36         while t < t_final:
37             while True:
38                 # Advance time by _time_step, but do not exceed
       t_final
39                 t_next = min(t + timedelta(minutes=self._time_step),
       t_final)
40                 # Actual step size in minutes
41                 dt = (t_next - t).total_seconds() / 60
42
43                 # Perform a single time step
```

```
                step = self.perform_time_step(derivatives, t, x, dt)

                if step['e'] > self._tol:
                    # Error exceeds tolerance, reduce step size and retry
                    self._time_step /= 2
                else:
                    # Step is acceptable
                    x = step['x']
                    t = t_next

                # Check if the error is zero or very small to avoid division by zero
                if step['e'] == 0:
                    step['e'] = 1e-10  # Assign a very small value

                    # Adjust next time step based on error estimate
                    self._time_step = max(1, min(60,
                        math.floor(self._time_step *
                        min(2, 0.9 * math.sqrt(self._tol / 2 / step['e'])))))

                    # Finish this step
                    break

        return x

    def perform_time_step(self, derivatives, t, x, dt):
        """
        Performs a single time step using RK2 (Heun) and RK1 (Explicit Euler).

        Args:
        - derivatives: Function that returns dx/dt.
        - t: Initial time.
        - x: Initial state.
        - dt: Time step in minutes.

        Returns:
        - A dictionary with the new state 'x' and the relative error 'e'.
        """
        k1 = times_scalar(derivatives(t, x), dt)
        k2 = times_scalar(derivatives(t + timedelta(minutes=dt), vector_sum(x, k1)), dt)

        # Compute error between Heun (RK2) and Explicit Euler (RK1)
        e = vector_sum(times_scalar(k1, -0.5), times_scalar(k2, 0.5))

        return {
            'x': vector_sum(x, times_scalar(k1, 0.5), times_scalar(k2, 0.5)),
            'e': norm(e) / norm(x)
        }
```

Código D.25: solver.py (DiabetesTool).

D.10. util

D.10.1. auxi.py

```
import argparse
import base64
import hashlib
import random
import string
import uuid
import secrets
import os
from datetime import datetime, timedelta
import math
import copy

# Logging config
FORMAT  = '[%(asctime)s:%(levelname)s] %(message)s'
MIN_H   = 60
S_MIN   = 60
H_DAY   = 24
MIN_DAY = MIN_H*H_DAY
MS_PER_MIN = 60 * 1000

def get_closest_sg(data, start_time):
    """ Obtains the closest SG value to a given start time from the data. """
    sgs = copy.deepcopy(data)

    # Convert 'sgs' dates to naive datetime objects
    for sg_entry in sgs:
        sg_entry['datetime'] = make_naive(datetime.fromisoformat(sg_entry['datetime']))

    # Find the entry closest to 'start_time'
    closest_sg = min(sgs, key=lambda sg_entry: abs(sg_entry['datetime'] - start_time))

    return closest_sg['sg']

def str2bool(value):
    """Convert a string to a boolean if the input is explicitly 'True' or 'False'."""
    if value.lower() not in {'true', 'false'}:
        raise argparse.ArgumentTypeError("Boolean value expected. Please enter 'True' or 'False'.")
    return value.lower() == 'true'

def get_current_time():
    """ Returns the current time in ISO 8601 format. """
    return datetime.now().strftime("%Y-%m-%dT%H:%M:%S")

def make_naive(dt):
    """Converts a timezone-aware datetime to naive by removing timezone info."""
    if dt.tzinfo is not None:
```

```
47          return dt.replace(tzinfo=None)
48      return dt
49
50  def make_naive_param(param,str):
51      """Converts a timezone-aware datetime to naive by removing
        timezone info."""
52      for p in param:
53          p[str] = make_naive(p[str])
54      return param
55
56  def make_naive_all(rs, ms, bas, bos):
57      """Converts a timezone-aware datetime to naive by removing
        timezone info."""
58      rs  = make_naive_param(rs,'time' )
59      ms  = make_naive_param(ms,'start')
60      bas = make_naive_param(bas,'start')
61      bos = make_naive_param(bos,'start')
62      return rs, ms, bas, bos
63
64  def random_b64_str(length):
65      """ Generate a random base64 string of a given length. """
66      random_chars = ''.join(random.choice(string.ascii_letters + string
        .digits) for _ in range(length + 10))
67      base64_string = base64.b64encode(random_chars.encode('utf-8')).
        decode('utf-8')
68      return base64_string[:length]
69
70  def random_uuid():
71      """ Generate a random UUID. """
72      return str(uuid.UUID(bytes=secrets.token_bytes(16)))
73
74  def random_android_model():
75      """ Generate a random Android model. """
76      models = ['SM-G973F', "SM-G988U1", "SM-G981W", "SM-G9600"]
77      random.shuffle(models)
78      return models[0]
79
80  def random_android_model():
81      """ Generate a random Android model. """
82      return hashlib.sha256(os.urandom(40)).hexdigest()
83
84  def random_device_id():
85      """ Generate a random device ID. """
86      return hashlib.sha256(os.urandom(40)).hexdigest()
87
88  def filter_str(start_time, end_time, params, str):
89      """ Filter parameters based on a string date range. """
90      return  [p for p in params if start_time <= datetime.fromisoformat
        (p[str]) <= end_time]
91
92  def filter_datetime(start_time, end_time, params, str):
93      """ Filter parameters based on a datetime range. """
94      filtered_params = []
95      for p in params:
96          if 'duration' in p:
97              end_param_time = p[str] + timedelta(minutes=p['duration'])
98              if start_time <= end_param_time and p[str] <= end_time:
99                  filtered_params.append(p)
```

```
100         else:
101             if start_time <= p[str] <= end_time:
102                 filtered_params.append(p)
103     return filtered_params
104 
105 def filter_all(rs, ms, bas, bos, s, e, rs_datetime = False):
106     """ Filter all parameters based on a date range. """
107     if rs_datetime:
108         rs  = filter_datetime(s,e,rs,'time' )
109     else:
110         rs  = filter_str( s,e,rs, 'time' )
111     ms  = filter_datetime(s,e,ms, 'start')
112     bas = filter_datetime(s,e,bas,'start')
113     bos = filter_datetime(s,e,bos,'start')
114     return rs, ms, bas, bos
115 
116 
117 def norm(x):
118     """ Computes the Euclidean norm of a vector. """
119     return math.sqrt(sum(value ** 2 for value in x.values()))
120 
121 def times_scalar(vec, scalar):
122     """Multiply each vector value by a scalar."""
123     return {key: value * scalar for key, value in vec.items()}
124 
125 def vector_sum(*vectors):
126     """ Add multiple vectors. """
127     result = {}
128     for vec in vectors:
129         for key, value in vec.items():
130             result[key] = result.get(key, 0) + value
131     return result
132 
133 
134 def momentary_activeInsulin(insulin_duration, t, ibolus):
135     """ Computes the momentary active insulin at time t. """
136     active_insulin = 0
137 
138     for bolus in ibolus:
139         start   = bolus['start']
140         insulin = bolus['i']
141         end     = start + timedelta(hours=insulin_duration)
142         if start <= t < end:
143             # Calculate the time elapsed from the beginning
144             elapsed_time = (t - start).total_seconds() / (S_MIN*MIN_H)
    # In hours
145             # Calculate the remaining active insulin proportion
146             remaining_factor = max(0, 1 - (elapsed_time /
   insulin_duration))
147             # Add active insulin
148             active_insulin += insulin * remaining_factor
149 
150     return active_insulin
```

Código D.26: auxi.py (DiabetesTool).

D.10.2. glucose_plot.py

```
import matplotlib.pyplot as plt # type: ignore
from datetime import datetime, timedelta
from carelink.json_carelink import load_json
from carelink.data import parse_markers, parse_results_sgs
from util.auxi import make_naive_all, filter_all, make_naive
import matplotlib.dates as mdates
MAX_I = 10

def carb_intake_function(t, meals):
    """ Function that returns the amount of carbohydrate intake at time 't' """
    return sum(
        meal['carbs'] if make_naive(meal['start']) <= t < make_naive(meal['start']) + timedelta(minutes=meal['duration']) else 0
        for meal in meals
    )

def insulin_intake_function(t, insulin_injections):
    """ Function that returns the insulin infusion rate at time 't' (IIR). """
    return sum(
        injection['iir'] if make_naive(injection['start']) <= t < make_naive(injection['start']) + timedelta(minutes=injection['duration']) else 0
        for injection in insulin_injections
    )

def insulin_bolus_bars(ax, ibolus, color='purple'):
    """
    Draws vertical bars on the plot to represent insulin boluses.

    Args:
    - ax: The axis of the plot where the bars will be drawn.
    - ibolus: List of insulin boluses. Each bolus must have 'start' and 'i' (units of insulin).
    """
    for bolus in ibolus:
        ax.axvline(x=bolus['start'], ymin=0, ymax=bolus['i']/MAX_I, color=color, linestyle='-', lw=5.0)
        # Add a label on top of the bar
        ax.text(bolus['start'], bolus['i'], f'{bolus["i"]}U', color=color, rotation=0, verticalalignment='bottom')



def plot_parameters(results, plot, fig0 = None, axs = None):
    """ Graph all physiological parameters returned by the `evaluate_parameter_atues_at (self, t_) function over time. """

    times = [datetime.fromisoformat(result['time']) for result in results]

    BW = [result['params']['BW'] for result in results]
    ka1 = [result['params']['ka1'] for result in results]
    ka2 = [result['params']['ka2'] for result in results]
    ka3 = [result['params']['ka3'] for result in results]
```

```
47      VG = [result['params']['VG'] for result in results]
48      F01 = [result['params']['F01'] for result in results]
49      EGP0 = [result['params']['EGP0'] for result in results]
50      tmaxI = [result['params']['tmaxI'] for result in results]
51      VI = [result['params']['VI'] for result in results]
52      ke = [result['params']['ke'] for result in results]
53      tmaxG = [result['params']['tmaxG'] for result in results]
54      SIT = [result['params']['SIT'] for result in results]
55      SID = [result['params']['SID'] for result in results]
56      SIE = [result['params']['SIE'] for result in results]
57      k12 = [result['params']['k12'] for result in results]
58      Gpeq = [result['params']['Gpeq'] for result in results]
59      AG = [result['params']['AG'] for result in results]
60
61      if fig0 is None:
62          fig, axs = plot_parameters_init()
63
64      axs[0, 0].plot(times, BW, label='BW (kg)')
65      axs[0, 1].plot(times, ka1, label='ka1 (1/min)')
66      axs[0, 2].plot(times, ka2, label='ka2 (1/min)')
67      axs[0, 3].plot(times, ka3, label='ka3 (1/min)')
68      axs[1, 0].plot(times, VG, label='VG (l/kg)')
69      axs[1, 1].plot(times, F01, label='F01 (mmol/min)')
70      axs[1, 2].plot(times, EGP0, label='EGP0 (mmol/kg/min)')
71      axs[1, 3].plot(times, tmaxI, label='tmaxI (min)')
72      axs[2, 0].plot(times, VI, label='VI (l/kg)')
73      axs[2, 1].plot(times, ke, label='ke (1/min)')
74      axs[2, 2].plot(times, tmaxG, label='tmaxG (min)')
75      axs[2, 3].plot(times, SIT, label='SIT (1/min/mU/l)')
76      axs[3, 0].plot(times, SID, label='SID (1/min/mU/l)')
77      axs[3, 1].plot(times, SIE, label='SIE (1/mU/l)')
78      axs[3, 2].plot(times, k12, label='k12 (1/min)')
79      axs[3, 3].plot(times, Gpeq, label='Gpeq (mg/dl)')
80      axs[4, 0].plot(times, AG, label='AG')
81
82      if plot:
83          plt.show()
84      fig = fig0
85      return fig, axs
86
87
88  def plot_parameters_init():
89      """ Initialize the figure and axes for plotting parameters. """
90      fig, axs = plt.subplots(5, 4, figsize=(18, 12))
91      fig.suptitle('Evolution of physiological parameters', fontsize=16)
92
93      font_size = 10
94      axs[0, 0].set_title('Body weight (BW)', fontsize=font_size)
95      axs[0, 1].set_title('ka1 (Deactivation rate)', fontsize=font_size)
96      axs[0, 2].set_title('ka2 (Deactivation rate)', fontsize=font_size)
97      axs[0, 3].set_title('ka3 (Deactivation rate)', fontsize=font_size)
98      axs[1, 0].set_title('VG (Volume of accessible compartment)',
        fontsize=font_size)
99      axs[1, 1].set_title('F01 (Non-insulin-dependent glucose flux)',
        fontsize=font_size)
100     axs[1, 2].set_title('EGP0 (Endogenous glucose production)',
        fontsize=font_size)
```

```
101     axs[1, 3].set_title('tmaxI (Time to maximum insulin absorption)',
    fontsize=font_size)
102     axs[2, 0].set_title('VI (Insulin distribution volume)', fontsize=
    font_size)
103     axs[2, 1].set_title('ke (Insulin elimination rate)', fontsize=
    font_size)
104     axs[2, 2].set_title('tmaxG (Maximum glucose absorption time)',
    fontsize=font_size)
105     axs[2, 3].set_title('SIT (Insulin sensitivity for transport)',
    fontsize=font_size)
106     axs[3, 0].set_title('SID (Insulin sensitivity for disposal)',
    fontsize=font_size)
107     axs[3, 1].set_title('SIE (Insulin sensitivity for endogenous
    production)', fontsize=font_size)
108     axs[3, 2].set_title('k12 (Transfer rate between compartments)',
    fontsize=font_size)
109     axs[3, 3].set_title('Steady-state glucose (Gpeq)', fontsize=
    font_size)
110     axs[4, 0].set_title('AG (CHO bioavailability)', fontsize=font_size
    )
111 
112     axs[4, 1].axis('off')
113     axs[4, 2].axis('off')
114     axs[4, 3].axis('off')
115 
116     for ax in axs.flat:
117         ax.xaxis.set_major_locator(plt.MaxNLocator(4))  # Maximum 4
    ticks on the X axis
118         ax.xaxis.set_major_formatter(mdates.DateFormatter('%d-%H:%M'))
      # Day/hour format
119 
120     fig.tight_layout(rect=[0, 0, 1, 0.96])
121     return fig, axs
122 
123 
124 def plot_glucose(results, meals, basal, bolus, plot, fig0=None, ax1 =
    None):
125     """
126     Function to graph glucose levels, carbohydrates and insulin
    infusion rate (IIR),
127     with a clear green background for glucose levels between 70 and
    180 mg/dl.
128     """
129 
130     if fig0 is None:
131         fig, ax1, ax2, ax3 = plot_init()
132 
133     if(type(results[0].get('time'))==str):
134         times = [datetime.fromisoformat(result['time']) for result in
    results]
135     else:
136         times = [result['time'] for result in results]
137     glucose_levels = [result['glucose'] for result in results]
138 
139 
140     # Graph glucose levels on the first axis (left)
141     ax1.plot(times, glucose_levels, label='Glucose levels (mg/dl)',
    color='blue')
```

```
142
143     if fig0 is None:
144         carb_intake = [carb_intake_function(t, meals) for t in times]
145         ax2.plot(times, carb_intake, label='Carbohydrate intake (g)',
    color='green', linestyle='--')
146
147         # Draw insulin bolus bars
148         insulin_bolus_bars(ax1, bolus)
149
150         insulin_intake = [insulin_intake_function(t, basal) for t in
    times]
151         ax3.plot(times, insulin_intake, label='Insulin infusion rate (
    IIR)', color='red', linestyle='-.')
152
153         plt.title('Evolution of glucose, carbohydrates and insulin
    levels (IIR)')
154         plt.xticks(rotation=45)
155
156         ax1.legend(loc='upper left')
157         ax2.legend(loc='upper right')
158         ax3.legend(loc='lower right')
159
160         fig.tight_layout()
161
162     if plot:
163         plt.show()
164     fig = fig0
165     return fig, ax1
166
167
168 def plot_state_variables(results, plot, fig0=None, axs = None):
169     """
170     Function to graph groups of state variables of the Cambridge model
    , including units.
171     """
172     times = [datetime.fromisoformat(result['time']) for result in
    results]
173
174     # Extract the state variables to be plotted
175     Q1 = [result['state']['Q1'] for result in results]  # Glucose in
    the accessible compartment
176     Q2 = [result['state']['Q2'] for result in results]  # Glucose in
    the non-accessible compartment
177     S1 = [result['state']['S1'] for result in results]  # Insulin in
    compartment 1
178     S2 = [result['state']['S2'] for result in results]  # Insulin in
    compartment 2
179     I  = [result['state']['I']  for result in results]  # Plasma
    insulin
180     x1 = [result['state']['x1'] for result in results]  # Insulin
    action on transport
181     x2 = [result['state']['x2'] for result in results]  # Insulin
    action on disposal
182     x3 = [result['state']['x3'] for result in results]  # Insulin
    action on endogenous production
183     D1 = [result['state']['D1'] for result in results]  # Glucose in
    the intestine compartment 1
```

```
184     D2 = [result['state']['D2'] for result in results]  # Glucose in
    the intestine compartment 2
185 
186     if fig0 is None:
187         fig, axs = plot_state_init()
188 
189     # Graph 1: Q1 and Q2
190     axs[0].plot(times, Q1, label='Q1 (Accessible glucose)', color='red
    ')
191     axs[0].plot(times, Q2, label='Q2 (Non-accessible glucose)', color=
    'orange')
192     axs[0].legend()
193 
194     # Graph 2: S1, S2 and I
195     axs[1].plot(times, S1, label='S1 (Insulin compartment 1)', color='
    green')
196     axs[1].plot(times, S2, label='S2 (Insulin compartment 2)', color='
    blue')
197     axs[1].plot(times, I, label='I (Plasma insulin)', color='purple')
198     axs[1].legend()
199 
200     # Graph 3: x1, x2 and x3
201     axs[2].plot(times, x1, label='x1 (Insulin action on transport)',
    color='brown')
202     axs[2].plot(times, x2, label='x2 (Insulin action on disposal)',
    color='cyan')
203     axs[2].plot(times, x3, label='x3 (Insulin action on endogenous
    production)', color='magenta')
204     axs[2].legend()
205 
206     # Graph 4: D1 and D2
207     axs[3].plot(times, D1, label='D1 (Intestinal glucose 1)', color='
    yellow')
208     axs[3].plot(times, D2, label='D2 (Intestinal glucose 2)', color='
    pink')
209     axs[3].legend()
210 
211     plt.tight_layout()
212 
213     if plot:
214         plt.show()
215     fig = fig0
216     return fig, axs
217 
218 def plot_state_init():
219     """Initialize the figure and axes for plotting state variables."""
220 
221     fig, axs = plt.subplots(4, 1, figsize=(10, 12))  # 4 rows, 1
    column
222 
223     # Graph 1: Q1 and Q2
224     axs[0].set_title('Q1 and Q2 (Glucose in compartments) [mmol]')
225     axs[0].set_ylabel('Glucose [mmol]')
226 
227     # Graph 2: S1, S2 and I
228     axs[1].set_title('S1, S2 and I (Insulin) [mU]')
229     axs[1].set_ylabel('Insulin [mU]')
230 
```

```
231     # Graph 3: x1, x2 and x3
232     axs[2].set_title('x1, x2 and x3 (Insulin action) [1/min]')
233     axs[2].set_ylabel('Insulin action [1/min]')
234 
235     # Graph 4: D1 and D2
236     axs[3].set_title('D1 and D2 (Intestinal glucose) [mmol]')
237     axs[3].set_ylabel('Intestinal glucose [mmol]')
238 
239     return fig, axs
240 
241 def animate_glucose_and_inputs(fig, ax1, glucose_line, insulin_line,
        carb_line,
242                                times, glucose, ti, insulin, tc, carbs,
        canvas = None):
243     """
244     Function to animate glucose levels, injected insulin, and meals in
        real time.
245 
246     Args:
247     - fig: Matplotlib figure.
248     - ax1: Axis for glucose.
249     - ax2: Axis for insulin and carbohydrates.
250     - glucose_line: Glucose line.
251     - insulin_line: Insulin line.
252     - carb_line: Carbohydrate line.
253     - times: Simulation times.
254     - glucose: Glucose levels.
255     - insulin: Insulin levels.
256     - carbs: Carbohydrate intake.
257     """
258     # Update plot lines
259     glucose_line.set_data(times, glucose)
260     insulin_line.set_data(ti,    insulin)
261     carb_line.set_data(   tc,    carbs)
262 
263     # Adjust axis limits
264     ax1.set_xlim(times[0], times[-1])
265 
266     # Draw and update figure
267     if canvas is None:
268         fig.canvas.draw()
269         fig.canvas.flush_events()
270 
271 def animate_glucose_init():
272     """Initialize the figure and axes for glucose animation."""
273 
274     # Initialization for animation
275     fig, ax1, ax2, ax3 = plot_init()
276 
277     glucose_line, = ax1.plot([], [], 'b-', label='Glucose')
278     insulin_line, = ax3.plot([], [], 'r-', label='Insulin')
279     carb_line, = ax2.plot([], [], 'g-', label='Carbohydrates')
280 
281     # Initial data for the plots
282     times   = []
283     ti      = []
284     tc      = []
285     glucose = []
```

```
286     insulin = []
287     carbs   = []
288 
289     return fig, ax1, glucose_line, insulin_line, carb_line, times,
    glucose, ti, insulin, tc, carbs
290 
291 def plot_init():
292     """Initialize the figure and axes for glucose, insulin, and
    carbohydrate plots."""
293 
294     # Initialization for animation
295     fig, ax1 = plt.subplots(figsize=(10, 6))
296     ax2 = ax1.twinx()  # Axis for insulin and carbohydrates
297 
298     ax1.set_xlabel('Time')
299     ax1.set_ylabel('Glucose (mg/dl)', color='blue')
300     ax1.tick_params(axis='y', labelcolor='blue')
301     ax1.set_ylim(40, 400)
302     ax1.grid(True)
303     ax1.axhspan(70, 180, color='lightgreen', alpha=0.3)  # Light green
    background
304 
305     ax2.set_ylabel('Carbohydrates (g)', color='green')
306     ax2.tick_params(axis='y', labelcolor='green')
307     ax2.set_ylim(0, 80)  # Limit carbohydrate scale from 0 to 80 g
308 
309     ax3 = ax1.twinx()  # Create a third axis that also shares the x-
    axis
310     ax3.spines['right'].set_position(('outward', 50))  # Adjust the
    axis so it does not overlap
311     ax3.set_ylabel('Insulin (U/h)', color='red')
312     ax3.tick_params(axis='y', labelcolor='red')
313     ax3.set_ylim(0, MAX_I)  # Limit insulin scale from 0 to 10 U/h
314 
315     return fig, ax1, ax2, ax3
316 
317 
318 def plot_compare(results, meals, basal, bolus, filename_simple,
    start_time, end_time, plot):
319     """Compare glucose levels, carbohydrates and insulin infusion rate
    (IIR) with real data in a plot."""
320 
321     # Initialization for animation
322     fig, ax1, ax2, ax3 = plot_init()
323 
324     # Read simplified JSON file
325     data = load_json(filename_simple)
326 
327     # Parse meals, basal and bolus from simplified file
328     real_meals, real_basal, real_bolus = parse_markers(data)
329 
330     # Parse glucose results (SGs)
331     real_results = parse_results_sgs(data.get('sgs', []))
332 
333     # Convert datetime fields in meals, bolus and basal to naive
334     real_results, real_meals, real_basal, real_bolus = make_naive_all(
    real_results, real_meals, real_basal, real_bolus)
335 
```

```
336     # Filter according to selected time range
337     if start_time and end_time:
338         real_results, real_meals, real_basal, real_bolus = filter_all(
    real_results, real_meals, real_basal, real_bolus, start_time,
    end_time, True)
339 
340     # Extract times and glucose levels
341     times = [datetime.fromisoformat(result['time']) for result in
    results]
342     real_times = [result['time'] for result in real_results]
343     glucose_levels = [result['glucose'] for result in results]
344     real_glucose_levels = [result['glucose'] for result in
    real_results]
345 
346     # Plot glucose levels
347     ax1.plot(times, glucose_levels, label='Glucose levels (mg/dl)',
    color='blue')
348     ax1.plot(real_times, real_glucose_levels, color='black')
349 
350     carb_intake = [carb_intake_function(t, meals) for t in times]
351     ax2.plot(times, carb_intake, label='Carbohydrate intake (g)',
    color='green', linestyle='--')
352 
353     # Draw insulin bolus bars
354     insulin_bolus_bars(ax1, bolus)
355 
356     insulin_intake = [insulin_intake_function(t, basal) for t in times
    ]
357     ax3.plot(times, insulin_intake, label='Insulin infusion rate (IIR)
    ', color='red', linestyle='-.')
358 
359     # Add title and adjust layout
360     plt.title('Evolution of Glucose Levels, Carbohydrates and Insulin
    (IIR)')
361 
362     # Rotate x-axis labels if they are dates
363     plt.xticks(rotation=45)
364 
365     # Add combined legend
366     ax1.legend(loc='upper left')
367     ax2.legend(loc='upper right')
368     ax3.legend(loc='lower right')
369 
370     # Adjust layout to avoid overlap
371     fig.tight_layout()
372 
373     # Show figure
374     if plot:
375         plt.show()
376     return fig
377 
378 def plot_data(filename_simple, start_time=None, end_time=None, plot=
    False):
379     """Load simplified JSON data and plot glucose, carbohydrate and
    insulin curves using glucose_plot functions."""
380 
381     fig, ax1_g, ax2_g, ax3_g = plot_init()
382 
```

```
383     # Load simplified JSON file
384     data = load_json(filename_simple)
385 
386     # Parse meals, basal and bolus
387     meals, basal, bolus = parse_markers(data)
388 
389     # Parse glucose results (SGs)
390     results = parse_results_sgs(data.get('sgs', []))
391 
392     # Convert datetimes to naive
393     results, meals, basal, bolus = make_naive_all(results, meals,
    basal, bolus)
394 
395     # Filter by selected time range
396     if start_time and end_time:
397         results, meals, basal, bolus = filter_all(results, meals,
    basal, bolus, start_time, end_time)
398 
399     # Plot results
400     fig, _ = plot_glucose(results, meals, basal, bolus, plot, fig,
    ax1_g)
401 
402     times = [result['time'] for result in results]
403     carb_intake = [carb_intake_function(t, meals) for t in times]
404     ax2_g.plot(times, carb_intake, label='Carbohydrate intake (g)',
    color='green', linestyle='--')
405     insulin_bolus_bars(ax1_g, bolus)
406     insulin_intake = [insulin_intake_function(t, basal) for t in times
    ]
407     ax3_g.plot(times, insulin_intake, label='Insulin infusion rate (
    IIR)', color='red', linestyle='-.')
408 
409     return fig
410 
411 def plot_compare_ins(filename_simple, basal_comp, bolus_comp,
    start_time=None, end_time=None, plot=False):
412     """Load simplified JSON data and compare two insulin plans in a
    glucose plot."""
413 
414     fig, ax1_g, ax2_g, ax3_g = plot_init()
415 
416     # Load simplified JSON file
417     data = load_json(filename_simple)
418 
419     # Parse meals, basal and bolus
420     meals, basal, bolus = parse_markers(data)
421 
422     # Parse glucose results (SGs)
423     results = parse_results_sgs(data.get('sgs', []))
424 
425     # Convert datetimes to naive
426     results, meals, basal, bolus = make_naive_all(results, meals,
    basal, bolus)
427 
428     # Filter by selected time range
429     if start_time and end_time:
430         results, meals, basal, bolus = filter_all(results, meals,
    basal, bolus, start_time, end_time, rs_datetime=True)
```

```
431
432     # Plot reference data
433     fig, _ = plot_glucose(results, meals, basal, bolus, plot, fig,
        ax1_g)
434
435     times = [result['time'] for result in results]
436     carb_intake = [carb_intake_function(t, meals) for t in times]
437     ax2_g.plot(times, carb_intake, label='Carbohydrate intake (g)',
        color='green', linestyle='--')
438     insulin_bolus_bars(ax1_g, bolus)
439     insulin_bolus_bars(ax1_g, bolus_comp, color='magenta')
440
441     insulin_intake_ref  = [insulin_intake_function(t, basal) for t in
        times]
442     insulin_intake_comp = [insulin_intake_function(t, basal_comp) for
        t in times]
443
444     ax3_g.plot(times, insulin_intake_ref,  label='Insulin infusion
        rate (IIR)', color='red',     linestyle='-.')
445     ax3_g.plot(times, insulin_intake_comp,
        color='magenta', linestyle='-.')
446
447     return fig
```

Código D.27: glucose_plot.py (DiabetesTool).

D.10.3. residuals.py

```
1  import matplotlib.pyplot as plt  # type: ignore
2  import logging as log
3  from carelink.data import parse_markers, parse_results_sgs
4  from carelink.json_carelink import load_json
5  import numpy as np  # type: ignore
6  from datetime import datetime, timedelta
7
8  from simulator.simulator import simulate_param_i
9  from util.auxi import filter_all, make_naive, get_closest_sg,
       make_naive_all
10
11
12 def plot_residuals(residuals, times, plot=False):
13     """
14     Function to plot residuals between real and simulated glucose
       values.
15
16     Args:
17     - residuals: List of residuals (difference between simulated and
       real values).
18     - times: List of timestamps corresponding to real data.
19     """
20     fig = plt.figure(figsize=(12, 6))
21
22     # Plot residuals
23     plt.plot(times, residuals, marker='o', linestyle='-', color='red',
       label='Residuals')
```

```
24
25      # Add chart details
26      plt.title('Residuals between simulated and real data')
27      plt.xlabel('Time')
28      plt.ylabel('Residuals (mg/dL)')
29      plt.axhline(y=0, color='black', linestyle='--', linewidth=1)
30      plt.legend(loc='upper right')
31
32      # Rotate x-axis labels if they are dates
33      plt.xticks(rotation=45)
34
35      # Adjust layout to avoid overlap
36      plt.tight_layout()
37
38      # Show plot
39      if plot:
40          plt.show()
41      return fig
42
43  def calculate_residuals(simulated, real):
44      """Calculates residuals as the difference between simulated and
    real data."""
45      residuals = []
46      times = []
47      for real_point in real:
48          real_time = make_naive(datetime.fromisoformat(real_point['
    datetime']))
49
50          # Find the simulated glucose value closest to the real data
    time
51          closest_sim = min(simulated, key=lambda x: abs(make_naive(
    datetime.fromisoformat(x['time'])) - real_time))
52          residuals.append(closest_sim['glucose'] - real_point['sg'])
53          times.append(real_time)
54
55      return residuals, times
56
57  def optimize_parameter(start_time, model, solver, parameter, duration,
      vx, vm, max_iter, real_data,
58                         i_delay, m_delay, i_file, e_file):
59      """Optimizes a model parameter to minimize the RMS of the
    residuals."""
60      best_param_value = None
61      best_rms = float('inf')
62      best_residuals = []
63      end_time = start_time + timedelta(minutes=duration)
64
65      real_data_copy = real_data.copy()
66      initial_g = get_closest_sg(real_data_copy, start_time)
67      for i in range(max_iter):
68          results, param_value = simulate_param_i(vm, vx, i, max_iter,
    parameter, model, solver,
69                                                  duration, i_delay,
    m_delay, initial_g,
70                                                  i_file, e_file,
    start_time)
71
```

```
72          results = [result for result in results if start_time <=
    make_naive(datetime.fromisoformat(result['time'])) <= end_time]
73          real_data = [data for data in real_data if start_time <=
    make_naive(datetime.fromisoformat(data['datetime'])) <= end_time]
74
75          # Calculate residuals
76          residuals, times = calculate_residuals(results, real_data)
77          rms = np.sqrt(np.mean(np.square(residuals)))
78
79          if rms < best_rms:
80              best_rms = rms
81              best_param_value = param_value
82              best_residuals = residuals
83              best_results = results
84              best_times = times
85
86          log.info(f"Iteration {i + 1}/{max_iter}, {parameter}: {
    param_value:.4f}, RMS: {rms:.4f}")
87
88      log.info("")
89      log.info(f"Best value for {parameter}: {best_param_value} with RMS
    : {best_rms}")
90      return best_param_value, best_residuals, best_results, best_times
91
92
93  def evaluate_comparison_metrics(results, filename_simple, start_time,
    end_time):
94      """
95      Computes RMS and MAE between simulated and real glucose levels
96      by matching each real point to the closest simulated point in time
    .
97      Logs the results.
98      """
99      # Read simplified input file
100
101     data = load_json(filename_simple)
102
103     # Parse meals, basal and bolus markers from file
104     real_meals, real_basal, real_bolus = parse_markers(data)
105
106     # Parse SG readings
107     real_results = parse_results_sgs(data.get("sgs", []))
108
109     # Convert dates to naive
110     real_results, real_meals, real_basal, real_bolus = make_naive_all(
    real_results, real_meals, real_basal, real_bolus)
111
112     # Filter by time interval
113     if start_time and end_time:
114         real_results, real_meals, real_basal, real_bolus = filter_all(
    real_results, real_meals, real_basal, real_bolus, start_time,
    end_time, True)
115
116     # Extract glucose levels and timestamps
117     simulated_times = [datetime.fromisoformat(result['time']) for
    result in results]
118     real_times = [result['time'] for result in real_results]
119     simulated_glucose = [result['glucose'] for result in results]
```

```
    real_glucose = [result['glucose'] for result in real_results]

    # Convert to numpy arrays
    simulated_times = np.array(simulated_times)
    real_times = np.array(real_times)
    simulated_glucose = np.array(simulated_glucose)
    real_glucose = np.array(real_glucose)

    # For each real point, find the closest simulated point
    matched_simulated_glucose = []
    matched_simulated_times = []
    for real_time in real_times:
        time_diffs = np.abs((simulated_times - real_time).astype('timedelta64[ms]').astype(float))
        nearest_index = np.argmin(time_diffs)
        matched_simulated_glucose.append(simulated_glucose[nearest_index])
        matched_simulated_times.append(simulated_times[nearest_index])

    matched_simulated_glucose = np.array(matched_simulated_glucose)

    # Compute errors
    errors = matched_simulated_glucose - real_glucose
    rms = np.sqrt(np.mean(errors ** 2))
    mae = np.mean(np.abs(errors))
    max_abs_error = np.max(np.abs(errors))
    bias = np.mean(errors)

    # Compute number of points and timespan based on real_times
    timespan = (real_times[-1] - real_times[0]).total_seconds() / 3600  # in hours
    n_points = len(real_glucose)

    # Log results
    log.info(f"Comparing results for file {filename_simple}:")
    log.info(f"Number of points used: {n_points}")
    log.info(f"Comparison timespan: {timespan:.2f} hours")
    log.info(f"RMS: {rms:.2f} mg/dL")
    log.info(f"MAE: {mae:.2f} mg/dL")
    log.info(f"Max Abs Error: {max_abs_error:.2f} mg/dL")
    log.info(f"Bias: {bias:.2f} mg/dL")
```

Código D.28: residuals.py (DiabetesTool).

Apéndice E

Código fuente del algoritmo de la bomba de insulina

E.1. Estructura

Se incluye el código de la bomba de insulina con comentarios en inglés. El proyecto está diseñado para ejecutarse en una Raspberry Pi con pantalla táctil, simulando el funcionamiento de una bomba real mediante una interfaz gráfica y un algoritmo de control de insulina. La estructura principal es la siguiente:

- `main.py`: Script principal que lanza la aplicación con la interfaz gráfica.
- `app/`: Módulos relacionados con la interfaz gráfica.
  - `app.py`: Inicialización general y gestión de pantallas.
  - `screen.py`: Clases para construir la pantalla principal, imágenes e indicadores.
  - `meal_screen.py`: Pantalla y lógica para administración de bolus por comidas.
  - `options_screen.py`: Pantalla para eventos (ejercicio, alcohol, etc.) y configuración de parámetros.
- `pump/`: Núcleo de lógica funcional del algoritmo de insulina.
  - `basal.py`: Cálculo de insulina basal con control PID.
  - `correction.py`: Cálculo de bolus de corrección ante hiperglucemias.
  - `data.py`: Lectura de datos de glucosa y estimación de insulina activa.
  - `events.py`: Gestión de eventos como ejercicio físico o estado anímico.
- `util/`: Funciones de soporte y utilidades generales.
  - `plot.py`: Generación de gráficos de evolución de glucosa e insulina.
  - `time.py`: Gestión del reloj de simulación y temporizadores.
  - `utils.py`: Funciones auxiliares para lectura y escritura de JSON.
- `data/`: Archivos de configuración de entrada.
  - `config.json`: Parámetros iniciales del sistema.
  - `events.json`: Lista de eventos definidos por el usuario.
  - `params.json`: Parámetros fisiológicos del modelo de control.
- `img/`: Recursos gráficos utilizados por la interfaz.

E.2. main.py

```
import tkinter as tk
from app.app import App

if __name__ == "__main__":
    root = tk.Tk()
    app = App(root)
    root.mainloop()
```

Código E.1: main.py (Pump algorithm).

E.3. app

E.3.1. app.py

```

from datetime import timedelta
import logging as log
import sys
import time
from tkinter import Button, Frame, IntVar, Label, Spinbox
from PIL import Image, ImageTk
from app.meal_screen import MealScreen
from app.options_screen import OptionsScreen
from pump.basal import basalDelivery, predict_glucose
from pump.correction import correctionBolus
from pump.data   import handle_pumpdataupdate, applyAI_from_inputs,
    last_correcion_in_data
from app.screen import SCREEN_SIZE, X_BUTTON, X_SCREEN, Y_INIT_B,
    Y_SEP_B, PumpImg, PumpScreen, PumpLabel, lcd
from util.plot import setup_graph
from util.utils import FONT_MONT_14, FONT_MONT_16, FONT_MONT_20,
    FONT_MONT_26, FONT_MONT_28, FONT_MONT_48,  align_text,
    set_flag_true
from util.time  import S_2_MS, TIMER1_PERIOD_S, TIMER2_PERIOD_S,
    CurrentTime, TimerSch, handle_timeupdate, synchronize
from app.screen import sensor_age_text, time_to_calib_progress

log.basicConfig(format='[%(asctime)s:%(levelname)s] %(message)s',
    datefmt='%Y-%m-%d %H:%M:%S', level=log.INFO)

class App:
    """Main application controller that initializes the GUI and
    handles events, timers, and screen updates."""

    def __init__(self, root):
        """Initialize the App instance and its core UI components."""
        self.root = root
        self.root.title("Minimed Mon PC")

        self.update_sim_time_id = None
```

```
30          self.update_id = None
31
32          self.root.protocol("WM_DELETE_WINDOW", self.on_close)
33
34          self.initialize_screen()
35          self.initialize_images()
36          self.initialize_labels()
37          self.initialize_text()
38          self.initialize_buttons()
39          self.initialize_timers()
40          self.active_canvases = []
41          self.eIntegral = 0
42          self.lastActiveInsulin = 0
43          self.errorHistory = []
44          self.add_buttons_with_images()
45
46          self.speed.set(1)
47          self.update_sim_time()
48          self.update()
49
50      def initialize_screen(self):
51          """Initialize the pump screen and LCD canvas layout."""
52          print("Creating screen")
53          self.screen = PumpScreen(self.root, 'img/icon_mmm.png')
54          self.scr1 = self.screen.scr
55          self.lcd = lcd(parent=self.scr1)
56
57      def initialize_images(self):
58          """Load and position all graphical image elements (battery,
    reservoir, sensor, etc.)."""
59          print("Create images")
60          self.imageBattery     = PumpImg("img/mm_batt_unk.png", x=6, y
    =0, parent=self.scr1)
61          self.imageReservoir   = PumpImg("img/mm_tank_unk.png", x=40, y
    =0, parent=self.scr1)
62          self.imageSensorConn  = PumpImg("img/mm_sensor_connection_nok.
    png", x=68, y=0, parent=self.scr1)
63          self.imageDrop        = PumpImg("img/mm_drop_unk.png", x=105,
    y=8, parent=self.scr1)
64          self.imageSage        = PumpImg("img/mm_sage_unk.png", x=135,
    y=0, parent=self.scr1)
65          self.imageShield      = PumpImg("img/mm_shield_none.png", x
    =65, y=33, parent=self.scr1)
66          self.imageBanner      = PumpImg("img/
    mm_banner_delivery_suspend.png", x=40, y=145, parent=self.scr1)
67          self.imageBanner.set_hidden(True)
68
69      def initialize_labels(self):
70          """Create all display labels for glucose, insulin, time, etc.
    """
71          print("Create labels")
72          self.labelBglValue    = PumpLabel('--', x=160, y=95, color=0
    xffffff, font=FONT_MONT_48, parent=self.scr1)
73          self.labelBglUnit     = PumpLabel('mg/dL', x=135, y=150, color
    =0x89abeb, font=FONT_MONT_16, parent=self.scr1)
74          self.labelActInsValue = PumpLabel('-- U', x=261, y=173, color
    =0xffffff, font=FONT_MONT_26, parent=self.scr1)
```

```
75          self.labelActIns      = PumpLabel('Act Insulin', x=231, y=200,
     color=0xffffff, font=FONT_MONT_16, parent=self.scr1)
76          self.labelActBasValue = PumpLabel('-- U/h', x=10, y=173, color
    =0xffffff, font=FONT_MONT_16, parent=self.scr1)
77          self.labelActBas      = PumpLabel('Act basal', x=10, y=200,
    color=0xffffff, font=FONT_MONT_16, parent=self.scr1)
78          self.labelTime        = PumpLabel('--:--', x=250, y=0, color=0
    xffffff, font=FONT_MONT_28, parent=self.scr1)
79          self.labelLastData    = PumpLabel('--', x=160, y=218, color=0
    xffffff, font=FONT_MONT_20, parent=self.scr1)
80          self.labelSage        = PumpLabel('', x=144, y=10, color=0
    xffffff, font=FONT_MONT_14, parent=self.scr1)
81
82      def initialize_buttons(self):
83          """Create control buttons including time sync, date, and speed
     settings."""
84          print("Adding buttons")
85          self.create_time_sync_button()
86          self.create_date_label()
87          self.create_speed_control()
88
89      def create_time_sync_button(self):
90          """Create a button to manually synchronize the simulation time
    ."""
91          self.buttonBattery = Button(self.root, text="Synchronize Time"
    , command=lambda: self.sync())
92          self.place_widget(self.buttonBattery, 0)
93
94      def sync(self):
95          """Synchronize the simulation time and reset control variables
    ."""
96          self.date.set_time(synchronize(self.timer, self.speed.get()))
97          self.eIntegral = 0
98          self.errorHistory  = []
99          self.dateOld   = None
100
101     def create_date_label(self):
102         """Create and place the label that displays the current
    simulation time."""
103         self.date = CurrentTime()
104         self.dateOld = None
105         self.label_date = Label(self.root, text=self.date.getTimeText
    ())
106         self.place_widget(self.label_date, 1)
107
108     def create_speed_control(self):
109         """Create a spinbox to control the simulation speed."""
110         self.speed = IntVar()
111         self.speed.set(1)
112         label_speed = Label(self.root, text="Speed:")
113         self.place_widget(label_speed, 2, x_offset=-50)
114         self.spinbox = Spinbox(self.root, from_=1, to=100000,
    increment=1, textvariable=self.speed, width=8)
115         self.place_widget(self.spinbox, 2)
116
117     def place_widget(self, widget, position, x_offset=0, width=None,
    height=None):
```

```
118         """Place a widget in a grid-like position with optional size
    adjustments."""
119         x = X_BUTTON + x_offset
120         y = Y_INIT_B + position * Y_SEP_B
121         widget.place(x=x, y=y, width=width, height=height)
122 
123     def initialize_text(self):
124         """Align static text labels such as units and section titles.
    """
125         align_text(self.labelBglUnit,"center",150)
126         align_text(self.labelActIns,"right",200)
127         align_text(self.labelActBas,"left",200)
128 
129 
130     def initialize_timers(self):
131         """Set up the simulation timers and initialize their state."""
132         print("Initializing timers")
133         self.lastUpdateTm        = None
134         self.is_time_update      = False
135         self.is_pump_data_update = False
136         self.timer = TimerSch(date=self.date.getTime())
137         self.timer.run(set_flag_true, "is_time_update",      self,
    TIMER1_PERIOD_S*S_2_MS, False)
138         self.timer.run(set_flag_true, "is_pump_data_update", self,
    TIMER2_PERIOD_S*S_2_MS, False)
139 
140         # Run some timer functions immediately to init timer_flags
141         set_flag_true("is_time_update",      self)
142         set_flag_true("is_pump_data_update", self)
143         set_flag_true("applyAI",             self)
144         print("Time and date successfully synched")
145 
146     def add_buttons_with_images(self):
147         """Add four square buttons with images that generate large
    square screens below."""
148         print("Adding pump buttons with images")
149 
150         button_size = 50
151         y_position_buttons = 3
152 
153         image_files = [
154             "img/icon1.png",
155             "img/icon2.png",
156             "img/icon3.png",
157             "img/icon4.png"
158         ]
159 
160         self.screens = []  #
161         self.inactivity_timer = None
162 
163         for i, img_path in enumerate(image_files):
164             img_original = Image.open(img_path)
165             img_resized = img_original.resize((button_size,
    button_size), Image.LANCZOS)
166             img = ImageTk.PhotoImage(img_resized)
167 
168             button = Button(
169                 self.root,
```

```
                image=img,
                command=lambda idx=i: self.create_large_screen(idx),
                width=button_size,
                height=button_size,
                bd=0,
                highlightthickness=0
            )
            button.image = img
            self.place_widget(button, position=y_position_buttons,
    x_offset=i * (button_size + 10) - 100)

            screen = Frame(self.root, width=SCREEN_SIZE, height=
    SCREEN_SIZE)
            screen.place_forget()
            self.screens.append(screen)

    def create_large_screen(self, index):
        """Show the large square screen corresponding to the clicked
    button and configure its content."""
        self.reset_inactivity_timer()

        for screen in self.screens:
            screen.place_forget()

        self.screens[index].place(x=X_SCREEN, y=Y_INIT_B + 5 * Y_SEP_B
    )

        if index == 0:
            canvas = setup_graph(self.screens[index], self.date.
    getTime(), SCREEN_SIZE)
            self.active_canvases.append(canvas)
        elif index == 1:
            applyAI = applyAI_from_inputs(self.date.getTime())
            MealScreen(self.screens[index], self.lastSG, self.date,
    self.activeInsulin, applyAI, SCREEN_SIZE)
        elif index == 2:
            OptionsScreen(self.screens[index], 'data/events.json',
    [5,5,5], SCREEN_SIZE, "events")
        elif index == 3:
            OptionsScreen(self.screens[index], 'data/config.json',
    [5,5,5], SCREEN_SIZE, "config")


    def reset_inactivity_timer(self):
        """Reset the inactivity timer to hide the screen after 30
    seconds."""
        if self.inactivity_timer:
            self.root.after_cancel(self.inactivity_timer)  #
        self.inactivity_timer = self.root.after(60000, self.
    hide_all_screens)

    def hide_all_screens(self):
        """Hide all screens due to inactivity."""
        print("Hiding all screens due to inactivity")
        for screen in self.screens:
            screen.place_forget()

    def on_close(self):
```

```
218         """Gracefully shut down the application and clean up all
    timers and canvases."""
219         try:
220             if hasattr(self, 'update_sim_time_id') and self.
    update_sim_time_id:
221                 self.root.after_cancel(self.update_sim_time_id)
222             if hasattr(self, 'update_id') and self.update_id:
223                 self.root.after_cancel(self.update_id)
224             if hasattr(self, 'inactivity_timer') and self.
    inactivity_timer:
225                 self.root.after_cancel(self.inactivity_timer)
226 
227             if hasattr(self, 'timer') and self.timer:
228                 self.timer.cancel_all_timers()  #
229 
230             for canvas in self.active_canvases:
231                 try:
232                     canvas.get_tk_widget().destroy()
233                     canvas.figure.clear()
234                 except Exception as e:
235                     log.error(f"Error clearing Matplotlib canvas: {e}"
    )
236 
237             self.active_canvases.clear()
238 
239         except AttributeError as e:
240             log.error(f"Error accessing an attribute during cleanup: {
    e}")
241         except Exception as e:
242             log.error(f"Error during cleanup: {e}")
243         finally:
244             # Cierra la ventana principal y finaliza el programa
245             try:
246                 self.root.destroy()
247             except Exception as e:
248                 log.error(f"Error destroying root window: {e}")
249             finally:
250                 time.sleep(1)
251                 sys.exit()
252 
253     def update_sim_time(self):
254         """Update the simulated date and check the timers."""
255         increment = timedelta(seconds=self.speed.get())
256         self.date.increment(increment)
257         self.label_date.config(text=self.date.getTimeText())
258 
259         self.timer.update_date(self.date.getTime())
260         self.timer.check_timers()
261         self.update_sim_time_id = self.root.after(1000, self.
    update_sim_time)
262 
263     def data_update(self):
264         """Refresh the UI with current sensor and insulin data values.
    """
265         haveData = True
266         if haveData:
267             self.imageBattery.set_img_src("img/mm_batt100.png")
268             self.imageReservoir.set_img_src("img/mm_tank200.png")
```

```
269             self.imageSage.set_img_src("img/mm_sage_green.png")
270             self.labelSage.set_text(sensor_age_text(100))
271         else:
272             self.imageBattery.set_img_src("img/mm_batt_unk.png")
273             self.imageReservoir.set_img_src("img/mm_tank_unk.png")
274             self.imageSage.set_img_src("img/mm_sage_unk.png")
275             self.labelSage.set_text("")
276 
277         self.imageSensorConn.set_img_src("img/mm_sensor_connection_ok.
    png")
278         time_to_calib_progress(100,"OK","OK", self.imageDrop, self.lcd
    )
279 
280         self.imageShield.set_img_src("img/mm_shield_none.png")
281         self.imageShield.set_hidden(False)
282 
283         if self.lastSG is None or self.lastSG < 0:
284             lastSG_str = "--"
285         else:
286             lastSG_str = str(self.lastSG)
287         self.labelBglValue.set_text(lastSG_str)
288         align_text(self.labelBglValue,"center",95)
289 
290         if self.activeInsulin is None or self.activeInsulin < 0:
291             activeInsulin_str = "-- U"
292         else:
293             activeInsulin_str = str(round(self.activeInsulin, 1))+" U"
294         self.labelActInsValue.set_text(activeInsulin_str)
295         align_text(self.labelActInsValue,"right",173)
296 
297         if self.bR is None or self.bR < 0:
298             bR_str = "-- U/h"
299         else:
300             bR_str = str(round(self.bR, 2))+" U/h"
301         self.labelActBasValue.set_text(bR_str)
302         align_text(self.labelActBasValue,"left",173)
303 
304         self.imageBanner.set_hidden(True)
305 
306 
307     def update(self):
308         """Main update loop that processes pump data and simulation
    time every 100 ms."""
309 
310         # Check if it's time to process new pump data
311         if self.is_pump_data_update:
312 
313             # Retrieve latest sensor glucose, timestamp and active
    insulin
314             self.lastSG, self.lastUpdateTm, self.activeInsulin =
    handle_pumpdataupdate()
315 
316             # If any of the retrieved values are missing, fallback to
    data update and skip correction
317             # This occurs in the very first update
318             if self.lastUpdateTm is None or self.lastSG is None or
    self.activeInsulin is None:
319                 self.data_update()
```

```
320                 self.dateOld    = self.date.getTime()
321                 self.is_pump_data_update = False
322 
323             else:
324 
325                 # Compute basal delivery using PID controller
326                 basal, self.bR, eIntegral, errorRate = basalDelivery(
    self.date.getTime(), self.dateOld, self.lastSG, self.errorHistory,
     self.eIntegral)
327 
328                 # Update visual and insulin-related elements
329                 self.data_update()
330 
331                 # Predict future glucose level based on current rate
    of change
332                 predicted_glucose = predict_glucose(self.lastSG,
    errorRate)
333                 # Check if the last correction bolus is considered or
    missed (if missed, do not apply a correction bolus yet)
334                 considerLastCorrection = last_correcion_in_data(self.
    dateOld, self.activeInsulin, self.lastActiveInsulin)
335 
336                 # If glucose is predicted to remain above hypo
    threshold and last correction is already considered
337                 # Compute and apply a correction Bolus if needed
338                 if predicted_glucose > 70 and self.dateOld is not None
     and considerLastCorrection:
339                     correctionBolus(self.date.getTime(), self.lastSG,
    self.activeInsulin, basal)
340 
341                 # Update PID memory state
342                 self.eIntegral           = eIntegral
343                 self.dateOld             = self.date.getTime()
344                 self.lastActiveInsulin   = self.activeInsulin
345                 self.is_pump_data_update = False
346 
347         # Check if the time display needs to be updated
348         if self.is_time_update:
349             handle_timeupdate(self.date.getTime(),  self.labelTime,
    self.labelLastData, self.lastUpdateTm)
350             self.is_time_update = False
351 
352         # Re-schedule the next update
353         self.update_id = self.root.after(100, self.update)
```

Código E.2: app.py (Pump algorithm).

E.3.2. meal_screen.py

```
1 import logging as log
2 from datetime import datetime
3 from tkinter import Frame, Label, Button, Entry, StringVar
4 from pump.correction import mealBolus
5 from util.utils import INPUTS, INPUTS_PATH, read_data_json, writeJson
6 
```

```
class MealScreen:
    """Displays the meal input interface and handles bolus calculation based on glucose and insulin levels."""
    def __init__(self, screen, glucose_value, currentTime, activeInsulin, apply_active_insulin, screen_size=200):
        self.screen = screen
        self.glucose_value = glucose_value
        self.screen_size = screen_size
        self.currentTime = currentTime
        self.bolus = 0
        self.activeInsulin = activeInsulin
        self.applyAI = apply_active_insulin
        self.setup_meal_screen()

    def setup_meal_screen(self):
        """Set up the meal screen layout."""
        col_width = self.screen_size // 3

        # Configure the screen size and background color
        self.screen.config(width=self.screen_size, height=self.screen_size, bg="black")

        # Row 1: Current glucose value
        Label(
            self.screen,
            text=f"{self.glucose_value} mg/dL",
            font=("Arial", 14),
            bg="black",
            fg="white"
        ).grid(row=0, column=0, columnspan=3, pady=5, sticky="w")

        # Row 2: CH input
        self.ch_var = StringVar(value="0.0")
        Label(
            self.screen,
            text="Carbs",
            font=("Arial", 12),
            bg="black",
            fg="white"
        ).grid(row=1, column=0, padx=10, pady=3, sticky="w")
        self.ch_entry = Entry(self.screen, font=("Arial", 12), width=5, textvariable=self.ch_var)
        self.ch_entry.grid(row=1, column=1, padx=5, pady=3, ipadx=col_width // 10)
        self.ch_result = Label(
            self.screen,
            text="--",
            font=("Arial", 12),
            bg="black",
            fg="white"
        )
        self.ch_result.grid(row=1, column=2, padx=10, pady=3, sticky="w")

        # Row 3: UGP input
        self.ugp_var = StringVar(value="0.0")
        Label(
            self.screen,
```

```
59              text="UGPs",
60              font=("Arial", 12),
61              bg="black",
62              fg="white"
63          ).grid(row=2, column=0, padx=10, pady=3, sticky="w")
64          self.ugp_entry = Entry(self.screen, font=("Arial", 12), width
    =5, textvariable=self.ugp_var)
65          self.ugp_entry.grid(row=2, column=1, padx=5, pady=3, ipadx=
    col_width // 10)
66          self.ugp_result = Label(
67              self.screen,
68              text="--",
69              font=("Arial", 12),
70              bg="black",
71              fg="white"
72          )
73          self.ugp_result.grid(row=2, column=2, padx=10, pady=3, sticky=
    "w")
74
75          # Row 4: Adjustment row
76          Label(
77              self.screen,
78              text="Adjustment",
79              font=("Arial", 12),
80              bg="black",
81              fg="white"
82          ).grid(row=3, column=0, padx=10, pady=3, sticky="w")
83          Label(
84              self.screen,
85              text="",
86              font=("Arial", 12),
87              bg="black"
88          ).grid(row=3, column=1, padx=10, pady=3)
89          self.adjustment_value = Label(
90              self.screen,
91              text="--",
92              font=("Arial", 12),
93              bg="black",
94              fg="white"
95          )
96          self.adjustment_value.grid(row=3, column=2, padx=10, pady=3,
    sticky="w")
97
98          # Row 5: Bolus row
99          Label(
100             self.screen,
101             text="Bolus",
102             font=("Arial", 12),
103             bg="black",
104             fg="white"
105         ).grid(row=4, column=0, padx=10, pady=3, sticky="w")
106         Label(
107             self.screen,
108             text="",
109             font=("Arial", 12),
110             bg="black"
111         ).grid(row=4, column=1, padx=10, pady=3)
112         self.bolus_value = Label(
```

```
113             self.screen,
114             text="--",
115             font=("Arial", 12),
116             bg="black",
117             fg="white"
118         )
119         self.bolus_value.grid(row=4, column=2, padx=10, pady=3, sticky
    ="w")
120 
121         # Button: Deliver Bolus
122         Button(
123             self.screen,
124             text="Deliver Bolus",
125             font=("Arial", 12),
126             command=lambda: self.administer_bolus(self.currentTime)
127         ).grid(row=5, column=0, columnspan=3, padx=50, pady=5,  sticky
    ="w", ipadx=col_width // 5)
128 
129         self.update_calculations()
130 
131         # Bind CH entry changes to update calculations
132         self.ch_entry.bind( "<KeyRelease>", self.update_calculations)
133         self.ugp_entry.bind("<KeyRelease>", self.update_calculations)
134 
135     def update_calculations(self, event=None):
136         """Update the dependent values based on CH input."""
137         try:
138             ch_value = float(self.ch_entry.get())
139         except ValueError:
140             ch_value = 0.0
141         try:
142             ugp_value = float(self.ugp_entry.get())
143         except ValueError:
144             ugp_value = 0.0
145 
146         self.result, self.adjustment, self.bolus, self.ugps =
    mealBolus(ch_value, ugp_value, self.currentTime.getTime(), self.
    glucose_value, self.activeInsulin, self.applyAI)
147 
148         result_ugp = round(sum([ugp['ugp_bolus'] for ugp in self.ugps
    ]), 1) if self.ugps else 0.0
149 
150         self.ch_result.config(text=f"{self.result} U")
151         self.ugp_result.config(text=f"{result_ugp} U")
152         self.adjustment_value.config(text=f"{self.adjustment} U")
153         self.bolus_value.config(text=f"{self.bolus} U")
154 
155     def administer_bolus(self, currentTime):
156         """Callback to administer the calculated bolus."""
157 
158         bolus  = float(self.bolus )
159         result = float(self.result)
160 
161         carbs = self.ch_entry.get()
162         try:
163             carbs = float(carbs)
164         except ValueError:
165             carbs = 0.0
```

```
166
167         # Read and update the inputs JSON
168         data = read_data_json(INPUTS_PATH)
169         if data is None:
170             data = {key: [] for key in INPUTS}
171
172         # Add the meal and bolus entries
173         bolus_entry = {"start": currentTime.getTimeText(), "i": bolus,
     "type": "Meal", "carbs": carbs, "i_result": result}
174         data['bolus'].append(bolus_entry)
175
176         for ugp in self.ugps:
177             ugp_entry = {
178                 "start": ugp['ugp_time'].strftime('%Y-%m-%dT%H:%M:%S')
    ,
179                 "i": ugp['ugp_bolus'],
180                 "type": "UGP",
181                 "ugp_num": ugp['ugp_num']
182             }
183             data['bolus'].append(ugp_entry)
184
185         # Write updated data back to the file
186         writeJson(data, INPUTS_PATH)
187
188         print(f"Bolus saved: {bolus_entry}")
```

Código E.3: meal_screen.py (Pump algorithm).

E.3.3. options_screen.py

```
1  from tkinter import Frame, Listbox, Button, Entry, Label, OptionMenu,
       StringVar
2  from tkinter.ttk import Style, Treeview
3  from util.utils import read_data_json, writeJson
4
5  class OptionsScreen:
6      """Handles the display and interaction logic for an options/
       settings screen."""
7      def __init__(self, screen, json_file, column_widths, screen_size,
       config):
8          self.screen = screen
9          self.json_file = json_file
10         self.column_widths = column_widths
11         self.screen_size = screen_size
12         self.parameters = {}
13         self.current_param_frame = None
14         self.main_listbox = None
15         self.config = config
16         self.setup_options()
17
18     def setup_options(self):
19         """Set up the main options screen with a list of parameters.
       """
20         self.parameters = read_data_json(self.json_file)
21
```

```
22          # Create a styled listbox to display parameter names
23          self.main_listbox = Listbox(
24              self.screen,
25              bg="black",
26              fg="white",
27              font=("Arial", 13),
28              selectbackground="white",
29              selectforeground="white",
30              highlightthickness=0
31          )
32          for key in self.parameters:
33              self.main_listbox.insert("end", self.parameters[key]['name
    '])
34          self.main_listbox.place(x=0, y=0, width=self.screen_size,
    height=self.screen_size)
35
36          # Bind the listbox selection to show the parameter details
37          self.main_listbox.bind("<<ListboxSelect>>", self.
    show_parameter_screen)
38
39      def show_parameter_screen(self, event):
40          """Replace the current screen with details for the selected
    parameter."""
41          selected_index = self.main_listbox.curselection()
42          if not selected_index:
43              return
44
45          selected_key = list(self.parameters.keys())[selected_index[0]]
46          selected_param = self.parameters[selected_key]
47
48          # Clear previous content
49          self.main_listbox.place_forget()
50          if self.current_param_frame:
51              self.current_param_frame.destroy()
52
53          self.current_param_frame = Frame(self.screen, bg="black")
54          self.current_param_frame.place(x=0, y=0, width=self.
    screen_size, height=self.screen_size)
55
56          Label(
57              self.current_param_frame,
58              text=selected_param['name'],
59              font=("Arial", 12),
60              bg="black",
61              fg="white"
62          ).place(x=10, y=10)
63
64          if "period" in selected_param:
65              self.setup_treeview(selected_param)
66          else:
67              self.setup_simple_entry(selected_param)
68
69      def setup_treeview(self, selected_param):
70          """Set up a Treeview for parameters with a period."""
71          if self.config == "config":
72              columns = ("From", "To", "Value")
73              new_period = {"from": "00:00", "to": "24:00", "value": "0"
    }
```

```
74          else:
75              first_field, second_field = "Start", "Duration"
76              if not selected_param['period']:
77                  third_field = selected_param.get('unit', 'None')
78              else:
79                  third_field = list(selected_param['period'][0].keys())
    [-1]
80              columns = (first_field, second_field, third_field.
    capitalize())
81              new_period = {"start": "2025-01-01T00:00:00", "duration":
    "30", third_field: ""}
82
83          style = Style()
84          style.configure("Custom.Treeview",
85                          background="black",
86                          foreground="white",
87                          fieldbackground="blue",
88                          bordercolor="white",
89                          rowheight=15)
90
91          style.configure("Custom.Treeview.Heading",
92                          background="grey",
93                          foreground="black",
94                          font=("Arial", 10))
95
96          style.map("Custom.Treeview",
97                  background=[("selected", "blue")],
98                  foreground=[("selected", "white")])
99
100         tree = Treeview(self.current_param_frame, columns=columns,
    show="headings", style="Custom.Treeview")
101         for col in columns:
102             tree.heading(col, text=col)
103             tree.column(col, width=int(self.column_widths[columns.
    index(col)]), anchor="center")
104
105         # Insert rows
106         for period in selected_param['period']:
107             values = [period.get(key.lower(), "") for key in columns]
108             tree.insert("", "end", values=values)
109
110         tree.place(x=10, y=40, width=self.screen_size - 20, height=
    self.screen_size - 100)
111
112         # Add buttons for add and remove
113         add_button = Button(
114             self.current_param_frame,
115             text="Add",
116             font=("Arial", 10),
117             command=lambda: self.add_period(tree, selected_param,
    new_period),
118             bg="#2E2E2E",  # Dark gray
119             fg="#FFFF00",  # Yellow
120             highlightbackground="#FFFF00",  # Yellow
121             highlightthickness=10
122         )
123         add_button.place(x=10, y=self.screen_size - 50, width=(self.
    screen_size - 30) // 3, height=30)
```

```
124
125         remove_button = Button(
126             self.current_param_frame,
127             text="Remove",
128             font=("Arial", 10),
129             command=lambda: self.remove_selected(tree, selected_param)
    ,
130             bg="#2E2E2E",  # Dark gray
131             fg="#FFFF00",  # Yellow
132             highlightbackground="#FFFF00",  # Yellow
133             highlightthickness=10
134         )
135         remove_button.place(x=(self.screen_size // 3) + 5, y=self.
    screen_size - 50, width=(self.screen_size - 30) // 3, height=30)
136
137         tree.bind("<Double-1>", lambda event: self.edit_period(event,
    tree, selected_param))
138
139     def setup_simple_entry(self, selected_param):
140         """Create a simple entry for parameters without periods."""
141
142         y_offset = 40
143         entries = {}  # Dictionary to keep track of Entry widgets and
    their keys
144
145         if "value" in selected_param and "unit" in selected_param:
146             # If "value" and "unit" are present, show a single entry
    with a unit label
147             value_entry = Entry(
148                 self.current_param_frame,
149                 font=("Arial", 10),
150                 width=5
151             )
152             value_entry.insert(0, selected_param['value'])
153             value_entry.place(x=10, y=y_offset)
154
155             Label(
156                 self.current_param_frame,
157                 text=selected_param['unit'],
158                 font=("Arial", 10),
159                 bg="black",
160                 fg="white"
161             ).place(x=120, y=y_offset)
162
163             entries['value'] = value_entry
164             y_offset += 30
165
166         else:
167             # For other records, create a label and entry for each key
    -value pair
168             for key, value in selected_param.items():
169                 if key != "name":
170                     Label(
171                         self.current_param_frame,
172                         text=key.capitalize(),
173                         font=("Arial", 10),
174                         bg="black",
175                         fg="white"
```

```
                    ).place(x=10, y=y_offset)

                    value_entry = Entry(
                        self.current_param_frame,
                        font=("Arial", 8),
                        width=50
                    )
                    value_entry.insert(0, value)
                    value_entry.place(x=70, y=y_offset)

                    entries[key] = value_entry
                    y_offset += 30

        # Add a save button
        save_button = Button(
            self.current_param_frame,
            text="Save",
            font=("Arial", 10),
            command=lambda: self.save_generic_values(entries,
    selected_param),
            bg="#2E2E2E",  # Dark gray background
            fg="#FFFF00",  # Yellow text
            highlightbackground="#FFFF00",  # Yellow border
            highlightthickness=2  # Border thickness
        )
        save_button.place(x=10, y=y_offset + 10, width=100, height=30)

    def save_generic_values(self, entries, selected_param):
        """Save all updated values for a generic record."""
        for key, entry in entries.items():
            selected_param[key] = entry.get()

        writeJson(self.parameters, self.json_file)
        self.setup_options()

    def add_period(self, tree, selected_param, new_period):
        """Add a new period to the parameter and refresh the treeview.
    """
        selected_param['period'].append(new_period)
        values = [new_period.get(key.lower(), "") for key in tree['
    columns']]
        tree.insert("", "end", values=values)
        writeJson(self.parameters, self.json_file)
        self.show_parameter_screen(None)

    def remove_selected(self, tree, selected_param):
        """Remove the selected period from the parameter."""
        selected_item = tree.selection()
        if selected_item:
            item_index = tree.index(selected_item[0])
            del selected_param['period'][item_index]
            tree.delete(selected_item[0])
            writeJson(self.parameters, self.json_file)

    def edit_period(self, event, tree, selected_param):
        """Enable editing of a selected cell in the Treeview."""
        item_id = tree.identify_row(event.y)
        column_id = tree.identify_column(event.x)
```

```
231         col = int(column_id[1:]) - 1
232 
233         if not item_id or col < 0:
234             return
235 
236         # Get current value and create an Entry or OptionMenu widget
237         current_value = tree.item(item_id, "values")[col]
238         options = selected_param.get("options", [])
239 
240         if options and col == 2:  # Use OptionMenu for specific
    columns with options
241             var = StringVar(value=current_value)
242             option_menu = OptionMenu(self.current_param_frame, var, *
    options)
243             option_menu.place(x=event.x, y=event.y)
244 
245             def save_option(*args):
246                 new_value = var.get()
247                 option_menu.destroy()
248 
249                 values = list(tree.item(item_id, "values"))
250                 values[col] = new_value
251                 tree.item(item_id, values=values)
252 
253                 # Update the JSON data
254                 item_index = tree.index(item_id)
255                 if self.config == "config":
256                     keys = ['from','to','value']
257                 else:
258                     keys = ['start','duration'] + list(selected_param[
    'period'][0].keys())[2:]
259 
260                 selected_param['period'][item_index][keys[col]] =
    new_value
261                 writeJson(self.parameters, self.json_file)
262 
263             var.trace_add("write", save_option)  # Use trace_add
    instead of trace for modern tkinter
264         else:
265             entry = Entry(self.current_param_frame)
266             entry.insert(0, current_value)
267             entry.place(x=event.x, y=event.y)
268             entry.focus_set()
269 
270             def save_edit(event):
271                 new_value = entry.get()
272                 entry.destroy()
273 
274                 values = list(tree.item(item_id, "values"))
275                 values[col] = new_value
276                 tree.item(item_id, values=values)
277 
278                 # Update the JSON data
279                 item_index = tree.index(item_id)
280                 if self.config == "config":
281                     keys = ['from','to','value']
282                 else:
```

```
283                     keys = ['start','duration'] + list(selected_param[
    'period'][0].keys())[2:]
284
285                 selected_param['period'][item_index][keys[col]] =
    new_value
286                 writeJson(self.parameters, self.json_file)
287
288             entry.bind("<Return>", save_edit)
```

Código E.4: options_screen.py (Pump algorithm).

E.3.4. screen.py

```
1  # Code adapted from: https://github.com/ondrej1024/m5-minimed-monitor
2
3  from tkinter import *
4  from tkinter import Canvas, PhotoImage
5
6  from util.utils import WINBORDER, WINHEIGHT, WINWIDTH
7
8  from PIL import Image, ImageDraw
9
10 X_BUTTON = 520
11 X_SCREEN = 432
12 Y_INIT_B = 5
13 Y_SEP_B  = 30
14 SCREEN_SIZE = 200
15
16 class PumpScreen:
17     """Graphical wrapper for the main pump window with a central
    canvas."""
18     def __init__(self,root, icon=None):
19         self.window = root
20         self.window.config(width=WINWIDTH+WINBORDER,height=WINHEIGHT+
    WINBORDER)
21         self.window.configure(bg='black')
22         self.window.title("Minimed Mon PC")
23         self.window.resizable(False, False)
24         if icon != None:
25             self.window.iconphoto(False, PhotoImage(file=icon))
26         self.scr = Canvas(self.window, width=WINWIDTH,height=WINHEIGHT
    ,bg='black',highlightthickness=0)
27         self.scr.place(relx=0.5, rely=0.5, anchor=CENTER)
28
29 class PumpImg:
30     """Manages an image placed on a canvas with visibility and source
    control."""
31     def __init__(self, img_file, x, y, parent):
32         self.img = PhotoImage(file=img_file)
33         self.scr = parent
34         self.img_h = self.scr.create_image(x, y, anchor=NW, image=self
    .img)
35     def set_hidden(self, hidden):
36         """Set image visibility."""
37         state = 'hidden' if hidden else 'normal'
```

```
38          self.scr.itemconfig(self.img_h, state=state)
39      def set_img_src(self, img_file):
40          """Change the image source."""
41          self.img = PhotoImage(file=img_file)
42          self.scr.itemconfig(self.img_h, image=self.img)
43      def set_pos(self, x, y):
44          """Set image position on canvas."""
45          pass
46
47  class PumpLabel:
48      """Text label component drawn on a canvas with position and size
        control."""
49      def __init__(self, text, x, y, color, font, parent):
50          self.scr = parent
51          self.txt_h = self.scr.create_text(x,y,anchor=NW,text=text,fill
        ="#%06X" % (color),font=(font))
52      def set_text(self, text):
53          """Change the label text."""
54          self.scr.itemconfig(self.txt_h,text=text)
55      def set_pos(self, x, y):
56          """Move the label to a new position."""
57          self.scr.coords(self.txt_h,x,y)
58      def get_width(self):
59          """Get the label's width in pixels."""
60          bounds = self.scr.bbox(self.txt_h)
61          return bounds[2] - bounds[0]
62      def get_height(self):
63          """Get the label's height in pixels."""
64          bounds = self.scr.bbox(self.txt_h)
65          return bounds[3] - bounds[1]
66
67  class lcd:
68      """Simulated LCD drawing functions for arcs and circular
        indicators."""
69      def __init__(self,parent):
70          self.scr = parent
71      def arc(self, x, y, radius, thick, startpos, endpos, color1,
        color2):
72          """Draw an arc representing a partial or full circle."""
73          x1 = x-radius
74          y1 = y-radius
75          x2 = x+radius
76          y2 = y+radius
77          start = (360-startpos)+90
78          extend = -endpos
79          outline = "#%06X" % (color1)
80          arc = self.scr.create_arc((x1,y1,x2,y2), start=start, extent=
        extend, style=ARC, width=thick, outline=outline)
81
82  def sensor_age_text(rem_hours):
83      """Format sensor remaining hours into a text value for display."""
84      if rem_hours == 255:
85          text = ""
86      elif rem_hours > 9:
87          text = str(round(rem_hours/24))
88      else:
89          text = str(rem_hours)
90      return text
```

```
91
92  def time_to_calib_progress(ttc,sst,cst, imageDrop, lcd):
93      """Draw calibration countdown progress as a circular arc and drop
    icon."""
94      centerX = 112
95      centerY = 17
96      radius  = 14
97      thick   = 4
98      endposfull = 359
99      endpos = int(360*((12-ttc)/12))
100     endpos = min(endpos,endposfull)
101     endpos = max(endpos,0)
102     if ttc == 255 or cst == "UNKNOWN": # unknown
103         # full blue circle, question mark
104         imageDrop.set_img_src("img/mm_drop_unk.png")
105         imageDrop.set_pos(106, 8)
106         lcd.arc(centerX, centerY, radius, thick, 0, endposfull,0
    x00cccc,0x00cccc)
107     elif ttc >= 12:
108         # full green circle, white drop
109         imageDrop.set_img_src("img/mm_drop_white.png")
110         imageDrop.set_pos(105, 8)
111         lcd.arc(centerX, centerY, radius, thick, 0, endposfull,0
    x33cc00,0x33cc00)
112     elif ttc > 3:
113         # decreasing green circle, white drop
114         imageDrop.set_img_src("img/mm_drop_white.png")
115         imageDrop.set_pos(105, 8)
116         lcd.arc(centerX, centerY, radius, thick, 0, endposfull,0
    x33cc00,0x33cc00)
117         lcd.arc(centerX, centerY, radius, thick, 0, endpos,0x000000,0
    x000000)
118     elif ttc > 0:
119         # decreasing red circle, white drop
120         imageDrop.set_img_src("img/mm_drop_white.png")
121         imageDrop.set_pos(105, 8)
122         lcd.arc(centerX, centerY, radius, thick, 0, endposfull,0
    xff0000,0xff0000)
123         lcd.arc(centerX, centerY, radius, thick, 0, endpos,0x000000,0
    x000000)
124     else:
125         if sst == "CALIBRATION_REQUIRED":
126             # no circle, red drop
127             imageDrop.set_img_src("img/mm_drop_red.png")
128             imageDrop.set_pos(100, 0)
129         else:
130             # no circle, white drop
131             imageDrop.set_img_src("img/mm_drop_white.png")
132             imageDrop.set_pos(105, 8)
133         lcd.arc(centerX, centerY, radius, thick, 0, endposfull,0
    x000000,0x000000)
134
135 def create_image( sg):
136     """Create a new image with a string label drawn on it."""
137     color1 = 'white'
138     width  = 100
139     height = 100
140     # Generate an image and draw a pattern
```

```
141     image = Image.new('RGB', (width, height), color1)
142     dc = ImageDraw.Draw(image)
143     dc.text([10,10],sg)
144 
145     return image
```

Código E.5: screen.py (Pump algorithm).

E.4. pump

E.4.1. basal.py

```
1  from pump.events import efficacy_factor, events_adjust
2  from util.utils import DATA_PATH, INPUTS, INPUTS_PATH, compute_std_dev
       , filter_records, get_period_value, read_data_json, writeJson
3  import logging as log
4  from datetime import datetime, timedelta
5  
6  MAX_INTEGRAL_LIMIT    =  400
7  MIN_INTEGRAL_LIMIT    = -400
8  HIPO_WEIGHT           =  3
9  RAPID_DROP_THRESHOLD =  -5  # mg/dL/min, equivalent to 25 mg/dL over 5
       minutes
10 LIMIT_STD_DIFFS       = 50
11 ERROR_LIMIT           = -10
12 WEIGHTED_ERROR_LIMIT = ERROR_LIMIT * HIPO_WEIGHT
13 LIMIT_BASAL_CHANGE    = 0.15  # Max 15% change in basal rate
14 
15 def basalDelivery(time, timeOld, glucose, error_history, eIntegral):
16     """Compute basal insulin delivery using an adaptive PID controller
       ."""
17 
18     params = read_data_json("data/params.json")
19     config = read_data_json("data/config.json")
20     events = read_data_json("data/events.json")
21 
22     last_update = datetime.fromisoformat(params['last_update']['value'
       ])
23     adapt_freq  = timedelta(hours=float(params['freq']['value']))
24 
25     nom_targetBG = get_period_value(config, 'target', time.time())
26     nom_coFactor = get_period_value(config, 'sensitivity', time.time()
       )
27     nom_targetBG = int(nom_targetBG['value'])
28     nom_coFactor = float(nom_coFactor['value'])
29 
30     targetBG = events_adjust(time, events, 'target', nom_targetBG)
31 
32     if should_adapt_params(time, last_update, adapt_freq):
33         adapt_pid_params(time, params, adapt_freq, targetBG)
34 
35     kP = float(params['kP']['value'])
36     kI = float(params['kI']['value'])
```

```
37      kD = float(params['kD']['value'])
38      bR = float(params['bR']['value'])
39      basalMax = float(config['basal']['value'])
40
41      # Compute current error
42      actualError = float(glucose - targetBG)
43      error = actualError * HIPO_WEIGHT if actualError < 0 else
    actualError
44
45      if timeOld is None:
46          error_history.append(actualError)
47          return 0, 0, 0, 0
48
49      print("==============Basal delivery calculation================")
50      print(f"-> Time    = {time}")
51      print(f"-> TimeOld = {timeOld}")
52      print(f"-> Glucose = {glucose} mg/dL")
53      print(f"-> Error = {actualError:.4f} mg/dL")
54
55      # Update integral of error
56      dt = (time - timeOld).total_seconds() / 3600  # Convert to hours
57      eIntegral += error * dt
58      eIntegral = max(min(eIntegral, MAX_INTEGRAL_LIMIT),
    MIN_INTEGRAL_LIMIT)
59      if targetBG - 10.0 <= glucose <= targetBG + 30.0:
60          eIntegral = 0  # Reset integral if glucose is near target
61
62      # Derivative term
63      errorRate = calculate_error_rate(actualError, error_history, dt)
    # mg/dL/h
64
65      if error < WEIGHTED_ERROR_LIMIT:
66          print(f"-> Error {actualError:.4f} mg/dl < {ERROR_LIMIT} mg/dl
     (Error limit)")
67          print(f"-> Basal Rate = {0} U/h")
68          return 0, 0, eIntegral, errorRate
69
70      if errorRate/60 < RAPID_DROP_THRESHOLD:
71          print(f"-> Error rate {errorRate/60} mg/dl/min < {
    RAPID_DROP_THRESHOLD} mg/dl/min (Rapid drop threshold)")
72          print(f"-> Basal Rate = {0} U/h")
73          return 0, 0, eIntegral, errorRate
74
75      # PID control
76      basalRate = round(bR + kP * error + kI * eIntegral + kD *
    errorRate,4)
77      print(f"-> P: error = {round(error,2)} mg/dl,          rate = {
    round( kP * error,2)} U/h")
78      print(f"-> I: error = {round(eIntegral,2)} mg/dl * h,  rate = {
    round( kI * eIntegral,2)} U/h")
79      print(f"-> D: error = {round(errorRate,2)} mg/dl / h,  rate = {
    round( kD * errorRate,2)} U/h")
80      print(f"-> Basal Rate = {basalRate} U/h   (Nominal was {bR} U/h)")
81      # Apply efficacy factor
82      efficacy = efficacy_factor(time, config, events)
83      basalRate *= efficacy
84      basalRate = max(0, min(basalRate, basalMax))  # Clamp to max
85
```

```
86      print(f"-> Final Basal Rate = {basalRate} U/h")
87
88      if basalRate > 0:
89          save_basal(basalRate, time, dt)
90
91      basal = round(basalRate * dt,2)
92      print(f"-> Basal = {basal} U")
93
94      return basal, basalRate, eIntegral, errorRate
95
96  def save_basal(basalRate, time, dt):
97      """ Save basal in inputs file """
98      data = read_data_json(INPUTS_PATH)
99      if data is None:
100         data = {key: [] for key in INPUTS}
101
102     basal_entry = {
103         "start": time.strftime('%Y-%m-%dT%H:%M:%S'),
104         "duration": round(dt * 60, 1),  # in minutes
105         "iir": basalRate,               # in U/h
106         "type": "Basal"
107     }
108     data['basal'].append(basal_entry)
109     # Write updated data back to the file
110     writeJson(data, INPUTS_PATH)
111
112     print(f"Basal saved: {basal_entry}")
113
114 def calculate_error_rate(error, error_history, dt):
115     """ Retrieve error rate, according to the average rate of the last
        5 records"""
116     error_history.append(error)
117     if len(error_history) > 5:
118         error_history.pop(0)   # Last 5 errors
119     errorAvg = sum([(error_history[i] - error_history[i - 1]) for i in
        range(1, len(error_history))]) / (len(error_history) - 1)
120     errorRate = errorAvg / dt
121     return errorRate
122
123 def predict_glucose(current_glucose, rate_of_change, prediction_time
    =30):
124     """Predict future glucose value in mg/dL after prediction_time (in
        minutes).If rate_of_change is positive, glucose is increasing."""
125     predicted_glucose = current_glucose + rate_of_change *
    prediction_time / 60
126     return predicted_glucose
127
128 def should_adapt_params(time, last_update, adapt_freq):
129     """ Compute if PID parameters should be adapted from last update
    """
130     return time - last_update >= adapt_freq
131
132 def adapt_pid_params(current_time, params, freq, target):
133     """ Adapt PID parameters accounting for stability and time in
    range"""
134
135     print(f"!!!!!!Adapting PID parameters at {current_time}!!!!!!!")
136
```

```
137     glucose_records     =     read_data_json(DATA_PATH  )
138     basal_records       =     read_data_json(INPUTS_PATH).get("basal",
    [])
139     start_time = current_time - freq
140 
141     relevant_basal_records      = filter_records(basal_records,
    start_time, current_time, "start")
142     relevant_glucose_records    = filter_records(glucose_records,
    start_time, current_time, "time" )
143 
144     stability     = analyze_stability(    relevant_glucose_records)
145     time_in_range = analyze_time_in_range(relevant_glucose_records,
    target)
146     print(f"---> Stability = {round(stability,2)},  time_in_range = {
    time_in_range} %")
147     if time_in_range < 0:
148         return
149 
150     print(f"---> kP = {params['kP']['value']}")
151     print(f"---> kI = {params['kI']['value']}")
152     print(f"---> kD = {params['kD']['value']}")
153 
154     kP_new = adjust_kP(float(params['kP']['value']), stability,
    time_in_range)
155     kI_new = adjust_kI(float(params['kI']['value']), stability,
    time_in_range)
156     kD_new = adjust_kD(float(params['kD']['value']), stability,
    time_in_range)
157 
158     print(f"------> kP_new = {kP_new}")
159     print(f"------> kI_new = {kI_new}")
160     print(f"------> kD_new = {kD_new}")
161 
162     bR_new = calculate_basal_rate(relevant_basal_records, float(params
    ['bR']['value']), start_time, current_time )
163 
164     new_hist_entry = {
165         "start": current_time.strftime('%Y-%m-%dT%H:%M:%S'),
166         "kP": kP_new,
167         "kI": kI_new,
168         "kD": kD_new,
169         "bR": bR_new
170     }
171     params['hist']['period'].append(new_hist_entry)
172     print(f"------> bR_new = {bR_new}")
173 
174     params['kP']['value'] = str(kP_new)
175     params['kI']['value'] = str(kI_new)
176     params['kD']['value'] = str(kD_new)
177     params['bR']['value'] = str(bR_new)
178     params['last_update']['value'] = current_time.strftime('%Y-%m-%dT%
    H:%M:%S')
179 
180     writeJson(params, "data/params.json")
181     print(f"PID parameters adapted at {current_time}")
182 
183 def calculate_basal_rate(basal_records, bR, start_time, end_time):
184     """ Compute basal rate"""
```

```
185
186     if not basal_records:
187         log.warning("No basal records found.")
188         return bR
189
190     # Total basal insulin delivered (U)
191     total_basal = sum(
192         float(record['iir']) * (float(record['duration']) / 60) #
    duration in hours
193         for record in basal_records
194     )
195     time_covered = (end_time - start_time).total_seconds() / 3600  #
    in hours
196     average_basal_rate = round(total_basal / time_covered, 3)
197     print(f"Basal calculation: Total basal = {total_basal:.4f} U, Time
     covered = {time_covered} h, Average basal rate = {
    average_basal_rate} U/h")
198
199     # Limit basal rate change
200     max_change = LIMIT_BASAL_CHANGE * bR
201     lower_limit = bR - max_change
202     upper_limit = bR + max_change
203
204     # Constrain within allowed limits
205     average_basal_rate = max(lower_limit, min(average_basal_rate,
    upper_limit))
206     print(f"Limited average basal rate to {average_basal_rate} U/h
    (10% range of {bR})")
207
208     return average_basal_rate
209
210
211 def analyze_time_in_range(glucose_records, targetBG):
212     """ Analyze the historical performance to determine stability and
    time in range. """
213     range_lower = targetBG - 10
214     range_upper = targetBG + 10
215
216     in_range_count = 0
217     total_count = len(glucose_records)
218     for record in glucose_records:
219         glucose = record['glucose']
220         if range_lower <= glucose <= range_upper:
221             in_range_count += 1
222
223     time_in_range = (in_range_count / total_count) * 100 if
    total_count > 0 else -1
224
225     return time_in_range
226
227
228 def analyze_stability(glucose_records):
229     """Calculate stability by analyzing oscillations in glucose
    records."""
230     if len(glucose_records) < 2:
231         return 1  # Maximum stability if there are not enough data
232     differences = [
```

```
233         glucose_records[i]['glucose'] - glucose_records[i - 1]['
    glucose']
234         for i in range(1, len(glucose_records))
235     ]
236
237     std_dev = compute_std_dev(differences)
238     stability = max(0, 1 - (std_dev / LIMIT_STD_DIFFS))
239     return stability
240
241 def adjust_kP(kP, stability, time_in_range):
242     """
243     Adjust proportional gain (kP).
244
245     - Role:
246         kP determines the system's responsiveness to the magnitude of
    the error.
247         A higher kP results in more aggressive corrections, while a
    lower kP reduces the risk of overshooting.
248
249     - Adaptation logic:
250         1. Increase kP if:
251             - The system shows slow response to large errors (e.g.,
    prolonged hyperglycemia).
252             - There is a lack of correction in persistent
    hyperglycemia.
253         2. Decrease kP if:
254             - Large oscillations occur, indicating overcorrection (e.g
    ., repeated swings into hypoglycemia).
255             - Instability is observed, with glucose repeatedly
    crossing the target range.
256
257     - Adjustment:
258         - Increase or decrease kP by +/-10% based on system
    performance.
259     """
260     if stability > 0.8 and time_in_range < 70:
261         return round(kP * 1.1, 5)  # Increase kP by 10%
262     elif stability < 0.7 and time_in_range > 70:
263         return round(kP * 0.9, 5)  # Decrease kP by 10%
264     else:
265         return round(kP, 5)
266
267
268 def adjust_kI(kI, stability, time_in_range):
269     """
270     Adjust integral gain (kI).
271
272     - Role:
273         kI accumulates the error over time, addressing persistent
    small deviations (e.g., steady-state errors).
274         A higher kI ensures the system corrects long-term offsets,
    while a lower kI prevents excessive accumulation.
275
276     - Adaptation logic:
277         1. Increase kI if:
278             - Persistent residual errors are observed (e.g., glucose
    stabilizing far from the target).
```

```
279             - The current basal rate is insufficient to maintain
    glucose within range.
280         2. Decrease kI if:
281             - Excessive accumulation of the integral term leads to
    overcorrections (e.g., hypoglycemia following hyperglycemia).
282             - Oscillations around the target occur due to
    overcompensation of accumulated error.
283
284     - Adjustment:
285         - Modify kI by +/- 5-10% depending on stability and time in
    range.
286     """
287     if time_in_range < 70:
288         return round(kI * 1.05, 5)  # Increase kI by 5%
289     elif time_in_range > 90:
290         return round(kI * 0.95, 5)  # Decrease kI by 5%
291     else:
292         return round(kI, 5)
293
294
295 def adjust_kD(kD, stability, time_in_range):
296     """
297     Adjust derivative gain (kD).
298
299     - Role:
300         kD addresses the rate of change of the error, acting as a
    buffer to reduce overshooting and oscillations.
301         A higher kD provides better damping for rapid changes, while a
     lower kD reduces excessive braking.
302
303     - Adaptation logic:
304         1. Increase kD if:
305             - Rapid oscillations occur due to overly aggressive
    proportional responses.
306             - The system fails to buffer rapid changes in glucose
    levels effectively.
307         2. Decrease kD if:
308             - The system becomes overly damped, delaying stabilization
    .
309             - The derivative term does not contribute to significant
    changes due to slow glucose variations.
310
311     - Adjustment:
312         - Adjust kD by +/-5% based on system stability and response
    speed.
313     """
314     if stability < 0.7:
315         return round(kD * 1.1, 5)  # Increase kD by 10%
316     elif stability > 0.9:
317         return round(kD * 0.9, 5)  # Decrease kD by 10%
318     else:
319         return round(kD, 5)
```

Código E.6: basal.py (Pump algorithm).

E.4.2. correction.py

```
from datetime import timedelta
from pump.events import efficacy_factor, events_adjust, events_apply
from util.utils import INPUTS, INPUTS_PATH, get_period_value,
    read_data_json, writeJson

def adjustMeal(currentTime, actualBG, activeInsulin,
    apply_activeInsulin):
    """Calculate adjust for meal bolus delivering"""
    # Load parameters from config.json
    config   = read_data_json('data/config.json')
    events   = read_data_json("data/events.json")
    efficacy = efficacy_factor(currentTime, config, events)
    if apply_activeInsulin:
        effective_activeInsulin = activeInsulin / efficacy
    else:
        effective_activeInsulin = 0

    nom_targetBG  = get_period_value(config, 'target',
    currentTime.time())
    nom_coFactor  = get_period_value(config, 'sensitivity',
    currentTime.time())

    if nom_targetBG is None:
        return
    if nom_coFactor is None:
        return

    nom_targetBG  = int( nom_targetBG[  'value'])
    nom_coFactor  = float(nom_coFactor ['value'])

    # Calculate BG error and correction bolus if needed
    targetBG  = events_adjust(currentTime, events, 'target',
    nom_targetBG )
    coFactor  = events_adjust(currentTime, events, 'sensitivity',
    nom_coFactor )

    # Glucose error
    BGerror = actualBG - targetBG
    # Correction bolus is function of the glucose error and the
    sensitivity factor. Active insulin is subtracted from the
    correction bolus.
    correctionBolus = BGerror / coFactor - effective_activeInsulin

    # Apply efficacy factor
    effective_correctionBolus = round(correctionBolus * efficacy, 1)

    return effective_correctionBolus

def correctionBolus(currentTime, actualBG, activeInsulin, basal = 0,
    verbose = True):
    """Calculate and save a correction bolus based on current blood
    glucose (BG) levels."""
    # Load parameters from config.json
    config   = read_data_json('data/config.json')
    events   = read_data_json("data/events.json")
    efficacy = efficacy_factor(currentTime, config, events)
```

```
47      effective_activeInsulin = activeInsulin / efficacy
48      effective_basalInsulin  = basal / efficacy
49
50      nom_targetBG  = get_period_value(config, 'target',
        currentTime.time())
51      nom_coFactor  = get_period_value(config, 'sensitivity',
        currentTime.time())
52
53      if nom_targetBG is None:
54          return
55      if nom_coFactor is None:
56          return
57
58      nom_targetBG  = int( nom_targetBG[  'value'])
59      nom_coFactor  = float(nom_coFactor ['value'])
60
61      # Calculate BG error and correction bolus if needed
62      targetBG  = events_adjust(currentTime, events, 'target',
        nom_targetBG )
63      coFactor  = events_adjust(currentTime, events, 'sensitivity',
        nom_coFactor )
64
65      # Glucose error
66      BGerror = actualBG - targetBG
67      # Correction bolus is function of the glucose error and the
        sensitivity factor. Active insulin is subtracted from the
        correction bolus.
68      correctionBolus = BGerror / coFactor - (effective_activeInsulin +
        effective_basalInsulin)
69      # Apply efficacy factor
70      effective_correctionBolus = round(correctionBolus * efficacy, 1)
71
72      # Check if correction is allowed
73      apply_correction = events_apply(currentTime, events)
74
75      # Log the correction bolus details
76      if verbose:
77          print("***************Correction bolus calculation
        *************:")
78          print(f"-> Current BG = {actualBG} mg/dL")
79          print(f"-> Nominal Target BG = {nom_targetBG} mg/dL")
80          print(f"->  Target BG = {targetBG} mg/dL")
81          print(f"-> Active Insulin = {activeInsulin:.4f} U")
82          print(f"-> Effective Active Insulin = {effective_activeInsulin
        :.4f} U")
83          print(f"-> Basal Insulin = {basal} U")
84          print(f"-> Effective Basal Insulin = {effective_basalInsulin
        :.4f} U")
85          print(f"-> Correction bolus = {BGerror} mg/dL / ({coFactor} mg
        /dL/U) - ({activeInsulin:.4f} U) - ({effective_basalInsulin} U) =
        {correctionBolus:.4f} U")
86          print(f"-> Apply correction: {apply_correction}")
87          print(f"-> Effective correction bolus: {
        effective_correctionBolus:.4f}")
88
89      # Save the correction bolus to a JSON file
90      if apply_correction and effective_correctionBolus >= 0.1:
91          save_correction_bolus(currentTime, effective_correctionBolus)
```

```
92
93
94  def mealBolus(ch_value, ugp_value, currentTime, actualBG,
        activeInsulin, apply_activeInsulin):
95      """ Compute the insulin bolus due to a meal intake"""
96
97      config       = read_data_json('data/config.json')
98      events       = read_data_json("data/events.json")
99      efficacy     = efficacy_factor(currentTime, config, events)
100     adjustment   = adjustMeal(currentTime, actualBG, activeInsulin,
        apply_activeInsulin)
101     ratio_period = get_period_value(config, 'ratio', currentTime.time
        ())
102
103     result = round( ch_value / float(ratio_period['value']) * efficacy
        , 1 )
104
105     if abs(adjustment) > result:
106         adjustment = - result
107
108     bolus  = round( result + adjustment, 1 )
109
110     # Calculate delayed bolus for UGPs
111     ugps = []
112     ugp_duration = 3  # Initial duration for 1 UGP (in hours)
113
114     for ugp_num in range(1, int(ugp_value) + 1):
115         ugp_bolus = round(result / ugp_value, 1)  # Divide
        carbohydrate bolus among UGPs
116         ugp_time = currentTime + timedelta(hours=ugp_duration)
117
118         ugps.append({
119             "ugp_bolus": ugp_bolus,
120             "ugp_time": ugp_time,
121             "ugp_num": ugp_num
122         })
123
124         # Increment duration by 1 hour for each additional UGP
125         ugp_duration += 1
126
127     return result, adjustment, bolus, ugps
128
129 def save_correction_bolus(currentTime, correctionBolus):
130     """Save the correction bolus to the bolus section of the inputs
        JSON file."""
131     data = read_data_json(INPUTS_PATH)
132     if data is None:
133         data = {key: [] for key in INPUTS}
134
135     # Append the new correction bolus
136     bolus_entry = {
137         "start": currentTime.strftime('%Y-%m-%dT%H:%M:%S'),
138         "i": correctionBolus,
139         "type": "Correction"
140     }
141     data['bolus'].append(bolus_entry)
142
143     # Write updated data back to the file
```

```
144     writeJson(data, INPUTS_PATH)
145 
146     print(f"Correction bolus saved: {bolus_entry}")
```

Código E.7: correction.py (Pump algorithm).

E.4.3. data.py

```
1 
2 
3 from datetime import datetime, timedelta
4 import logging as log
5 
6 from util.utils import DATA_PATH, INPUTS_PATH, read_data_json
7 
8 ERROR_AI = 0.04  # mg/dl Error tolerance for active insulin
    calculations
9 
10 def handle_pumpdataupdate():
11     """ Handles the update of pump data by reading the latest entry
    from the JSON file."""
12 
13     data    = read_data_json(DATA_PATH)
14     try:
15         latest_entry = max(data, key=lambda x: x['time'])
16         lastUpdateTm = datetime.strptime(latest_entry['time'], "%Y-%m-%
    dT%H:%M:%S")
17         lastSG = latest_entry['glucose']
18         activeInsulin = float(latest_entry['activeInsulin'])
19 
20     except KeyError as e:
21         print(f"ERROR: Missing key in data - {e}")
22         return None, None, None
23     except Exception as e:
24         print(f"ERROR: Problem processing data - {e}")
25         return None, None, None
26     return lastSG, lastUpdateTm, activeInsulin
27 
28 
29 def applyAI_from_inputs(date):
30     """ Whether to apply active insulin from inputs.
31     Do not apply if the date is within 1 hour of a meal, so that the
    active insulin is not considered.
32     """
33 
34     inputs = read_data_json(INPUTS_PATH)
35 
36     for meal in inputs['meals']:
37         start = datetime.strptime(meal['start'], "%Y-%m-%dT%H:%M:%S")
38         duration = timedelta(minutes=meal['duration'])
39 
40         time_end = start + duration + timedelta(hours=1)
41 
42         if start <= date <= time_end:
43             return False
```

```
44
45      return True
46
47
48  def last_correcion_in_data(dateOld, activeInsulin, lastActiveInsulin):
49      """
50      Checks whether the active insulin from the last correction should
    be considered in the data.
51
52      Args:
53          dateOld (datetime): Date and time of the last correction.
54          activeInsulin (float): Current active insulin.
55          lastActiveInsulin (float): Previous active insulin.
56
57      Returns:
58          bool: True if the condition is met to consider the active
    insulin from the last correction.
59      """
60      if dateOld is None or activeInsulin is None or lastActiveInsulin
    is None:
61          return True
62
63      # Read the data from JSON files
64      data = read_data_json(DATA_PATH)
65      inputs = read_data_json(INPUTS_PATH)
66      foundCorrection = False
67
68      # Look for the correction bolus entry matching datetime dateOld
69      for entry in inputs["bolus"]:
70          if entry["type"] == "Correction" and datetime.fromisoformat(
    entry["start"]) == dateOld.replace(microsecond=0):
71              bolus_i = float(entry["i"])  # Correction insulin found
72              log.debug(f"Correction bolus found at {dateOld} with
    insulin {bolus_i}")
73              foundCorrection = True
74
75              # Look for the closest active insulin entry before or at
    dateOld
76              closest_entry = None
77              closest_time = None
78
79              for data_entry in data:
80                  entry_time = datetime.fromisoformat(data_entry["time"
    ])
81
82                  # Check if entry_time is before or equal to dateOld
83                  if entry_time <= dateOld:
84                      if closest_time is None or entry_time >
    closest_time:
85                          closest_entry = data_entry
86                          closest_time = entry_time
87                  else:
88                      log.debug(f"Ignoring data entry with time {
    entry_time.isoformat()} greater than {dateOld.isoformat()}")
89
90              if closest_entry:
91                  last_active_i_from_data = float(closest_entry['
    activeInsulin'])
```

```
92                  log.debug(f"Active insulin found in data for {
    closest_time.isoformat()}: {last_active_i_from_data}, dateOld = {
    dateOld}")
93
94                  # Check the conditions
95                  if abs(activeInsulin - lastActiveInsulin - bolus_i) <=
     ERROR_AI and abs(lastActiveInsulin - last_active_i_from_data) <=
    ERROR_AI:
96                      log.debug(f"Conditions met: activeInsulin -
    lastActiveInsulin = {bolus_i}, lastActiveInsulin = {
    last_active_i_from_data}, dateOld = {dateOld}")
97                      return True
98                  else:
99                      log.debug(f"Conditions not met: activeInsulin -
    lastActiveInsulin = {activeInsulin - lastActiveInsulin}, expected
    {bolus_i}")
100                     log.debug(f"lastActiveInsulin = {lastActiveInsulin
    }, expected {last_active_i_from_data}, dateOld = {dateOld}")
101                     return False
102             else:
103                 log.debug(f"No active insulin entry found in data
    before or at {dateOld}")
104             break
105
106     if not foundCorrection:
107         log.debug(f"No correction bolus found at {dateOld}")
108
109     return True
```

Código E.8: data.py (Pump algorithm).

E.4.4. events.py

```
1
2
3  from datetime import datetime, timedelta
4  import logging as log
5
6  ADD_FACTOR_EFF = 0.3
7
8  def efficacy_factor(current_time, config, events):
9      """ Calculate the efficacy factor based on the current time and
    reservoir last change time."""
10
11     reservoir_effect_duration = float(config['reservoir']['value'])
12
13     reservoir_events = events.get("reservoir", {})
14     start_time = datetime.fromisoformat(reservoir_events.get("start"))
15     periodic = reservoir_events.get("periodic", "No") == "Yes"
16     days_between_changes = int(reservoir_events.get("days", 0)) if
    periodic else None
17
18     # Periodic changes of reservoir
19     if periodic:
```

```
20          time_since_start = (current_time - start_time).total_seconds()
    / 3600 / 24  # days
21          cycles_since_start = int(time_since_start //
    days_between_changes)
22          last_change_time = start_time + timedelta(days=
    cycles_since_start * days_between_changes)
23      else:
24          last_change_time = start_time
25
26      days_since_change = (current_time - last_change_time).
    total_seconds() / 3600 / 24
27
28      log.debug(f"Reservoir effect duration: {reservoir_effect_duration}
    ")
29      log.debug(f"Days between changes: {days_between_changes}")
30      log.debug(f"Days since change: {days_since_change}")
31      log.debug(f"Last change time: {last_change_time}")
32      log.debug(f"Current time: {current_time}")
33      log.debug(f"Start time: {start_time}")
34
35      # Full effect of reservoir
36      if days_since_change < reservoir_effect_duration:
37          multiplication_factor = 1.0
38      # Partial effect of reservoir
39      elif reservoir_effect_duration <= days_since_change <=
    days_between_changes:
40          multiplication_factor = 1.0 + ADD_FACTOR_EFF * (
    days_since_change - reservoir_effect_duration) / (
    days_between_changes - reservoir_effect_duration)
41      else:
42          multiplication_factor = 1.0
43
44      return multiplication_factor
45
46
47  def events_adjust(time, events, name, value):
48      """ Adjust the target or sensitivity based on events."""
49
50      if name not in ['target', 'sensitivity']:
51          return value  # If not 'target' or 'sensitivity', return the
    original value
52
53      exercise_periods = events.get("exercise", {}).get("period", [])
54      alcohol_periods  = events.get("alcohol",  {}).get("period", [])
55      illness_periods  = events.get("illness",  {}).get("period", [])
56      mood_periods     = events.get("mood",     {}).get("period", [])
57
58      for period in exercise_periods:
59          try:
60              start_time = datetime.fromisoformat(period['start'])
61              duration = int(period['duration'])  # Duration in minutes
62              intensity = int(period['intensity'])  # Intensity level
63              end_time = start_time + timedelta(minutes=duration)
64
65              if name == 'target':
66                  # Check conditions for adjusting 'target'
67                  if intensity < 5 and duration < 60:
```

```
68                      if start_time - timedelta(hours=2) <= time <
    end_time + timedelta(minutes=30):
69                          value += 50
70                  elif intensity < 5 and duration >= 60:
71                      if start_time - timedelta(hours=3) <= time <
    end_time + timedelta(minutes=30):
72                          value += 50
73                  elif intensity >= 5 and duration >= 60:
74                      if start_time - timedelta(hours=2) <= time <
    end_time + timedelta(minutes=30):
75                          value += 50
76                  # If intensity >= 5 and duration < 60, 'target' does
    not change
77              elif name == 'sensitivity':
78                  max_sensitivity = value * (1 + intensity * 0.1)  #
    Maximum sensitivity based on intensity
79                  duration_sensitivity = duration +  30*intensity
80                  if duration < 30:
81                      # Triangular shape: increases linearly during
    exercise, decreases after
82                      if start_time <= time < end_time:
83                          # Ascending phase: increases linearly until
    exercise ends
84                          time_elapsed = (time - start_time).
    total_seconds() / 60  # Minutes passed
85                          value = value + (max_sensitivity - value) * (
    time_elapsed / duration_sensitivity)
86                      elif end_time <= time < end_time + timedelta(
    minutes=duration_sensitivity):
87                          # Descending phase: decreases linearly back to
     original value
88                          time_since_end = (time - end_time).
    total_seconds() / 60  # Minutes after exercise
89                          value = max_sensitivity - (max_sensitivity -
    value) * (time_since_end / duration_sensitivity)
90                  else:
91                      # Trapezoidal shape: increases, stays constant,
    then decreases
92                      if start_time <= time < start_time + timedelta(
    minutes=30):
93                          # Ascending phase: increases linearly to max
    in 30 minutes
94                          time_elapsed = (time - start_time).
    total_seconds() / 60  # Minutes passed
95                          value = value + (max_sensitivity - value) * (
    time_elapsed / 30)
96                      elif start_time + timedelta(minutes=30) <= time <
    end_time:
97                          # Constant phase: stays at maximum during
    exercise
98                          value = max_sensitivity
99                      elif end_time <= time < end_time + timedelta(
    minutes=duration_sensitivity):
100                         # Descending phase: decreases linearly back to
     original value
101                         time_since_end = (time - end_time).
    total_seconds() / 60  # Minutes after exercise
```

```
                        value = max_sensitivity - (max_sensitivity -
value) * (time_since_end / duration_sensitivity)
            except (ValueError, KeyError):
                # Ignore periods with missing or invalid data
                continue

        for period in alcohol_periods:
            try:
                start_time = datetime.fromisoformat(period['start'])
                duration = int(period['duration'])  # Duration in minutes
                amount = float(period['amount'])  # Intensity level
                end_time = start_time + timedelta(minutes=duration)

                if name == 'target':
                    # Check conditions for adjusting 'target'
                    if amount < 5:
                        if start_time <= time < end_time + timedelta(
minutes=30):
                            value += amount*10
                    elif amount >= 5:
                        if start_time - timedelta(minutes=30) <= time <
end_time + timedelta(minutes=30*amount):
                            value += 50
                elif name == 'sensitivity':
                    max_sensitivity = value * (1 + amount * 0.1)  #
Maximum sensitivity based on intensity
                    if 2 < amount < 5:
                        # Triangular shape: increases linearly during
exercise, decreases after
                        if start_time <= time < end_time:
                            # Ascending phase: increases linearly until
exercise ends
                            time_elapsed = (time - start_time).
total_seconds() / 60  # Minutes passed
                            value = value + (max_sensitivity - value) * (
time_elapsed / duration)
                        elif end_time <= time < end_time + timedelta(
minutes=30):
                            # Descending phase: decreases linearly back to
 original value
                            time_since_end = (time - end_time).
total_seconds() / 60
                            value = max_sensitivity - (max_sensitivity -
value) * (time_since_end / duration)
                    elif amount >= 5:
                        # Trapezoidal shape: increases, stays constant,
then decreases
                        if start_time <= time < start_time + timedelta(
minutes=30):
                            # Ascending phase: increases linearly to max
in 30 minutes
                            time_elapsed = (time - start_time).
total_seconds() / 60  # Minutes passed
                            value = value + (max_sensitivity - value) * (
time_elapsed / 30)
                        elif start_time + timedelta(minutes=30) <= time <
end_time + timedelta(minutes=30*amount/2):
```

```
                        # Constant phase: stays at maximum during exercise
                        value = max_sensitivity
                    elif end_time + timedelta(minutes=30*amount/2) <= time < end_time + timedelta(minutes=30):
                        # Descending phase: decreases linearly back to original value
                        time_since_end = (time - end_time).total_seconds() / 60  # Minutes after exercise
                        value = max_sensitivity - (max_sensitivity - value) * (time_since_end / 30)

            except (ValueError, KeyError):
                # Ignore periods with missing or invalid data
                continue

        for period in illness_periods:
            try:
                start_time = datetime.fromisoformat(period['start'])
                duration = int(period['duration'])  # Duration in minutes
                type = period['type']  # Illness type
                end_time = start_time + timedelta(minutes=duration)

                if name == 'target':
                    pass  # No adjustment for 'target' during illness
                elif name == 'sensitivity':
                    if type == "Dehydration":
                        if start_time <= time < end_time:
                            value = value * 0.9
                    elif type in ['Infection','Fever']:
                        min_sensitivity = value * 0.8  # Minimum sensitivity during illness
                        # Trapezoidal shape: increases, stays constant, then decreases
                        if start_time <= time < start_time + timedelta(minutes=30):
                            # Ascending phase: increases linearly to max in 30 minutes
                            time_elapsed = (time - start_time).total_seconds() / 60  # Minutes passed
                            value = value + (min_sensitivity - value) * (time_elapsed / 30)
                        elif start_time + timedelta(minutes=30) <= time < end_time :
                            # Constant phase: stays at maximum during exercise
                            value = min_sensitivity
                        elif end_time <= time < end_time + timedelta(minutes=30):
                            # Descending phase: decreases linearly back to original value
                            time_since_end = (time - end_time).total_seconds() / 60  # Minutes after exercise
                            value = min_sensitivity - (min_sensitivity - value) * (time_since_end / 30)
            except (ValueError, KeyError):
                # Ignore periods with missing or invalid data
                continue
```

```
182
183     for period in mood_periods:
184         try:
185             start_time = datetime.fromisoformat(period['start'])
186             duration = int(period['duration'])  # Duration in minutes
187             type = period['type']  # Illness type
188             end_time = start_time + timedelta(minutes=duration)
189
190             if name == 'target':
191                 if type == "Stress":
192                     if start_time <= time < end_time:
193                         value = value + 20   # Increase target by 20
    mg/dL in case there are corrections, so that they are less
    aggressive
194             elif name == 'sensitivity':
195                 if type == "Stress":
196                     if start_time <= time < start_time + timedelta(
    minutes=30) < end_time and duration > 120:
197                         value = value * 0.9
198                 elif type == "Relax":
199                     if start_time <= time < end_time:
200                         value = value * 1.1
201         except (ValueError, KeyError):
202             # Ignore periods with missing or invalid data
203             continue
204
205     return float(value)
206
207
208 def events_apply(time, events):
209     """ Check if insulin corrections are allowed based on events."""
210
211     exercise_periods = events.get("exercise", {}).get("period", [])
212     alcohol_periods  = events.get("alcohol",  {}).get("period", [])
213
214     for period in exercise_periods:
215         try:
216             start_time = datetime.fromisoformat(period['start'])
217             duration = int(period['duration'])  # Duration in minutes
218             intensity = int(period['intensity'])  # Intensity level
219             end_time = start_time + timedelta(minutes=duration)
220
221             if duration >= 60 and intensity < 5:
222                 # False from 3 hours before start to 2 hours after end
223                 if start_time - timedelta(hours=3) <= time <= end_time
    + timedelta(hours=2):
224
225                     return False
226             elif duration >= 60 and intensity >= 5:
227                 # False from start to 2 hours after end
228                 if start_time <= time <= end_time + timedelta(hours=2)
    :
229                     return False
230         except (ValueError, KeyError):
231             # Ignore periods with invalid or missing data
232             continue
233
234     for period in alcohol_periods:
```

```
235         try:
236             start_time = datetime.fromisoformat(period['start'])
237             duration = int(period['duration'])  # Duration in minutes
238             amount   = float(period['amount'])  # Intensity level
239             end_time = start_time + timedelta(minutes=duration)
240 
241             if amount > 5:
242                 # False from 3 hours before start to 2 hours after end
243                 if start_time <= time <= end_time + timedelta(minutes
    =30*amount):
244 
245                     return False
246         except (ValueError, KeyError):
247             # Ignore periods with invalid or missing data
248             continue
249 
250     # True for all other cases
251     return True
```

Código E.9: events.py (Pump algorithm).

E.5. util

E.5.1. plot.py

```
1  import matplotlib.pyplot as plt
2  from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
3  from util.utils import DATA_PATH, read_data_json
4  from datetime import datetime, timedelta
5  import logging as log
6  import matplotlib.dates as mdates
7 
8  def setup_graph(screen, end_time, screen_size):
9      """Set up the glucose graph on the given screen."""
10     data = read_data_json(DATA_PATH)
11 
12     if not data:
13         log.error("No data found in the JSON file.")
14         return
15 
16     # Define the time range
17     start_time = end_time - timedelta(hours=3)
18 
19     # Filter data within the time range
20     filtered_data = [
21         entry for entry in data
22         if start_time <= datetime.fromisoformat(entry['time']) <=
    end_time
23     ]
24 
25     # If not enough data in range, take the most recent available
26     if not filtered_data:
27         filtered_data = [
```

```
            entry for entry in data
            if datetime.fromisoformat(entry['time']) <= end_time
        ][-1 * len(data):]  # Take up to the last 3 hours of available data

    if not filtered_data:
        log.warning("No data available for the specified time range.")
        return

    # Extract time and glucose values for the graph
    times = [datetime.fromisoformat(entry['time']) for entry in filtered_data]
    glucose_values = [entry['glucose'] for entry in filtered_data]

    # Destroy previous graph if it exists
    for widget in screen.winfo_children():
        widget.destroy()

    # Create Matplotlib figure
    fig, ax1 = plt.subplots(figsize=(screen_size / 90, screen_size / 110), dpi=100)

    # Add data to the graph
    ax1.plot(times, glucose_values, label="Glucose", color="blue")
    ax1.xaxis.set_major_formatter(mdates.DateFormatter("%H:%M"))

    # Highlight normal glucose range
    ax1.axhspan(70, 180, color='lightgreen', alpha=0.3, label="Normal Range")

    # Configure axis limits and style
    ax1.set_ylim(50, 250)
    ax1.tick_params(axis='x', labelsize=6)
    ax1.tick_params(axis='y', labelsize=6)

    # Labels and title
    ax1.set_title("Glucose Levels", fontsize=8)
    ax1.set_xlabel("Time", fontsize=6)
    ax1.set_ylabel("Glucose (mg/dL)", fontsize=6)
    ax1.grid(True)

    # Embed graph in Tkinter
    canvas = FigureCanvasTkAgg(fig, master=screen)
    canvas_widget = canvas.get_tk_widget()
    canvas_widget.pack(fill="both", expand=True)
    canvas.draw()
    return canvas
```

Código E.10: plot.py (Pump algorithm).

E.5.2. time.py

```

from datetime import datetime, timedelta
import logging as log
```

```
4  import time
5
6  from util.utils import DATA_PATH, align_text, read_data_json,
       writeJson
7
8  # Timer 1: 10 sec (periodic) // timeupdate
9  TIMER1_PERIOD_S = 10
10 # Timer 2: 300 sec (periodic) // pumpdataupdate
11 TIMER2_PERIOD_S = 300
12
13 S_2_MS = 1000 # seconds to miliseconds
14
15 DELTA_SYNC = 65 # seconds
16
17 class CurrentTime:
18     """ Class to manage the current time. """
19     def __init__(self):
20         """ Initializes the current time to now. """
21         self.time = datetime.now()
22     def getTime(self):
23         """ Returns the current time. """
24         return self.time
25     def getTimeText(self):
26         """ Returns the current time as a string. """
27         return self.getTime().strftime('%Y-%m-%d %H:%M:%S')
28     def increment(self, delta):
29         """ Increments the current time by a given delta in seconds.
       """
30         self.time += delta
31         return self.time
32     def set_time(self, new_time):
33         """ Sets the current time to a new value."""
34         if isinstance(new_time, datetime):
35             self.time = new_time
36
37 class TimerSch:
38     """ Class to manage timers. """
39     def __init__(self, date):
40         """ Initializes the timer scheduler with a given date. """
41         self.date = date  # Reference to the simulated date
42         self.callbacks = []  # List of registered timers
43
44     def update_date(self, date):
45         """ Updates the date of the timer scheduler. """
46         self.date = date
47
48     def run(self, func, flag_name, instance, period, oneshot):
49         """
50         Schedules the execution of a periodic or one-time function.
51
52         Args:
53             func (callable): Function to execute.
54             flag_name (str): Name of the associated flag.
55             instance (object): Instance where the flag will be applied
       .
56             period (int): Period in simulated milliseconds.
57             oneshot (bool): If True, executes the function only once.
58         """
```

```
59          period_seconds = period / 1000  # Convert period to seconds
60          if oneshot:
61              # For one-time events, execute once when the time is
    reached
62              self.callbacks.append({
63                  "func": func,
64                  "flag_name": flag_name,
65                  "instance": instance,
66                  "period": period_seconds,
67                  "last_exec": None,  # Execute only once
68                  "oneshot": True
69              })
70          else:
71              # For periodic events, register with initial time
72              self.callbacks.append({
73                  "func": func,
74                  "flag_name": flag_name,
75                  "instance": instance,
76                  "period": period_seconds,
77                  "last_exec": self.date,
78                  "oneshot": False
79              })
80
81      def cancel_all_timers(self):
82          """ Cancels all registered timers. """
83          self.callbacks.clear()
84
85      def check_timers(self):
86          """
87          Checks all timers and executes functions as needed.
88          """
89          for callback in self.callbacks[:]:
90              elapsed = 0
91              if callback['last_exec']:
92                  elapsed = (self.date - callback['last_exec']).
    total_seconds()
93              if callback['oneshot'] and not callback['last_exec']:
94                  # Execute one-time events
95                  callback['func'](callback['flag_name'], callback['
    instance'])
96                  callback['last_exec'] = self.date
97                  self.callbacks.remove(callback)  # Remove after
    execution
98              elif not callback['oneshot'] and elapsed >= callback['
    period']:
99                  # Execute periodic events
100                 callback['func'](callback['flag_name'], callback['
    instance'])
101                 callback['last_exec'] = self.date
102
103
104 def wait_ms(time_ms):
105     """ Wait for a given time in milliseconds. """
106     time.sleep(time_ms/1000)
107
108
109 def time_delta(tm, date):
110     """ Calculate the time difference between two datetime objects.
```

```
111     It is the time since the last data update.
112     """
113     if tm is not None:
114         delta = date - tm
115         delta_minutes = delta.total_seconds() // 60
116 
117         if delta_minutes == 0:
118             delta_txt = "Now"
119         elif delta_minutes > 15 or delta.total_seconds() >= 3600:
120             delta_txt = "No data"
121         else:
122             delta_txt = f"{int(delta_minutes)} min ago"
123     else:
124         delta_txt = "---"
125     return delta_txt
126 
127 def handle_timeupdate(date, labelTime, labelLastData, lastUpdateTm):
128     """ Update the time and last data update label. """
129 
130     try:
131         # Update the current time
132         display_time = date.strftime("%H:%M")
133         labelTime.set_text(display_time)
134         align_text(labelTime, "right", 0)
135 
136         # Update time from the last update
137         labelLastData.set_text(time_delta(lastUpdateTm, date))
138         align_text(labelLastData, "center", 218)
139     except Exception as e:
140         log.error(f"Error in handle_timeupdate: {e}")
141 
142 def synchronize(timer, speed =1):
143     """ Synchronize the time with the data. """
144     print("Synchronizing time with data")
145     if speed == 1:
146         mult = 1
147     else:
148         mult = speed ** 0.5
149     data = read_data_json(DATA_PATH)
150     latest_entry = max(data, key=lambda x: x['time'])
151     lastUpdateTm = datetime.strptime(latest_entry['time'], "%Y-%m-%dT%
    H:%M:%S")
152     lastUpdateTm_plus = lastUpdateTm + timedelta(seconds=DELTA_SYNC*
    mult)
153 
154     for callback in timer.callbacks[:]:
155         callback['func'](callback['flag_name'], callback['instance'])
156         callback['last_exec'] = lastUpdateTm_plus
157 
158     params = read_data_json("data/params.json")
159     params['last_update']['value'] = lastUpdateTm_plus.strftime('%Y-%m
    -%dT%H:%M:%S')
160     writeJson(params, "data/params.json")
161 
162     return lastUpdateTm_plus
```

Código E.11: time.py (Pump algorithm).

E.5.3. utils.py

```
1
2  from datetime import datetime, timedelta
3  import json
4  import logging as log
5
6  FONT_MONT_14 = 'Helvetica 10'
7  FONT_MONT_16 = 'Helvetica 12'
8  FONT_MONT_20 = 'Helvetica 12'
9  FONT_MONT_26 = 'Helvetica 18'
10 FONT_MONT_28 = 'Helvetica 20'
11 FONT_MONT_48 = 'Helvetica 44'
12
13 # Window geometry
14 WINWIDTH  = 640
15 WINHEIGHT = 340
16 WINBORDER = 30
17
18 # DATA_PATH   = "C:/Users/palom/Desktop/p_data/data.json"
19 # INPUTS_PATH = "C:/Users/palom/Desktop/p_data/inputs.json"
20 DATA_PATH   = "C:/Users/palom/Desktop/codigo/data/data/data.json"
21 INPUTS_PATH = "C:/Users/palom/Desktop/codigo/data/inputs/inputs.json"
22
23 INPUTS = ['meals','bolus','basal']
24 KEY_MAP = {
25     "Meals": ['start','duration','carbs','UGPs'],
26     "Bolus": ['start','i'],
27     "Basal": ['start','duration','iir'],
28 }
29
30 def set_flag_true(flag_name, instance):
31    """Set a flag to True in the given instance."""
32    setattr(instance, flag_name, True)
33
34 def writeJson(jsonobj, filename):
35    """Write a JSON object to a file."""
36    try:
37       f = open(filename, "w")
38       f.write(json.dumps(jsonobj,indent=3))
39       f.close()
40    except Exception as e:
41       print("ERROR: failed to save %s (%s) " % (filename, str(e)))
42       return False
43    else:
44       return True
45
46 def align_text(label,pos,y):
47     """Align the text of a label to the specified position."""
48     if pos=="left":
49         label.set_pos(x=0,y=y)
50     elif pos=="center":
51         label.set_pos(x=160-int(label.get_width()/2),y=y)
52     elif pos=="right":
53         label.set_pos(x=320-label.get_width(),y=y)
54
55 def read_data_json(path):
56     """Read a JSON file and return its content."""
```

```
57      try:
58          # Leer el archivo JSON
59          with open(path, "r") as f:
60              data = json.load(f)
61
62      except FileNotFoundError:
63          print("ERROR: The file was not found in /data.json")
64          return None
65      except Exception as e:
66          print(f"ERROR: Problem reading the JSON file - {e}")
67          return None
68      return data
69
70
71
72  def get_period_value(config, param, currentTime):
73      """Find the applicable parameters based on the current time."""
74      for period in config[param]['period']:
75          # Handle periods ending at '24:00'
76          if period['to'] == '24:00':
77              period_to = (datetime.strptime('00:00', '%H:%M') -
    timedelta(microseconds=1)).time()
78          else:
79              period_to = datetime.strptime(period['to'],   '%H:%M').
    time()
80          period_from   = datetime.strptime(period['from'], '%H:%M').
    time()
81
82          if period_from <= currentTime <= period_to:
83              return period
84
85      log.warning(f"No {config[param]['name']} period found for the
    current time.")
86      return None
87
88  def filter_records(records, start_time, end_time, label):
89      """Filter records based on the given time range."""
90      filtered_records = []
91
92      for r in records:
93          record_start = datetime.fromisoformat(r[label])
94          if 'duration' in r:
95              record_end = record_start + timedelta(minutes=r['duration'
    ])
96
97              if record_start < end_time and record_end > start_time:
98                  new_start = max(record_start, start_time)
99                  new_end   = min(record_end,   end_time)
100
101                 new_record = r.copy()
102                 new_record[label] = new_start.isoformat()
103                 new_record['duration'] = (new_end - new_start).
    total_seconds() / 60
104
105                 filtered_records.append(new_record)
106         else:
107
108             if start_time <= record_start <= end_time:
```

```
                filtered_records.append(r)

    return filtered_records


def compute_mean(vars):
    """Compute the mean of a list of variables."""
    return sum(vars) / len(vars)

def compute_std_dev(vars):
    """Compute the standard deviation of a list of variables."""
    mean = compute_mean(vars)
    return   (sum((v - mean) ** 2 for v in vars) / len(vars)) ** 0.5
```

Código E.12: utils.py (Pump algorithm).