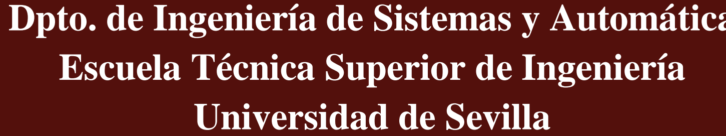
Trabajo de Fin de Máster

Máster en Automática, Robótica y Telemática

Integración de librerías de optimización en LabVIEW para aplicaciones en control predictivo.

Autor: Pablo Polo Garcimartín

Tutor: Fernando Dorado Navas



Dpto. de Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2019

Trabajo de Fin de Máster
Máster en Automática, Robótica y Telemática

Integración de librerías de optimización en LabVIEW para aplicaciones en control predictivo.

Autor:
Pablo Polo Garcimartín

Tutor:
Fernando Dorado Navas

Dpto. de Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2019

Proyecto Fin de Máster: Integración de librerías de optimización en LabVIEW para aplicaciones en control predictivo.

Autor: Pablo Polo Garcimartín

Tutor: Fernando Dorado Navas

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2019

El Secretario del Tribunal

Agradecimientos

En primer lugar agradezco a mis padres que han permitido que pueda llegar hasta aquí, sin su apoyo y esfuerzo nada habría sido siquiera posible, y aunque ha habido momentos duros que afrontar en todo este trayecto, en ningún momento han dejado de estar ahí. A mi tutor ayudándome cuando he encontrado dificultades y, aunque realizar este tipo de trabajos a distancia no suele ser la mejor de las alternativas, haya sabido entenderlo y seguir al pie del cañón. Y por último a toda la gente que me rodea día a día, que está cuando lo necesitas sin pedirlo y anima a que sigas adelante cuando flaquean las fuerzas.

Pablo Polo Garcimartín

Sevilla, 2019

Resumen

En el ámbito del control de procesos existe un gran interés en ofrecer la posibilidad de tener una resolución de problemas en línea de forma robusta y eficiente. En el caso concreto del control predictivo, muy extendido en la industria en la actualidad, al tener una base teórica y unos resultados afianzados en cuanto a prestaciones se refiere, se encuentra una barrera a la hora de su implementación en tiempo real. Ya sea por limitaciones sobre los sistemas físicos que no poseen la capacidad de cálculo necesaria o, en caso contrario, que se haya superado esta limitación pero sólo algunos ingenieros sean capaces de implementar semejantes estructuras de control.

Por ello en este trabajo se propone como objetivo el desarrollo de una aplicación que sin ceder en cuestiones de prestaciones en cuanto a fiabilidad de las soluciones aportadas por el controlador predictivo, proporcione la posibilidad de una resolución en línea de problemas de optimización.

Para llevar a cabo este proceso se recurrirá a la utilización de librerías de optimización externas, probadas y con resultados verificados, como serán las librerías de optimización Gurobi, las cuales ofrecen unos servicios de gran calidad en cuanto a soporte y libertad de utilización en multitud de plataformas. Pero la implementación de las mismas requerirá de un entorno que permita la gestión eficiente y en tiempo real de la adquisición de datos. Para este propósito se elegirá la utilización del entorno LabVIEW que aunque carezca de la facilidad para las manipulaciones complejas de cálculos matriciales y algoritmos de optimización, proporciona una gran sencillez para la adquisición y trata de información, a la vez que permite la adición de código ejecutable externo que permitirá el uso de las librerías de optimización.

Así pues en este trabajo se presentará la integración de librerías de optimización Gurobi en el entorno de LabVIEW, para su posterior aplicación a un problema de control en un proceso real.

Índice

Agradecimientos **viii**

Resumen **x**

Índice **xii**

Índice de figuras **xvi**

Notación y Conceptos **xxii**

1 Presentación y Objetivos **1**

2 Desglose del proceso a desarrollar **2**
2.1 Conocimientos previos: 3
2.2 Entornos de programación: 3
2.3 Organización de la programación de las diferentes partes: 4
2.4 Aplicación de laboratorio: 5

3 Librerías Gurobi **6**
3.1 Presentación de las librerías Gurobi: 6
3.2 Introducción para la utilización de las librerías Gurobi: 10
3.3 Análisis de los elementos que se tienen a disposición: 11
3.3.1 Carpeta “docs”: 12
3.3.2 Carpeta “examples”: 14
3.3.3 Resto de carpetas de interés: 15
3.4 Puesta a punto de las librerías dentro del entorno de Visual Studio: 16
3.4.1 Ejecutar Visual Studio y crear un nuevo proyecto: 17
3.4.2 Añadir códigos en C: 19
3.4.3 Inclusión de las librerías Gurobi: 20
3.4.4 Cambio de las opciones del proyecto y del compilador a "x64": 23
3.5 Análisis de códigos y funciones de mayor utilidad: 23
3.5.1 Funciones de inicialización y función de error: 24
3.5.2 Adición de términos a la función objetivo: 25
3.5.3 Modificación del sentido de la optimización "MODELSENSE": 27
3.5.4 Adición de restricciones al modelo: 27
3.5.5 Optimización y escritura del modelo: 29
3.5.6 Captura de valores: 29
3.5.7 Liberación del sistema: 30
3.6 Ejemplo de resolución de un QP con las librerías de optimización Gurobi: 30
3.6.1 Función de error: 31
3.6.2 Declaración de variables: 32
3.6.3 Creación de entorno y modelo: 33
3.6.4 Adición de variables al modelo: 33
3.6.5 Cambio del sentido de la optimización: 34
3.6.6 Adición de restricciones: 35
3.6.7 Optimización y escritura: 36

3.6.8 Captura de solución: 38
3.6.9 Liberación del sistema: 38

4 Control predictivo basado en modelo 39

4.1 Planteamiento de las bases sobre estrategias de control predictivo: 39

4.1.1 Estrategia general de los controladores predictivos: 39
4.1.2 Modelo del proceso: 41
4.1.3 Modelo de perturbaciones: 42
4.1.4 Función objetivo: 42
4.1.5 Respuesta libre y respuesta forzada: 44
4.1.6 Restricciones: 45

4.2 Presentación de estrategias de control predictivo: 45

4.2.1 *Dynamic Matrix Control* (DMC, Control de Matriz Dinámica): 46
4.2.2 *Model Algorithmic Control* (MAC): 48
4.2.3 *Predictive Functional Control* (PFC): 49
4.2.4 *Generalized Predictive Control* (GPC, Control Predictivo Generalizado): 51

4.3 Elección del control predictivo generalizado (GPC, Generalized Predictive Control): 57

4.4 Formulación práctica del cálculo recursivo de un GPC: 58

4.4.1 Resolución de la ecuación diofántica: 59
4.4.2 Desarrollo de funciones para cálculo vectorial y matricial en C: 68
4.4.3 Cálculo diferenciado de los diferentes términos de la función objetivo: 71
4.4.4 Programación del bucle de control de un GPC: 75

5 Creación e implementación de DLLs para su uso en el entorno LabVIEW 77

5.1 Puesta a punto del entorno Visual Studio para la creación de una DLL con la inclusión de las librerías Gurobi: 77

5.1.1 Ejecutar Visual Studio y creación un nuevo proyecto: 77
5.1.2 Modificación de la plataforma en opciones del proyecto y del compilador a "x64": 78
5.1.3 Inclusión de las librerías Gurobi: 79
5.1.4 Añadir código de encabezado: 81

5.2 Formato de los archivos de una DLL para su uso en el entorno LabVIEW: 84

5.2.1 Formato del archivo “*dllmain.cpp*”: 84
5.2.2 Formato del archivo de encabezado: 85
5.2.3 Formato general del archivo “*.cpp*” principal: 86

5.3 Proceso de implementación de una DLL en LabVIEW: 87

5.4 Ejemplo de implementación en LabVIEW de librerías para el cálculo de matrices de un GPC: 93

5.4.1 Creación de las funciones de cálculo: 93
5.4.2 Creación de las funciones exportables de la DLL: 95
5.4.3 Creación del instrumento virtual de LabVIEW: 96

6 Optimización de códigos para su uso en el entorno LabVIEW 98

6.1 Planteamiento completo de la implementación en LabVIEW de un GPC: 98
6.2 Planteamiento de la estructura de la función que realice el cálculo del GPC: 99
6.3 Planteamiento de la estructura de la función que realice el bucle de control: 100
6.4 Ampliación del uso de LabVIEW para mejorar las funcionalidades y limitaciones de la DLL: 102
6.5 Ejemplo completo de implementación de un GPC en LabVIEW: 105
6.6 Problemas de integración de una DLL que utiliza las librerías Gurobi dentro de LabVIEW: 110

7 Aplicación de laboratorio 112

7.1 Presentación del sistema real: 112

7.1.1 Conexión del sistema real y la tarjeta de adquisición: 113
7.1.2 Identificación del sistema: 115

7.2 Planificación de los experimentos: 118
7.3 Toma, exposición y análisis de resultados: 119
7.4 Perspectivas de mejora y ampliación: 122

Anexos 124

Anexo A: Resolución de un QP con las librerías de optimización Gurobi (QPejemplo) **124**

Anexo B: Resolución de la ecuación diofántica en Matlab (DiofanEQmatlab) **127**

Anexo C: Resolución de la ecuación diofántica en C (GPCscript) **129**

Anexo D: Implementación en LabVIEW de librerías para el cálculo matricial de un GPC (GPCdll) **134**

Anexo E: Script de resolución de un único blucle de control (GPCunciclo) **142**

Anexo F: Código completo de la implementación de una librería para el cálculo de un GPC en LabVIEW (GPCcompleto) **154**

Anexo G: Código completo con la adición de restricciones de entrada y salida (GPCrestrict) **169**

Referencias **186**

ÍNDICE DE FIGURAS

Figura 0-1. Diagrama de desglose del trabajo completo. 2
Figura 1-1. Aplicaciones Gurobi. 6
Figura 1-2. Interfaces aplicables. 8
Figura 1-3. Academia Center Gurobi. 8
Figura 1-4. Diagrama de introducción al análisis de las librerías Gurobi. 10
Figura 1-5. Carpeta principal de la instalación Gurobi. 11
Figura 1-6. Carpeta “docs” Gurobi. 12
Figura 1-7. Carpeta “examples” Gurobi. 14
Figura 1-8. Carpeta “include” Gurobi. 15
Figura 1-9. Carpeta “lib” Gurobi. 16
Figura 1-10. Proyecto nuevo Visual Studio. 18
Figura 1-11. Creación proyecto vacío Visual Studio. 18
Figura 1-12. Agregar código C Visual Studio. 19
Figura 1-13. Añadir archivo “.c” Visual Studio. 19
Figura 1-14. Propiedades de proyecto Visual Studio. 20
Figura 1-15. Propiedades proyecto versión de Windows. 20
Figura 1-16. Propiedades proyecto directorios de archivos de inclusión. 21
Figura 1-17. Agregar dirección de la carpeta “include”. 21
Figura 1-18. Agregar directorio de la carpeta “lib”. 21
Figura 1-19. Propiedades proyecto, Directorios de inclusión adicioneales. 22
Figura 1-20. Agregar librería estática de Gurobi al Vinculador. 22
Figura 1-21. Modificación plataforma a “x64”. 23
Figura 1-22. Ejemplo ecuaciones de un QP. 31
Figura 1-23. Inclusión de librerías en C. 31
Figura 1-24. Función de error de Gurobi. 31
Figura 1-25. Función externa para comprobación de errores de Gurobi. 32
Figura 1-26. Declaración de variables para Gurobi. 32
Figura 1-27. Creación del modelo y entorno en Gurobi. 33
Figura 1-28. Adición de variables lineales y cuadráticas en Gurobi. 33
Figura 1-29. Función de cambio de sentido de optimización. 34
Figura 1-30. Adición de restricciones en Gurobi. 35
Figura 1-31. Funciones de optimización y escritura de Gurobi. 36

Figura 1-32. Formato de un archivo LP para escritura del modelo en Gurobi. 36
Figura 1-33. Archivo resultado de una optimización en Gurobi. 37
Figura 1-34. Captura de solución a través de Gurobi. 38
Figura 1-35. Liberación del sistema. 38
Figura 2-1. Metodología control predictivo. 40
Figura 2-2. Modelo de respuesta impulsional y ante escalón. 41
Figura 2-3. Modelo basado en función de transferencia. 41
Figura 2-4. Modelo en espacio de estados. 42
Figura 2-5. Modelo de perturbaciones. 42
Figura 2-6. Función objetivo genérica. 42
Figura 2-7. Señales de entrada y salida para la respuesta libre y forzada respectivamente. 44
Figura 2-8. Ecuaciones de restricciones base. 45
Figura 2-9. Ecuación de predicción DMC. 46
Figura 2-10. Respuesta libre DMC. 46
Figura 2-11. Ecuación de predicción matricial DMC. 46
Figura 2-12. Función objetivo DMC. 47
Figura 2-13. Obtención del vector de señales de control. 47
Figura 2-14. Ecuación de predicción MAC. 48
Figura 2-15. Suma completa de la ecuación de predicción MAC. 48
Figura 2-16. Predicción en forma matricial MAC. 48
Figura 2-17. Función objetivo MAC. 49
Figura 218-18. Señales de control futuras PFC. 50
Figura 2-19. Función objetivo PFC. 50
Figura 2-20. Función objetivo PFC. 50
Figura 2-21. Señal de control minimizada PFC. 51
Figura 2-22. Modelo CARIMA. 51
Figura 2-23. Modelo GPC. 51
Figura 2-24. Modelo basado en función de transferencia. 52
Figura 2-25. Función objetivo general GPC. 52
Figura 2-26. Ecuación diofántica. 52
Figura 2-27. Multiplicación de la ecuación diofántica desarrollo. 53
Figura 2-28. Salida despejada en la ecuación diofántica. 53
Figura 2-29. Ecuación de predicción GPC. 53
Figura 2-30. Formato vectores ecuación diofántica. 54
Figura 2-31. Obtención sucesiva polinomios ecuación diofántica. 54
Figura 2-32. Obtención sucesiva F. 54
Figura 2-33. Obtención sucesiva G (1). 55
Figura 2-34. Obtención recursiva G (2). 55
Figura 2-35. Ecuación de predicción matricial GPC. 55

Figura 2-36. Ecuación de predicción GPC simplificada. 56
Figura 2-37. Función objetivo matricial GPC. 56
Figura 2-38. Desarrollo función objetivo GPC. 56
Figura 2-39. Minimización función objetivo GPC sin restricciones. 56
Figura 2-40. Modelo CARIMA para sistema multivariable. 57
Figura 2-41. Función objetivo GPC multivariable. 57
Figura 2-42. Inicialización cálculo recursivo ecuación diofántica en Matlab. 60
Figura 2-43. Cáculo recursivo matriz F en Matlab. 60
Figura 2-44. Cálculo recursivo matriz E en Matlab. 61
Figura 2-45. Convolución preparatoria para calcular G en Matlab. 61
Figura 2-46. Cálculo previo matrices GPC en Matlab. 61
Figura 2-47. Reorganización términos G definitiva en Matlab. 62
Figura 2-48. Funciones de reserva de memoria dinámica en C. 63
Figura 2-49. Función de convolución de vectores en C. 64
Figura 2-50. Inicialización valores de entrada y matrices de salida. 65
Figura 2-51. Inicialización y cálculo recursivo F en C. 65
Figura 2-52. Cálculo recursivo matriz E en C. 66
Figura 2-53. Cálculo recursivo de la matriz G en C. 66
Figura 2-54. Obtención de los valores de la respuesta libre dentro de G. 67
Figura 2-55. Colocación definitiva matriz Gf. 67
Figura 2-56. Función trapuesta de una matriz en C. 68
Figura 2-57. Función producto entre vectores en C. 68
Figura 2-58. Función producto entre matrices en C. 69
Figura 2-59. Función producto entre una matriz y un vector en C. 70
Figura 2-60. Función producto entre un vector y una matriz en C. 70
Figura 2-61. Modelo basado en función de transferencia. 71
Figura 2-62. Modelo basado en función de transferencia. 71
Figura 2-63. Cálculo término *(f-w)* de la función objetivo. 72
Figura 2-64. Cálculo del término independiente de la función objetivo. 72
Figura 2-65. Cálculo del término lineal de la función objetivo. 72
Figura 2-66. Traspuesta de G y cálculo de la matriz H. 73
Figura 2-67. Obtención de la matriz H con los términos cuadráticos definitivos. 73
Figura 2-68. Cálculo de los vectores de las variables cuadráticas. 74
Figura 2-69. Inicialización de variables del bucle de control 75
Figura 2-70. Inicialización de la salida y del vector de referencias futuras. 76
Figura 2-71. Actualización de las variables del bucle de control. 76
Figura 3-1. Creación de un archivo DLL en Visual Studio. 77
Figura 3-2. Opciones de creación de un proyecto en Visual Studio. 78
Figura 3-3. Elección del tipo de aplicación en Visual Studio. 78

Figura 3-4. Modificación opciónes plataforma en Visual Studio. 79
Figura 3-5. Desplegable propiedades de un proyecto DLL en Visual Studio. 79
Figura 3-6. Modificación versión de Windows de una DLL en Visual Studio. 80
Figura 3-7. Adición de librerías de entrada en el Vinculador en Visual Studio. 81
Figura 3-8. Desplegable de solución de una DLL iniciado por defecto. 82
Figura 3-9. Desplegable agregar nuevo elemento de encabezado en Visual Studio. 82
Figura 3-10. Creación de un archivo de encabezado en una DLL en Visual Studio. 83
Figura 3-11. Desplegable solución de una DLL completo. 84
Figura 3-12. Formato archivo *dllmain.cpp*. 85
Figura 3-13. Formato del archivo de encabezado de una DLL. 85
Figura 3-14. Formato de las librerías de inclusión típicas en una DLL con Gurobi. 86
Figura 3-15. Ejemplo definición de prototipos en C. 87
Figura 3-16. Desplegable paleta de funciones de LabVIEW. 87
Figura 3-17. Opciones de configuración del bloque de llamada a una librería en LabVIEW. 88
Figura 3-18. Selección de una librería externa en LabVIEW. 89
Figura 3-19. Variable de retorno de una librería externa en LabVIEW. 90
Figura 3-20. Definición de variables de una librería externa en LabVIEW. 91
Figura 3-21. Adición de matrices de dos dimensiones en LabVIEW. 92
Figura 3-22. Función de linealización de una matriz por filas en C. 92
Figura 3-23. Función de construcción de una matriz linealizada por filas en C. 93
Figura 3-24. Cálculo de las matrices de la ecuación diofáncita en C. 94
Figura 3-25. Comparación códigos para cálculo de matrices del GPC en C. 95
Figura 3-26. Funciones exportables de una DLL a LabVIEW. 95
Figura 3-27. Declaración de las funciones exportables en el archivo de encabezado. 96
Figura 3-28. Definición variables principales de una librería en LabVIEW. 96
Figura 3-29. Comprobador de horizontes en LabVIEW. 97
Figura 3-30. Diagrama de bloques de LabVIEW completo. 97
Figura 4-1. Diagrama LabVIEW para cálculo matricial del GPC 99
Figura 4-2. Diagrama de flujo del bucle de control del GPC 100
Figura 4-3. Diagrama LabVIEW para bucle de control del GPC 101
Figura 4-4. Comprobación de valores límite de variables manipulables por el usuario. 102
Figura 4-5. Tiempo de muestreo del modelo implementado en LabVIEW. 103
Figura 4-6. Visualización gráfica de variables en LabVIEW. 103
Figura 4-7. Selector de modo de control implementado en LabVIEW. 104
Figura 4-8. Visualización de parámetros a través de una llamada a librería en LabVIEW. 104
Figura 4-9. Interfaz de usuario en LabVIEW. 105
Figura 4-10. Diagrama de programación de LabVIEW completo. 106
Figura 4-11. Inicialización y reconstrucción de matrices del GPC. 106
Figura 4-12. Inicialización entorno y modelo de Gurobi. 107

Figura 4-13. Adición de todas las variables al modelo de Gurobi. 107
Figura 4-14. Adición de restricciones al modelo de Gurobi. 108
Figura 4-15. Optimización, captura de solución y liberación de Gurobi. 109
Figura 4-16. Selección de modalidad de control. 109
Figura 4-17. Actualización del bucle de control y toma de variables en línea. 110
Figura 4-18. Cálculo de dimensiones de vectores erróneo. 110
Figura 5-1. Sistema físico elegido. 112
Figura 5-2. Tarjeta de adquisición de datos. 113
Figura 5-3. Frontal de conexiones del sistema físico. 113
Figura 5-4. Conexión de la salida (arriba) y la entrada (abajo). 114
Figura 5-5. Conexión de las tierras. 114
Figura 5-6. Menú del bloque de adquisición de datos de LabVIEW. 115
Figura 5-7. Señal para realizar la identificación del sistema. 116
Figura 5-8. Cálculo de la ganancia para el flanco de subida. 116
Figura 5-9. Cálculo de la ganancia para el flanco de bajada. 117
Figura 5-10. Cálculo de la constante de tiempo para el flanco de subida. 117
Figura 5-11. Cálculo de la constante de tiempo para el flanco de bajada. 118
Figura 5-12. Prueba del sistema en modo automático. 119
Figura 5-13. Prueba del sistema pasando por el controlador externo. 120
Figura 5-14. Prueba del sistema modificando parámetros de sintonización. 121
Figura 5-15. Prueba del sistema comparando diferentes valores. 121

Notación y Conceptos

Hardware	Elementos físicos que componen un sistema electrónico
Software	Programa o conjunto de programas que dan soporte a un sistema informático
C	Lenguaje de programación C
C++	Lenguaje de programación C++
DLL/.dll	Librería de vínculos dinámicos
LIB/.lib	Archivos de librería estática
QP	Programación cuadrática
MPC	Control predictivo basado en modelo
GPC	Control predictivo generalizado
int	Variable tipo entero
dbl/double	Variable tipo double
Array	Conjunto de valores en disposición matricial
String	Variable en disposición de cadena de caracteres
Header	Archivos de cabecera
GRB	Gurobi/Librerías de optimización Gurobi
NULL	Valor asignado a una variable de Gurobi para que asigne valores por defecto
Env	Entorno definido en Gurobi
Model	Modelo definido en Gurobi
Constr	Restricciones definidas en Gurobi
Attr	Atributo definido en Gurobi
Par/Param	Parámetro definido en Gurobi
Set	Imponer un valor a un Atributo en Gurobi
Get	Toma de un valor de un Atributo en Gurobi
=	Igual
<=	Menor o igual
>=	Mayor o igual

1 PRESENTACIÓN Y OBJETIVOS

El objetivo de este trabajo es ofrecer una solución robusta y eficiente para resolver problemas de optimización dinámica en línea. Estos problemas suelen aparecer frecuentemente en el ámbito de control de procesos en general y de forma particular en el control predictivo basado en modelo (MPC), que es el origen de este trabajo. Para ello se van a utilizar diferentes herramientas con el objetivo de poder integrarlas obteniendo el mejor resultado de cada una de ellas.

Las técnicas de control predictivo basado en modelos cuentan a día de hoy con unos conceptos teóricos suficientemente desarrollados, y unos resultados en cuanto a estabilidad y prestaciones de sobra bien establecidos, pero se han encontrado con la barrera, muchas veces infranqueable, de su implementación en tiempo real dada la complejidad de cálculo que pueden llegar a alcanzar. Esta complejidad abarca dos vertientes, distintas pero complementarias. Por una parte el *hardware* necesario para la resolución de los problemas de optimización, se encuentra principalmente en el entorno industrial, el cual es muchas veces el factor limitante para que una aplicación con técnicas de control avanzado alcance la mayor eficiencia. Por otro lado, el avance de la tecnología ha permitido que se tengan a disposición equipos capaces de proporcionar la potencia de cálculo suficiente para llevar a cabo la carga de cálculo requerido, pero al costo de un esfuerzo de implementación fuera del alcance de una gran parte de los ingenieros dedicados al control, sobre todo en el entorno industrial. Por estas razones, la mayoría de las veces se optaba por la implementación de soluciones subóptimas en las que solía simplificarse el problema de forma que fuera o fácilmente implementable o fácilmente resoluble, pero a costa de ceder prestaciones en la solución.

Con estos datos puestos en perspectiva, un entorno como LabVIEW que permita una gestión eficiente y en tiempo real de la adquisición de datos, es empleada con asiduidad en ámbitos industriales como en laboratorios de desarrollo. Sin embargo carece de la sencillez necesaria para las manipulaciones complejas de cálculos matriciales y algoritmos de optimización. Por otro lado la implementación de algoritmos robustos de optimización en lenguajes de programación de bajo nivel (como podrían ser C o C++), capaces de asegurar la resolución en tiempo real y así adaptarse a cada problema, se convierte en una barrera infranqueable, sobre todo si hay que asegurar una solución factible y realizable en sistemas de control que pueden ser críticos en cuanto a su seguridad.

Por esta razón, acudir a librerías de optimización con algoritmos lo suficientemente probados y verificados, proporciona la garantía de poder resolver los problemas de optimización de una forma consistente. Hoy en día se tienen a disposición multitud de librerías de optimización, muchas de ellas de pago y otras de libre distribución. A este respecto se ha elegido el paquete de librerías de Gurobi por ser una referencia en cuanto a eficacia de los algoritmos disponibles así como, de código libre, disponibilidad para multitud de lenguajes de programación y plataformas y con un servicio y una comunidad extremadamente activa por parte no sólo de los propios usuario, sino que también de los desarrolladores.

Así pues en este trabajo se presentará la integración de librerías de optimización Gurobi en el entorno de LabVIEW, para su posterior aplicación a un problema de control en un proceso real.

2 DESGLOSE DEL PROCESO A DESARROLLAR

A la vista del objetivo que se ha planteado, será necesaria la integración de varios entornos de programación entre sí, lo que requerirá del uso de diferentes plataformas con las opciones que ofrece cada una de ellas y el lenguaje que se requiera para ser utilizada. Por esto será necesario poseer conocimientos previos en algunos de los apartados más básicos, por ejemplo, en cuanto a programación o en cuanto al uso básico de programas a nivel de usuario, aunque se proporcionará información detallada de los apartados más importantes del proceso, ahondando en todas las posibilidades que ofrecen las herramientas a disposición, ya que al ser varias integradas al mismo tiempo ha de tenerse claro las funciones que deben ser delegadas a cada una de ellas. Es por esto que a continuación se dispone de un diagrama aclaratorio del proceso desglosado de forma global:

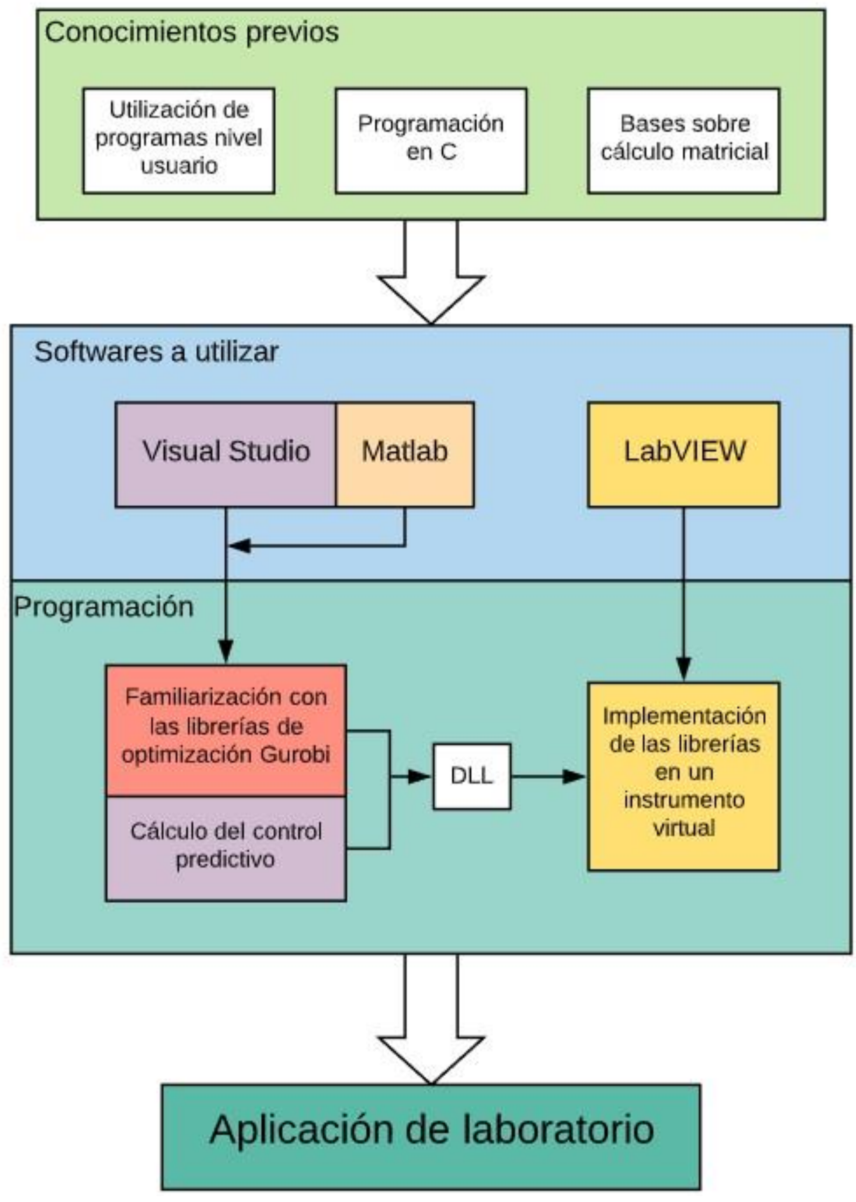


Figura 0-1. Diagrama de desglose del trabajo completo.

Procediendo en orden descendente a través de la figura anterior, se irán abordando los diferentes bloques para poner en perspectiva la información más relevante que contendrán.

2.1 Conocimientos previos:

Por el hecho de estar tratándose de un trabajo que engloba la utilización de diferentes herramientas, las cuales cada una tiene funcionalidades y formas diferentes, además de tratar un aspecto del área del control relativamente complejo, como es el control predictivo, se deben hacer ciertas concesiones en los apartados más básicos que se pueden abordar o no, por ello se necesitará tener ciertos conocimientos previos:

- Conocimientos de utilización de programas a nivel de usuario:

Dentro de cada una de las interfaces y posibles opciones que ofrecen los diferentes entornos, no se ofrecerán conocimientos más que los específicos e indispensables que se necesite desarrollar para este trabajo, por lo que tener cierta experiencia utilizando menús de aplicaciones de Windows a nivel de usuario será algo indispensable.

- Conocimientos de programación en C [21]:

Dado que la gran mayoría del peso computacional de este trabajo será desarrollado en lenguaje C, será indispensable tener cierta experiencia utilizando este lenguaje, dando por sentado que se tienen conocimientos de las estructuras básicas de un script en C, sintaxis básica, declaración y utilización de funciones, y más específicamente relacionado con lo que se va a abordar en este trabajo, utilización de memoria dinámica y, por último, conocimientos básicos sobre librerías dinámicas y las estructuras y formatos que pueden tener, serán puntos que harán la entrada a este tipo de programaciones mucho más accesibles.

Aunque en lo referente al último punto sobre conocimientos de librerías dinámicas, será un apartado que se aborde en bastante profundidad, no en lo que podrían ser los apartados teóricos más básicos de las mismas, pero sí en los puntos específicos de desarrollo e implementación de librerías dinámicas en los entornos que se van a manejar, ya que hay multitud de formas de abordar este tipo de tareas de creación de librerías propias.

- Conocimientos de cálculo matricial:

Teniendo presente la modalidad de trabajo que se ha tomado como objetivo, integrando librerías de cálculo dentro de un controlador predictivo, será totalmente imprescindible tener conocimientos sobre cálculo matricial, ya que todo el proceso que se realizará dentro de la programación del control contendrá una gran cantidad de cálculos puramente matriciales, en primer lugar realizados de forma teórica para matrices de cualquier combinación de dimensiones y, posteriormente, en un caso concreto a través de una aplicación práctica.

2.2 Entornos de programación:

A día de hoy se tienen a disposición multitud de entornos que pueden cumplir funciones similares a las que se van a precisar. Para realizar toda la programación en C se utilizará el entorno de ***Visual Studio***, el cual es recomendado por los desarrolladores de las librerías Gurobi para realizar aplicaciones en C que las incluya, los apartados de este entorno que se abordarán en posteriores apartados serán los correspondientes a la inclusión de las librerías en el entorno, toda la configuración previa a su utilización, además de la programación con las mismas en C. Más adelante se ampliará este conocimiento añadiendo los apartados para realizar este mismo proceso pero en caso de creación de librerías propias, eso sí, para su uso preferentemente en el entorno LabVIEW.

De manera adicional y en ningún momento necesaria, se pueden utilizar herramientas de programación auxiliares, como en el caso de este trabajo ***Matlab***, que en el caso que se precise una herramienta que proporcione más facilidades a la hora de la prueba de códigos durante el desarrollo.

Por último como ya se anticipó en el primer apartado, se utilizará el entorno ***LabVIEW*** [18] en labores principalmente de adquisición y gestión de datos en tiempo real, relegando la mayor parte del cálculo a las librerías programadas en Visual Studio, aunque con algunas pequeñas salvedades, que se pondrán en perspectiva posteriormente, aprovechando facilidades que puede dar LabVIEW a la hora de tratar la toma de datos de un usuario que pueda utilizar la aplicación.

2.3 Organización de la programación de las diferentes partes:

El orden tomado para la programación ha sido elegido por el desarrollo lógico de las diferentes partes, ya que es beneficioso para el proceso completo conocer al menos cada una de las partes que lo componen, pero realmente el orden de cada uno de los apartados previos a la aplicación en laboratorio podrían ser intercambiables, al estar realizando una programación en base a un modelo genérico.

- Familiarización con el uso de las librerías de optimización Gurobi:

Como punto de entrada más lógico se tomó la decisión de analizar y familiarizarse con el uso de las librerías Gurobi, siendo estas el elemento principal en torno al que gira este trabajo y sobre el cual se necesitan tener más conocimientos específicos. Ya que dentro del mundo de las librerías destinadas a realizar tareas en concreto, cada una de ellas necesita emplear unas formas, un método de instalación y unos códigos concretos para su correcto funcionamiento.

Es por este hecho que el primer punto de análisis serán las librerías de Gurobi junto a toda la documentación que se pone a disposición sobre ellas, además de un análisis de los apartados necesarios para la instalación de las mismas en Visual Studio para, posteriormente, adentrarse en la forma de programación de las librerías, mencionando así las funciones de mayor interés, y por último, un ejemplo completo de implementación de las librerías Gurobi para la resolución de un QP.

- Programación del control predictivo:

A continuación se tratará el desarrollo completo del control predictivo, eso sí, después de realizar un pequeño repaso sobre las características básicas que posee el control predictivo, y las diferentes alternativas que se pueden elegir. Posterior a esta exposición se profundizará más en los detalles de la modalidad de control predictivo elegida para este trabajo, el Control Predictivo Generalizado (GPC).

Y por último, ya en una faceta mucho más práctica, se proporcionará el desarrollo completo de la programación de un GPC para un modelo genérico de forma recursiva, y por último, el cálculo del bucle de control de forma detallada.

- Creación de un archivo DLL que comprenda todo el control:

De forma muy similar al primer apartado, y por orden lógico, se abordará la implementación de los códigos anteriormente desarrollados, dentro de un archivo DLL. En primer lugar, se proporcionarán los detalles de puesta a punto de Visual Studio para la creación de este tipo de librerías para que, posteriormente, sean exportables al entorno de LabVIEW.

Procediendo seguidamente con la organización de los códigos anteriormente creados, dentro de funciones modulares que realicen las tareas necesarias dentro del control predictivo, como pueden ser, preparación para el cálculo matricial o el propio bucle de control.

- Implementación de la DLL en un instrumento virtual en LabVIEW:

Mezclado con el anterior apartado, ya que van íntimamente unidos, se creará un instrumento virtual en el que se implementen las diferentes funciones que se crearon en la librería. Para esto se proporcionará toda la información referente al bloque de llamada de función necesario, todas las opciones que éste ofrece, además del ejemplo práctico anteriormente desarrollado implementado en un instrumento virtual completo.

2.4 Aplicación de laboratorio:

Como paso final para la prueba completa del funcionamiento de todas las partes anteriores, se realizará una aplicación real en el laboratorio, en este punto ya utilizando un sistema concreto que necesitará ser identificado y modelado como paso inicial, antes de congregarlo con la programación en LabVIEW. A la que se deberá de hacer ciertas modificaciones para verificar que funcione correctamente y todas las partes de la librería se ejecuten a un cierto tiempo de muestreo, sean modificables en línea y además se muestren resultados por pantalla, con el fin de realizar pruebas, tomar y analizar los datos finales sobre el experimento.

3 LIBRERÍAS GUROBI

3.1 Presentación de las librerías Gurobi:

Gurobi Optimization es un equipo de personas especializadas [1] y centradas en el desarrollo del mejor solucionador de problemas de optimización posible, con el mejor soporte para los clientes y un precio asequible, siendo utilizado en casi una veintena de diferentes tipos de industrias y más de 1400 empresas, con objetivos tan diversos como los que se pueden ver a continuación [2].

Ejemplo de Industrias

- Publicidad y Marketing
- Aeroespacial y Defensa
- Aerolíneas y Aeropuertos
- Automotriz
- Biotecnología, Medica y Farmacéutica
- Química y Petróleo
- Energía y Utilities
- Servicios Financieros
- Alimentos y Bebidas
- Gobierno
- Transporte Terrestre y Maritímo
- Automatización Industrial y Maquinaria
- Metales, Materiales y Minería
- Celulosa y Papel
- Comercio Detallista
- Semiconductores
- Programación de Deportes
- Cadena de Abastecimiento
- Telecomunicaciones

Ejemplo de Problemas Empresariales

Producción

- Optimización de inventario
- Mix de producción
- Localización de máquinas

Distribución

- Minimización de uso de combustible
- Planificación del mantenimiento
- Carga ligera de camiones

Abastecimiento

- Reordenamiento y almacenado de inventario
- Selección de vendedor
- Planificación de envíos

Finanzas

- Presupuesto de capital
- Gestión de efectivo
- Optimizacion de Ingresos

Inversiones

- Optimización de portafolio
- Clonación de fondos
- Gestión de bonos

Recursos Humanos

- Programación de fuerza de trabajo
- Asignación de oficinas

Figura 1-1. Aplicaciones Gurobi.

Entre los productos que el grupo proporciona, en su propia página web se puede encontrar:

- ***Gurobi Compute Server*** [3] Con el objetivo de la creación de aplicaciones de optimización de alto rendimiento, y así satisfacer las necesidades que tengan dentro de una empresa simplificando la tarea de desarrollo e implementación de dicha aplicación. El sistema combina una librería que envía trabajos de optimización a potentes servidores, incluyendo en éstos, la puesta en cola, balance de carga y manejo de estos trabajos. Resultando en el desarrollo de aplicaciones fiables y escalables.

  En su propia página web se desarrollan un poco más en detalle todas las características que ofrece este servicio, además de las ya mencionadas se puede disponer de optimización distribuida, seguridad añadida a la comunicación y además otras herramientas de administración como, ajuste de prioridades de trabajo, desconexión de servidores para tareas de mantenimiento o revisión de información de uso de la propia aplicación.

  Gurobi garantiza al menos un mínimo de funciones que siempre son proporcionados por el *Compute server,* uso de múltiples plataformas de clientes, no requerimiento de licencias y un servicio técnico accesible. El uso de esta herramienta se puede realizar a través de un servidor propio o a través de la nube de Gurobi (*Gurobi Instant Cloud*), otro de los servicios que ponen a disposición de los clientes que lo requieran, con el que también se puede acceder al propio servicio técnico además de forma local.

- ***Gurobi Instant Cloud*** [4]: Utilización simple de la plataforma Gurobi a través de la nube, únicamente requiriendo la precarga de un programa exclusivo de Gurobi, en la cantidad de ordenadores que se precise para ser utilizado, o a través de un navegador de internet si se dispone de un ordenador con Windows, Linux o Mac como sistema operativo.

  En la propia página web se puede obtener más información, además de acceso a la propia aplicación e instrucciones sobre su uso, sobre funcionalidades de este producto, siendo entre las más destacables la facilidad de su utilización y compatibilidad con los sistemas operativos mencionados anteriormente, obviando el servicio técnico completo que ofrece Gurobi sobre sus productos.

  Como último punto, ya que se está hablando de servicios de trabajo en la nube, Gurobi ofrece centros de datos con los que comunicarse alrededor de todo el mundo, América, Asia y Europa, ofreciendo varios puntos dentro de cada uno para reducir los problemas de latencias a la hora de la comunicación.

- ***Gurobi Optimizer*** [5]: Motor de optimización computacional creado por Gurobi, utilizado en multitud de tipos de industrias y empresas, como se ha mostrado anteriormente, y diseñado desde cero para aprovechar las arquitecturas y procesadores de mayor actualidad, permitiendo así una mayor flexibilidad para los diferentes tipos de usuarios que lo utilicen y una variedad en los tipos de problemas a los que hace objetivo:

  - Programación lineal (LP, *Linear Programming*).
  - Programación lineal entera mixta (MILP, *Mixed Integer Linear Programming*).
  - Programación cuadrática entero mixta (MIQP, *Mixed Integer Quadratic Programming*).
  - Programación cuadrática (QP, *Quadratic Programming*).
  - Programación restringida cuadráticamente (QCP, *Quadratic Constrained Programming*).
  - Programación entera mixta restringida cuadráticamente (MIQCP, *Mixed Integer Quadratic Constrained Programming*).

Gurobi además ofrece una gran libertad a la hora de elegir un interfaz de modelado o lenguaje de programación, para así maximizar la diversificación del uso de su producto y la facilidad del cliente para elegir un método con el que esté más familiarizado:

- Interfases orientado a objetos para C++, Java, .Net y Python
- Interfases orientado a matrices para C, MATLAB® y R
- Enlaces a lenguajes de modelado estándar: AIMMS, AMPL, GAMS y MPL
- Enlaces a Excel a traves de Premium Solver Platform y Risk Solver Platform

Figura 1-2. Interfaces aplicables.

Este último producto es el que se utilizará para el desarrollo trabajo, siendo el objetivo principal del mismo la implementación de unas librerías de optimización para aplicaciones en control predictivo, Gurobi probablemente sea una de las mejores alternativas en el mercado por diferentes razones:

- Flexibilidad para la obtención de licencias de uso:

Existen una amplia variedad de tipos de licencias que Gurobi ofrece para el uso de sus productos, licencias perpetuas enfocadas para el uso empresarial, para un equipo único y una sola persona designada, para un equipo y usuarios añadidos o para un número ilimitado de equipos. También se ofrecen servicios de una índole similar al anterior pero para el uso a través de la nube, aunque lo que más interesa a este respecto son las licencias de uso académico, para lo cual Gurobi tiene un apartado entero [6] dedicado a la enseñanza.

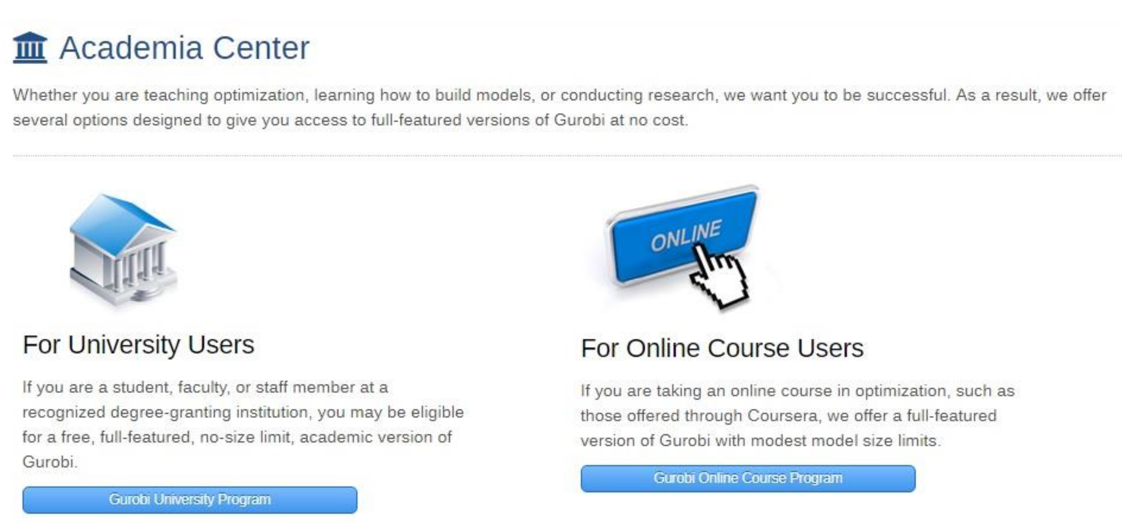


Figura 1-3. Academia Center Gurobi.

Dentro de este apartado se puede elegir entre la obtención de licencias para el uso de universidades o para la impartición de cursos en materia de optimización, sea cual sea el objetivo con el que se adquiera una licencia de este tipo es necesaria la dada de alta de una cuenta dentro de la propia página de Gurobi, y simplemente con esto, posteriormente de haber leído lo que se ofrece en cada uno de los tipos de licencia y los requerimientos que hay que cumplir para cada una, siguiendo los pasos de instalación y obtención de una clave, se puede acceder al producto durante, por ejemplo, 12 meses en el caso de una licencia académica, la cual puede ser actualizada anualmente sin problemas.

- Ritmo constante de actualizaciones del *software*:

Cuando se comenzó el desarrollo de este trabajo, la versión más actual de las librerías Gurobi era la versión 7.5.2, la cual es la que se ha usado para todo lo referente a optimización en este proyecto, poco tiempo después se hizo pública la versión 8.0 de las librerías y hace escasas semanas de la fecha de publicación de este trabajo se hizo pública la versión 8.1 de estas librerías, por lo que en el lapso de aproximadamente un año se han publicado tres versiones diferentes, con funciones añadidas, de las librerías, recalcando más aún el constante desarrollo de la aplicación.

- Activo soporte técnico:

Desde los inicios del grupo, los propios trabajadores utilizaban la facilidad que ofrece Google para la creación de un grupo (llamado *Gurobi Optimization*), al que poder acceder de manera gratuita, para resolver desinteresadamente las dudas de los diferentes usuarios que utilizaban su herramienta para multitud de variedad de proyectos, proveyendo respuestas, documentación y ayuda de forma directa y rápida. Pero el pasado mes de abril optaron por actualizar esta forma de dar soporte online a los usuarios a una propia sección dentro de su página web [7], permitiendo así tener una organización sobre los temas que se abordan, para facilitar la búsqueda a usuarios nuevos en el futuro y tener libertad a la hora de mostrar este tipo de contenido con, un enlace para últimos anuncios, preguntas frecuentes, artículos de interés, actividad más reciente en el foro o una simple opción de búsqueda para los usuarios.

Toda la documentación y todas las versiones del producto se pueden obtener a través de la página web de Gurobi, posteriormente al dar de alta una cuenta dentro de la misma web [8] se da acceso a todo el contenido, de forma online o de forma local descargando los archivos de la versión que se elija. Dentro de los archivos que se proporcionan se incluye una guía completa sobre la obtención de una licencia, instalación y uso de las librerías, los cuales se abordarán en detalle en los siguientes apartados.

3.2 Introducción para la utilización de las librerías Gurobi:

Conocidos los servicios que Gurobi pone a disposición de sus usuarios y las ventajas que pueden ofrecer frente a otros desarrolladores de librerías de optimización, a continuación, se dará una introducción visual a la forma en la que se va a proceder en los siguientes apartados, en los que se hará un análisis en detalle de todo lo que ofrece y es relevante para este trabajo en la documentación disponible de las librerías Gurobi, la estructura que se debe utilizar para su implementación en el entorno de programación, junto con algunas de las funciones de mayor utilidad para cada uno de los apartados.

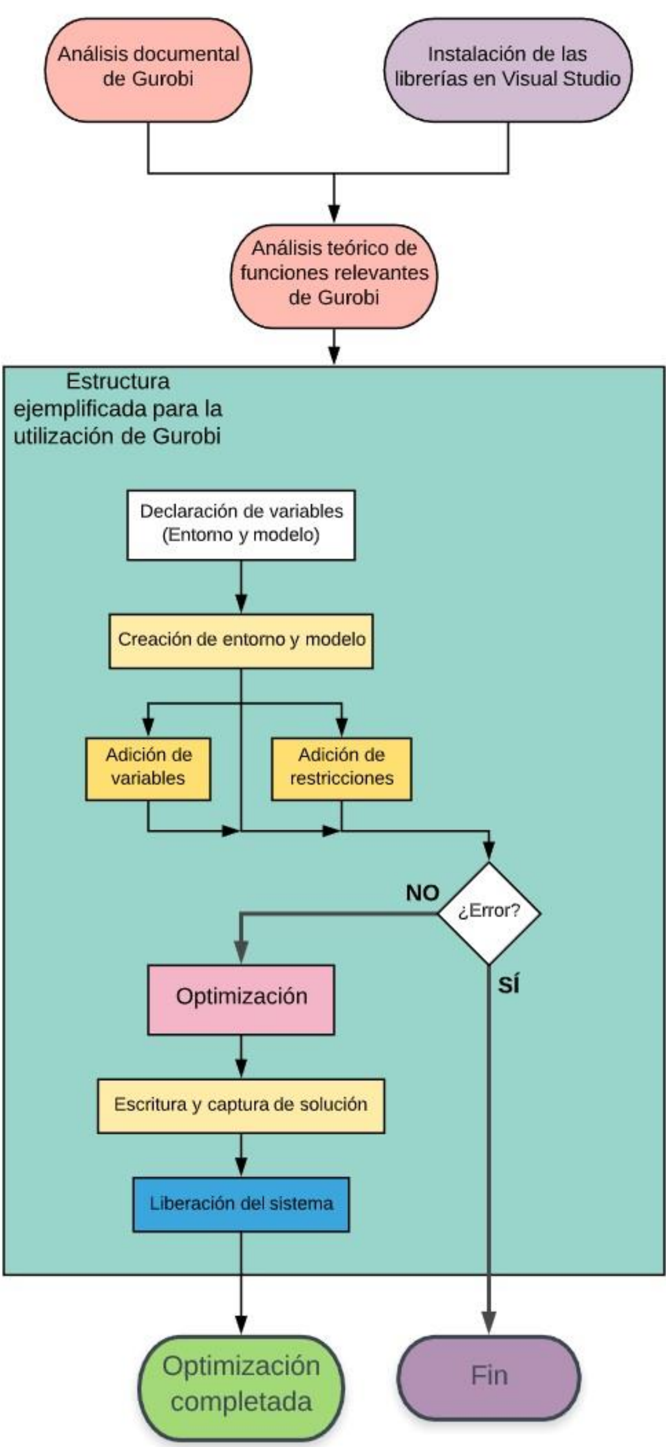


Figura 1-4. Diagrama de introducción al análisis de las librerías Gurobi.

3.3 Análisis de los elementos que se tienen a disposición:

Además de poder acceder a todos los archivos de las últimas versiones librerías de optimización de la forma anteriormente mostrada, también a través de su propia web se puede acceder a toda la documentación disponible para diferentes sistemas operativos [9].

Mencionado esto, ya que es importante destacar que los propios desarrolladores ponen toda la documentación, códigos y ayuda posible a libre disposición sin que sea necesaria la instalación de su producto, ahora se va a analizar la documentación existente dentro de los ficheros que se encuentran dentro de la última versión de las librerías.

Según se procede a ejecutar el instalador, después de extraer los archivos en el directorio que se elijan, se tendrá a disposición la siguiente distribución de carpetas:

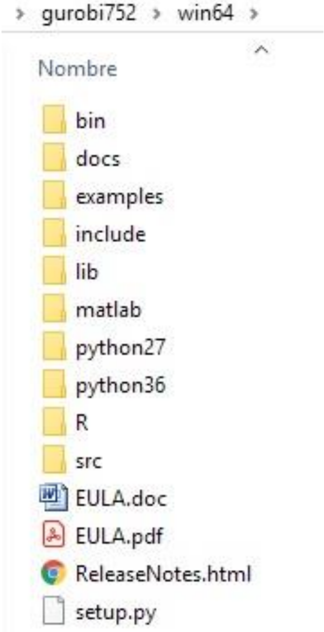


Figura 1-5. Carpeta principal de la instalación Gurobi.

Entre las carpetas que requieren atención y análisis las carpetas "docs", "examples", "include" y "lib" (aunque éstas dos últimas no requieran análisis, propiamente dicho, al contener únicamente los archivos de cabeceras de las librerías que se van a utilizar y los archivos de librería estáticas, "*.lib*" respectivamente), las demás carpetas contienen los archivos necesarios para el uso de las librerías en entornos que no van a ser utilizados en este trabajo.

3.3.1 Carpeta “docs”:

Dentro de la carpeta "docs" se disponen la siguiente serie de subcarpetas:

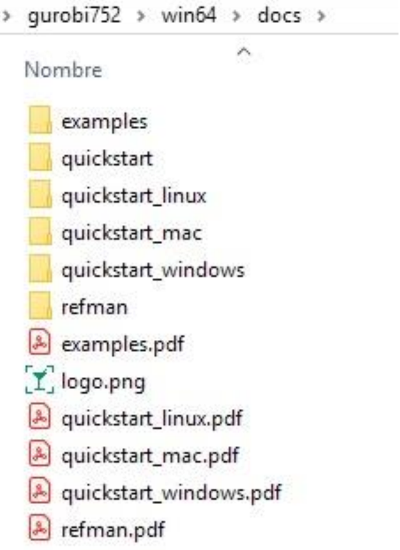


Figura 1-6. Carpeta “docs” Gurobi.

Y como se ha hecho referencia antes, todos los documentos están disponibles de manera online, por lo que aquí los desarrolladores proporcionan los enlaces referentes a todos los apartados de la documentación, que aquí se encontrará concentrada en archivos “*.pdf*”, separados en diferentes archivos “*.html*” que están enlazados a su respectiva página dentro de la web de Gurobi en el navegador predeterminado.

3.3.1.1 "*examples.pdf*":

Documento en el cual se hace un recorrido a través de todos los códigos de ejemplo que ofrecen los desarrolladores, en primera instancia se expone una lista comentando el objetivo que cumple cada código de ejemplo de una manera general, posteriormente se proporciona información y explicaciones básicas en todos los lenguajes a los que se le da soporte sobre:

- Funciones de carga y solución de modelos desde archivo.
- Funciones de construcción de modelos.
- Información sobre elementos de modelado adicionales.
- Funciones sobre modificación de modelos.
- Funciones sobre cambio de parámetros.
- Automatización de puesta a punto de parámetros.
- Diagnóstico de irrealizabilidad.
- Funciones de retorno de información durante la optimización de un modelo.

3.3.1.2 "*quickstart_windows/linux/mac.pdf*":

Documento en el cual (como se explica dentro de su propio primer punto de introducción) se da una introducción básica a las librerías Gurobi, dependiendo del sistema operativo que se esté utilizando (entre Windows, Mac o Linux), y específicamente en el caso que es de real interés, en sistemas operativos Windows, se abordan temas desde:

- Información sobre la obtención de una licencia de Gurobi.
- Utilización y puesta a punto de una licencia Gurobi.
- Simple guía de instalación.
- Explicación sobre el uso del optimizador de Gurobi a través de línea de comandos y resolución de un modelo simple a través de la misma.
- Explicación exhaustiva de un código simple de ejemplo (ya presentado en el documento anterior), pero además poniendo mucho hincapié en cada uno de los apartados que se van abordando y dando explicaciones al respecto de cada uno de ellos, no sólo mostrando el código de ejemplo. Esta explicación exhaustiva que se realiza de un código de ejemplo en específico se realiza para todos los lenguajes de programación a los que se le da soporte con la misma profundidad y detenimiento.
- Por último, y de hecho algo a lo que no se hace alusión dentro del propio índice de este documento, de bastante relevancia a la hora de la implementación de las librerías, al final de cada una de las explicaciones detalladas del código de ejemplo para cada uno de los lenguajes, se aborda un último apartado sobre recomendaciones a la hora de compilar y utilizar el ejemplo en del sistema. En caso del ejemplo en C los desarrolladores recomiendan la utilización de los archivos con todos los códigos de ejemplo en Visual Studio, y en el caso de C++ se comentan cuáles de los diferentes archivos de librerías que se ponen a disposición y en qué casos utilizar unos u otros (ya que como se expuso a la hora de analizar el contenido de los propios archivos de librerías, será necesaria la utilización de las librerías de C++ para la compilación de librerías propias).

  Siendo los archivos acompañados de las siglas "MD" las librerías que se usarán para crear DLLs (*Dinamic Link Library*, librería de vínculos dinámicos), y los archivos acompañados de las siglas "MT" se utilizarán si lo que se quiere crear son librerías estáticas (archivos "*.lib*"). Además se avisa de que a la hora de incluir las librerías en lenguaje C++ (y con ello su respectivo archivo de librería estático) se deben incluir también la librería del lenguaje C para que funcionen correctamente los programas que sean creados a través de esos archivos.

3.3.1.3 "*refman.pdf*":

Último documento de real interés que se pone a disposición, en este caso este sería el documento más técnico de todos, siendo un manual de referencia sobre todas las funciones que se ponen a disposición dentro de las librerías, divididos por lenguaje de programación y dentro de cada uno por objetivo de uso de cada una, en orden de exposición para el caso de lenguaje C:

- Funciones para creación y destrucción de entornos.
- Funciones para creación y modificación de modelos.
- Funciones para resolución de modelos.
- Funciones para la obtención de información del modelo.

- Funciones de lectura y escritura de modelos.
- Funciones para tratar atributos.
- Funciones el manejo y la puesta a punto de parámetros.
- Funciones para la monitorización del progreso.
- Funciones para la modificación del comportamiento del optimizador.
- Funciones para el trato de errores.
- Rutinas avanzadas.

Dentro de los demás lenguajes se abordarán funciones similares, pero ordenadas de diferentes formas y con añadidos o modificaciones para cada uno de los lenguajes en específico, las cuales no serán expuestas en este trabajo al basarse únicamente en la programación el lenguaje C.

Por último quedaría un documento por comentar, éste sería "*remoteservice.pdf*", fue añadido en la versión 8.0 de las librerías Gurobi pero este trabajo al haber sido realizado con la versión 7.5.2 aún no había sido publicado, igualmente trata una función que en este caso no va a ser explotada, que sería el uso de las librerías en un servidor en remoto, cosa que no concierne.

3.3.2 Carpeta "examples":

Dentro de la carpeta "examples" se encuentran una serie de subcarpetas, que representan todos los lenguajes de programación disponibles en el que estarán los archivos ejecutables de la lista de ejemplos, que estaba expuesta dentro del documento "*examples.pdf*" anterior. Pero los desarrolladores además de poner a disposición directamente los archivos con los códigos, ponen a disposición dos carpetas adicionales más:

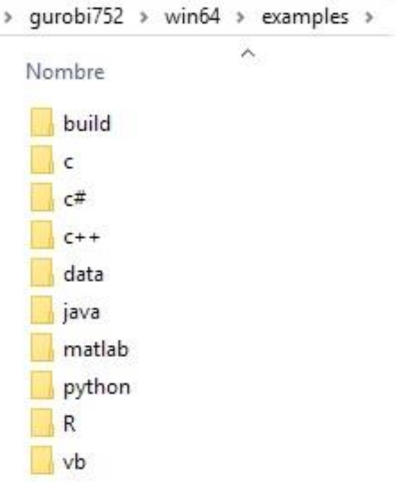


Figura 1-7. Carpeta "examples" Gurobi.

3.3.2.1 Carpeta “data”:

Contiene archivos cuyo propósito es ser utilizados como ejemplos y práctica se tiene interés en utilizar el optimizador de Gurobi a través, directamente, de la consola de comandos del ordenador, por lo cual no es algo que sea de interés para este trabajo, más allá de probar el funcionamiento del optimizador a través de algún archivo creado con un modelo simple en él.

3.3.2.2 Carpeta “build”:

En cambio, en comparación con la anterior carpeta, en ésta sí se tiene información que puede servir para practicar, comprender y ejecutar los ejemplos básicos que los desarrolladores de Gurobi proporcionan, al contener varios archivos (para las diferentes versiones de Visual Studio 2015 y 2017) con proyectos completos que contienen todos los códigos que se mostraron en el documento "*examples.pdf*" en lenguaje C, C++, C# y Visual Basic.

3.3.3 Resto de carpetas de interés:

Dentro de las demás carpetas sólo quedan los archivos referentes a las librerías Gurobi pertenecientes a todos los lenguajes de programación a los que se les da soporte, además de los archivos de la propia ejecución de las librerías Gurobi (exactamente los que se crean de manera automática cuando se realiza la instalación de las mismas).

Las carpetas que serán de mayor interés en cuanto a los archivos de las librerías propiamente dichos estarán incluidas dentro de:

3.3.3.1 Carpeta “include”:

Esta carpeta contiene únicamente dos archivos, "*gurobi_c.h*" y "*gurobi_c++.h*", ambos incluyen las cabeceras de todas las funciones disponibles dentro de las librerías en lenguajes C y C++, y ambas van a ser de vital utilidad, en este apartado en el que únicamente se ejecutarán códigos de pruebas de las funciones en C para probar y entender cómo funcionan las librerías, se utilizará el archivo "*gurobi_c.h*". Pero a la hora de, posteriormente, trabajar con librerías de vínculos dinámicos (DLL), las cuales serán ejecutadas a través de códigos en formato ".cpp" (lo que es decir, en leguaje C++), será necesario incluir las cabeceras del archivo "*gurobi_c++.h*" para poder utilizar las librerías de optimización correctamente.

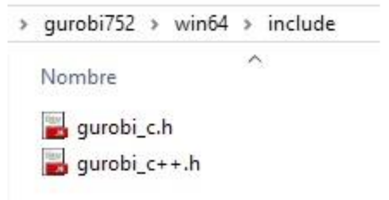


Figura 1-8. Carpeta “include” Gurobi.

3.3.3.2 Carpeta “lib”:

Esta carpeta incluye, propiamente dicho, el código de las funciones de las librerías de optimización. Éste está en formato de librería estática "*.lib*", aunque realmente a la hora de incluirlo en Visual Studio no habrá diferencias, en comparación con los pasos que se deben seguir para incluirlas en los proyectos, en el caso que las librerías fueran estáticas, "*.lib*", o dinámicas "*.dll*" (este tema será abordado en el siguiente apartado del proyecto con más profundidad). Esta carpeta contiene el archivo correspondiente a las librerías para lenguaje C, "*gurobi81.lib*" (el número que está contenido en el nombre de este archivo variará según la versión de Gurobi que se esté utilizando, en el caso de este trabajo las librerías serían específicamente "*gurobi75.lib*"), y los archivos correspondientes a las librerías en C++ (los restantes pertenecen a otros lenguajes de programación y no serán analizados).

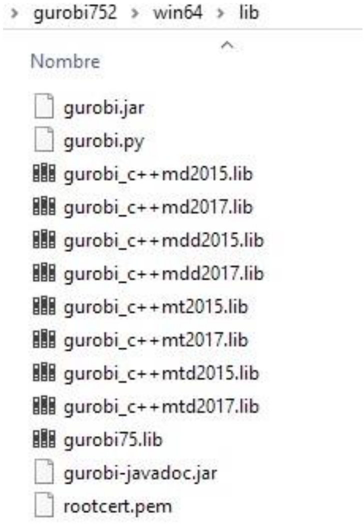


Figura 1-9. Carpeta "lib" Gurobi.

En lo referente a este último lenguaje de programación se encuentran ocho archivos diferentes, y como se dijo cuándo se analizó el documento "*quickstart.pdf*", donde se podrá escoger en primer lugar la versión de Visual Studio (en las que Gurobi da soporte para la versión 2015 y 2017), y dentro de cada uno de estos grupos se debe seleccionar el correspondiente a las necesidades del tipo de librería que se va a crear, los archivos que vienen acompañados de las siglas "MD" serán utilizados para incluirlos en librerías dinámicas y los archivos acompañados de las siglas "MT para librerías estáticas, en el caso particular de este proyecto a la hora de crear librerías se incluirá el archivo "*gurobi_c++mdd.lib*", además de incluir el archivo de librerías de C "*gurobi75.lib*".

3.4 Puesta a punto de las librerías dentro del entorno de Visual Studio:

Previamente a entrar en la integración de las librerías dentro del entorno de programación los desarrolladores recomiendan adquirir una licencia para obtener una clave de su producto y acceder a la utilización de las librerías a través de la consola de comandos (todo este procedimiento está completamente explicado en los apartados "2-Obtaining a Gurobi License" y "4-Retrieving and Seting up a Gurobi License", dentro de los documentos pdf "quickstart" para cada uno de los sistemas operativos a los que se les da soporte) para tener la primera toma de contacto con la utilización de las librerías y su funcionamiento (cuyas explicaciones están expuestas dentro del documento "quickstart" dentro del apartado "5-Solving a Simple Model _ The Gurobi Command Line").

Esta opción que se pone a disposición gracias a los desarrolladores de Gurobi, es un muy buen punto de partida para entender que las librerías de Gurobi y el proceso de optimización de modelos se desarrollan dentro de un entorno (entorno que debe ser creado cada vez que se inicia un programa en el que se lleve a cabo una optimización) al que agregar el modelo, dándole a conocer al optimizador la cantidad de variables que tiene que tener en cuenta (lineales y cuadráticas, dependiendo del tipo de modelo con el que se trabaje), la función objetivo y las restricciones (cada una de las cuales será añadida de forma individual y haciendo uso de una función diferente que realice cada una de las tareas).

Pero al margen de esto, el entorno de desarrollo que se utilizará será Visual Studio, si es posible en su última versión disponible para descargar, y para empezar se deben poner en perspectiva varios puntos importantes:

- Visual Studio es un software excesivamente genérico:

Si se profundiza dentro del entorno Visual Studio, fácilmente se puede deducir por el número de opciones de inicio que se proporciona, o dentro de los menús destinados a configuración, se da la opción de modificar absolutamente todo lo que se pueda imaginar o programar, y además en multitud de lenguajes. Este hecho no es relativamente una desventaja para elegir este entorno por encima de otros, pero al combinar softwares que permiten muchas modalidades de trabajo diferentes, puede ocurrir que se generen incompatibilidades o que existan más problemas al realizarlo. Esto está siendo expuesto por el hecho de que en este trabajo se van a mezclar varios software; Gurobi, Visual Studio y LabVIEW, estando enfocados en abarcar muchas áreas diferentes, por lo que no será muy extraño encontrar dificultades durante las integraciones entre ellos.

- La integración de librerías no es totalmente directa:

A raíz de esto último que se acaba de comentar se puede suponer que al integrar las librerías de Gurobi dentro de Visual Studio no sea una tarea directa, por mucho que lo deba parecer en primera instancia y en teoría, una tarea que debería ser automática, incluir una librería dentro de un proyecto en Visual Studio, puede dar multitud de problemas a la hora de encontrar la librería dentro del directorio que se seleccione, al incluir la cabecera y la librería juntas en el proyecto, es por esto que además de, en los apartados que vienen a continuación, explicar con detalle todos los pasos que se deben seguir para llegar al objetivo de incluir las librerías de Gurobi en Visual Studio, también se va a proporcionar un archivo de plantilla en el que ya están incluidas las librerías y preparado el proyecto para su utilización (eso sí, sólo y únicamente, si se han instalado las librerías de Gurobi dentro del disco duro principal del ordenador, como una instalación por defecto).

- Visual Studio no está creado para realizar proyectos pequeños:

Es cierto que dentro de la solución que se ha creado dentro de un proyecto vacio, se pueden incluir además multitud de proyectos dentro, pero lo que realmente ocurre es que en cada uno de estos se deben incluir las librerías que se precisen pero si esto se hace en varios proyectos aumentará el tamaño del archivo completo del proyecto. Cuando se menciona el desmesurado tamaño de archivo, es porque, con facilidad, pueden superar la marca de los 100 MB, lo cual es bastante y lo que es extraíble de esto es que Visual Studio está realmente hecho para trabajar en proyectos complejos.

Después de haber dejado claro estos puntos, a continuación se procederá con la puesta a punto de las librerías de Gurobi dentro del entorno Visual Studio.

3.4.1 Ejecutar Visual Studio y crear un nuevo proyecto:

Posteriormente a su instalación (la cual se deberá realizar a través de la página oficial [12]), eligiendo la versión que cumpla mejor las necesidades dentro de lo que se ofrece), se ejecutará la aplicación y dentro de la pantalla de inicio se seleccionará dentro de la barra de herramientas entrar en "Archivo > Nuevo > Proyecto" o dentro de la página de inicio bajo la sección "Nuevo Proyecto" al final del todo se encuentra la opción "Crear nuevo proyecto...", cualquiera de las dos opciones abrirá el siguiente menú:

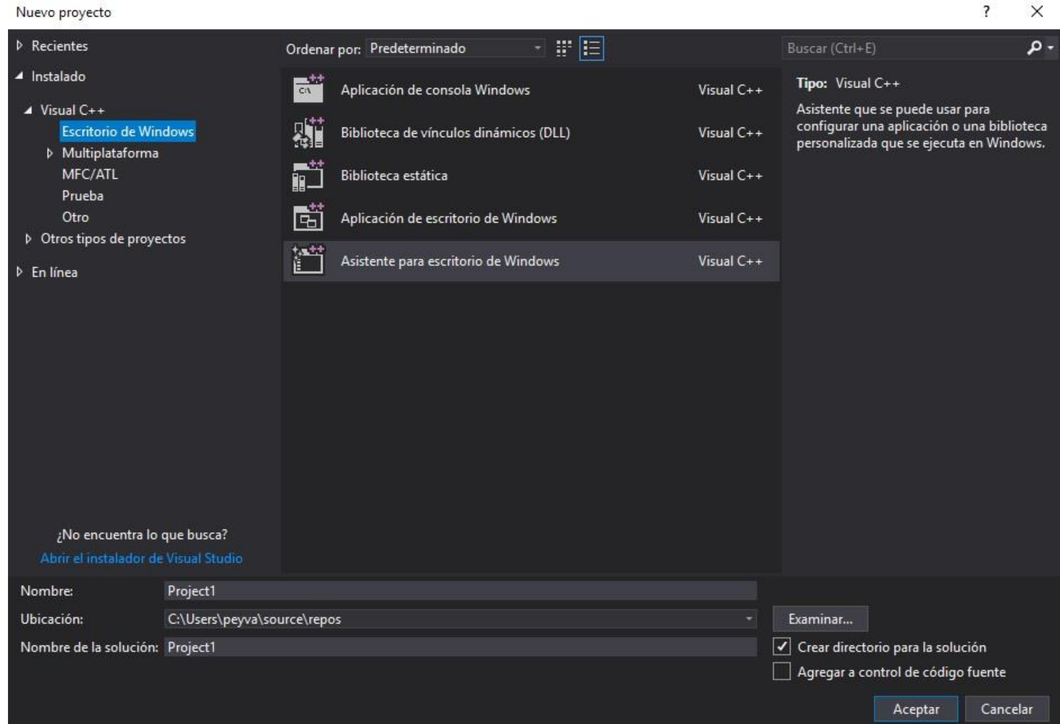


Figura 1-10. Proyecto nuevo Visual Studio.

Por defecto debería estar seleccionada la pestaña del desplegable a la izquierda "Visual C/C++" (en caso contrario se seleccionará) y se verán a la derecha los tipos de proyectos por los que se puede optar, aquí estarán además de las tres primeras opciones las plantillas personalizadas creadas, y se elegirá la opción de "Proyecto Vacío" y se creará un proyecto que no contiene ningún archivo en la solución (apartado desplegable que por defecto se encuentra a la derecha de la ventana).

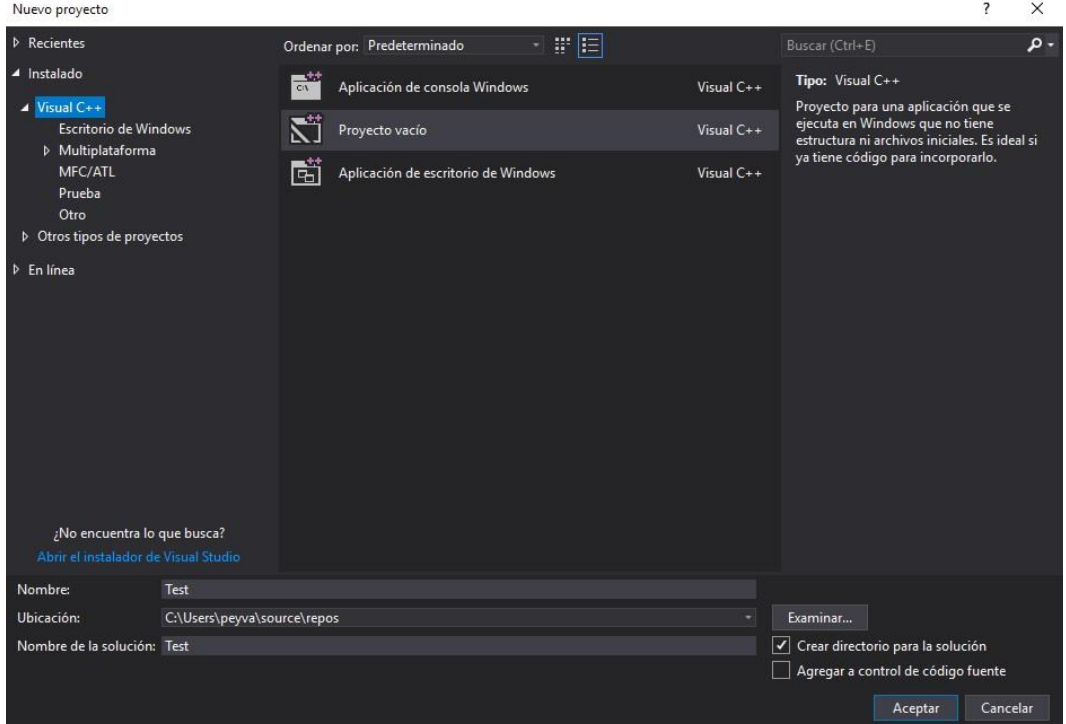


Figura 1-11. Creación proyecto vacío Visual Studio.

3.4.2 Añadir códigos en C:

Para añadir cualquier tipo de elemento lo que se debe hacer es hacer clic derecho en las carpetas que se crean dentro del desplegable de solución del proyecto, y aquí y en los menús que salen siempre la primera opción es la que permite realizar esta tarea. En el caso específico para añadir un código en C, se hará clic derecho en la carpeta "Archivos de recurso", posteriormente en "Agregar >Nuevo elemento", se abrirá un desplegable como el que se contempla a continuación.

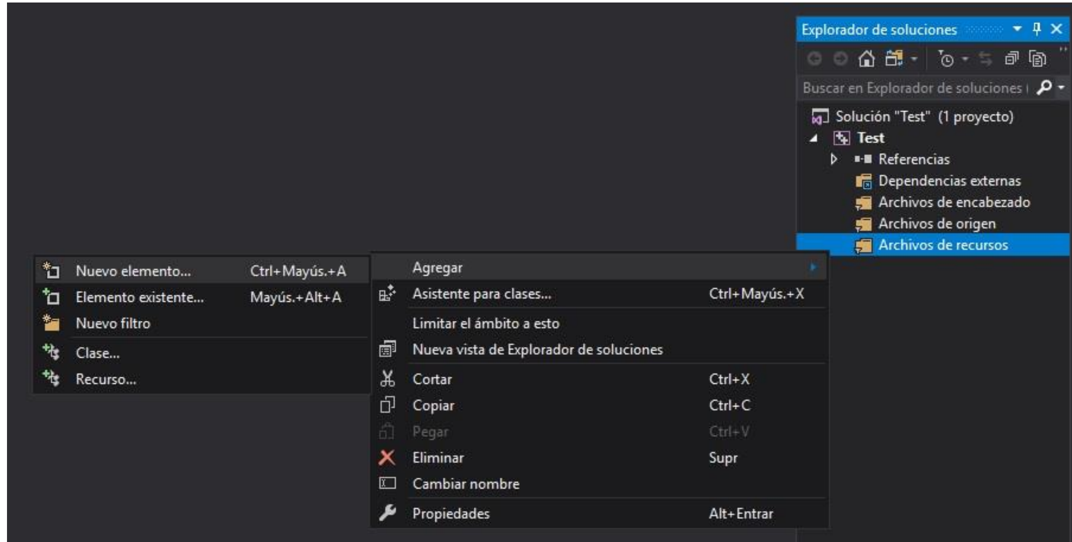


Figura 1-12. Agregar código C Visual Studio.

Cuando aparezca la ventana para la creación del nuevo elemento que se quiere añadir a la carpeta de recurso se especificará en el nombre del archivo, en la barra en la parte de abajo de la ventana, el nombre que se le quiera dar seguido de ".c" para especificar que el archivo va a contener código programado en C (Figura *1-12*)

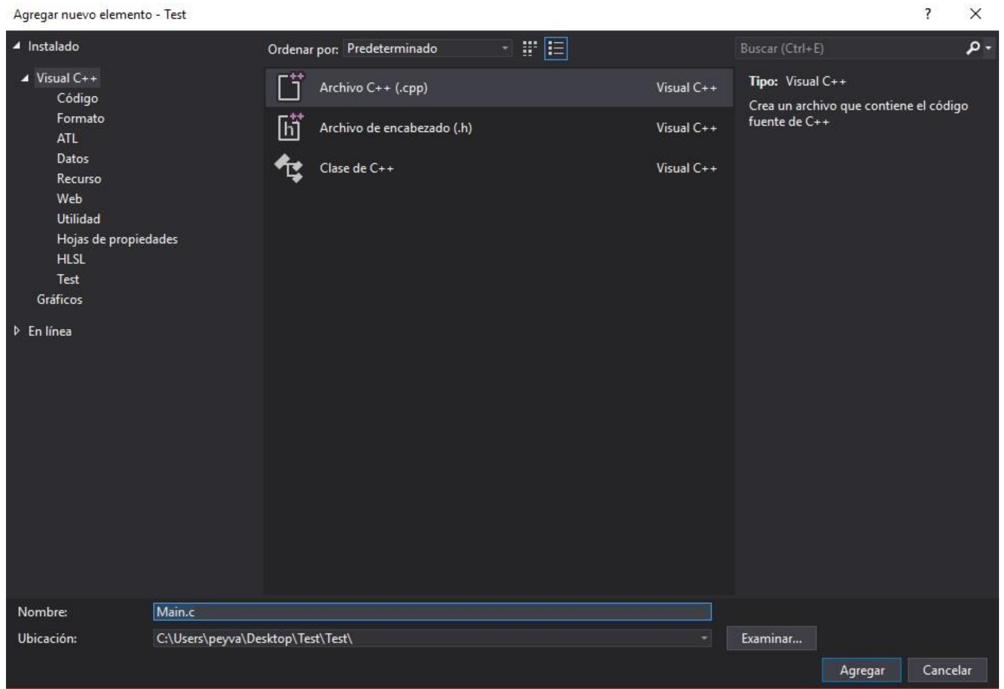


Figura 1-13. Añadir archivo ".c" Visual Studio.

3.4.3 Inclusión de las librerías Gurobi:

En este apartado se realizará la tarea principal para la inclusión de las librerías, ésta tendrá varias partes ya que dentro del mismo menú se modificarán varios parámetros. Todo se realizará dentro del menú de "Propiedades" del proyecto, desplegándose haciendo clic derecho en el nombre del proyecto y yendo a la última opción, como se ve en la siguiente figura.

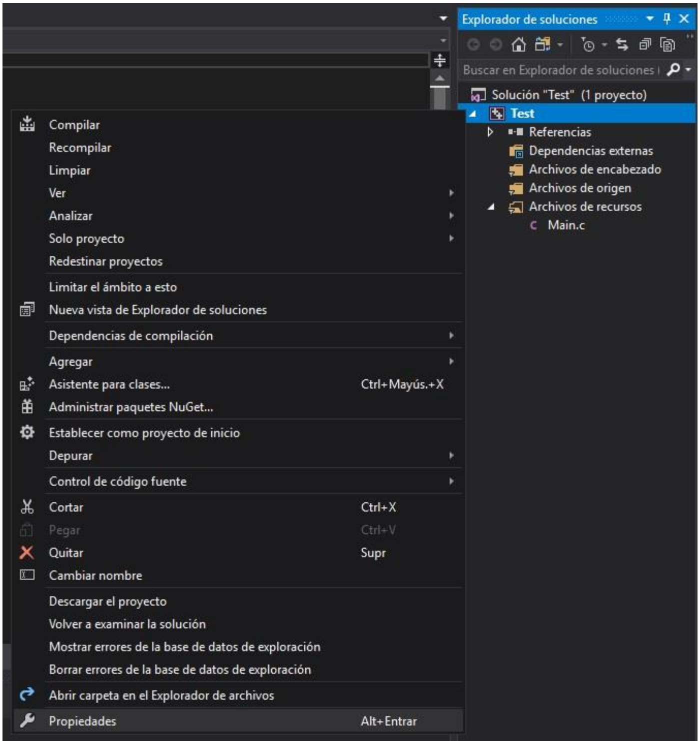


Figura 1-14. Propiedades de proyecto Visual Studio.

En orden descendiendo por las opciones que se abre en el desplegable que se muestra a la izquierda:

3.4.3.1 Cambiar la versión de Windows:

En el apartado que se abre por defecto al abrir las propiedades del proyecto, que deberá ser "General", se elegirá la opción "8.1", ésta puede no estar disponible por defecto como opción si se ha realizado una instalación de demasiado básica de Visual Studio, pero si se regresa al instalador de Visual Studio esta opción se puede agregar muy sencillamente.

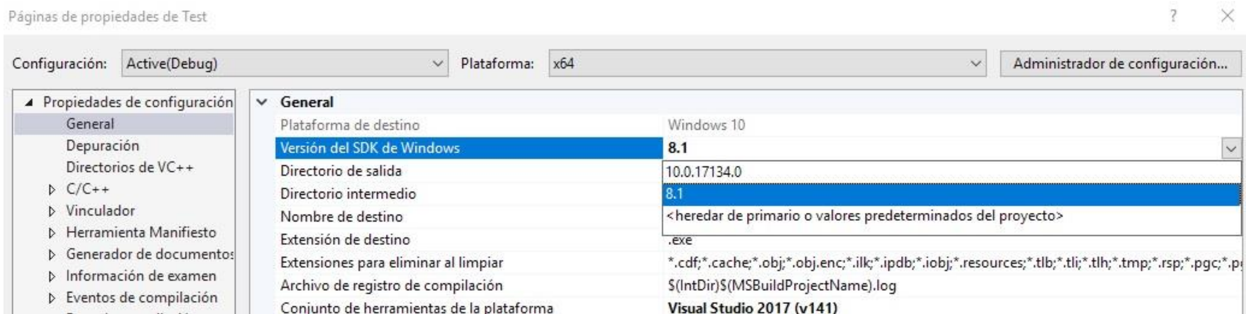


Figura 1-15. Propiedades proyecto versión de Windows.

3.4.3.2 Cambio de los directorios de VC++:

Se seleccionará en el desplegable de la izquierda el menú "Directorios de VC++", y aquí se realizarán dos modificaciones, la primera en la opción "Directorios de archivos de inclusión", se clicará en la opción de "Editar".

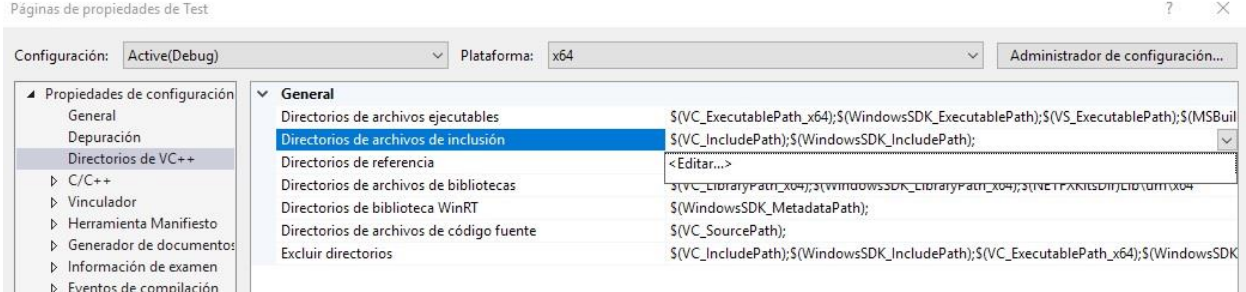


Figura 1-16. Propiedades proyecto directorios de archivos de inclusión.

Dentro de la ventana que aparecerá se copiará la ruta de la carpeta "include" de los archivos de la instalación de Gurobi.

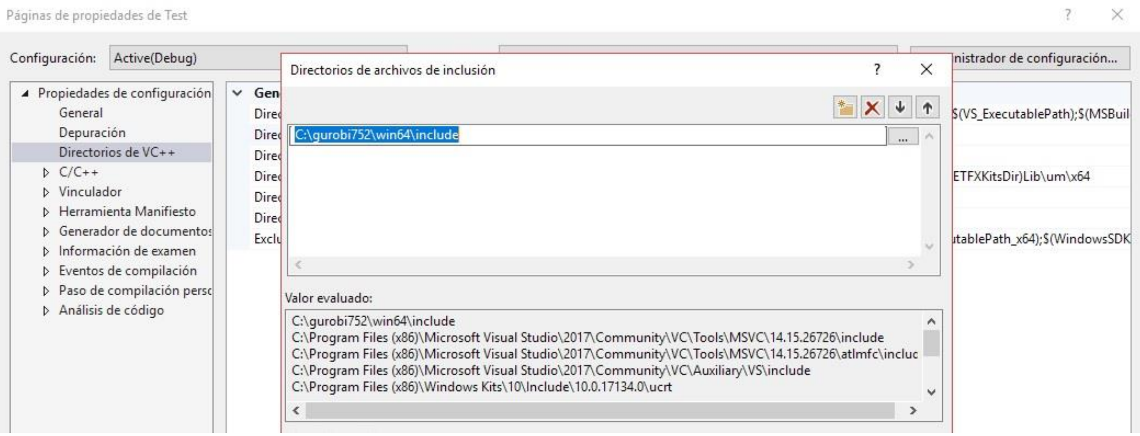


Figura 1-17. Agregar dirección de la carpeta “include”.

En segundo lugar se hará un cambio similar, pero copiando la ruta de la carpeta "lib", de los archivos de instalación de Gurobi, dentro de la opción "Directorios de archivos de bibliotecas", de la misma forma que antes seleccionando la opción "Editar". Para ambas modificaciones a continuación se verán las figuras indicativas del proceso en orden de explicación.

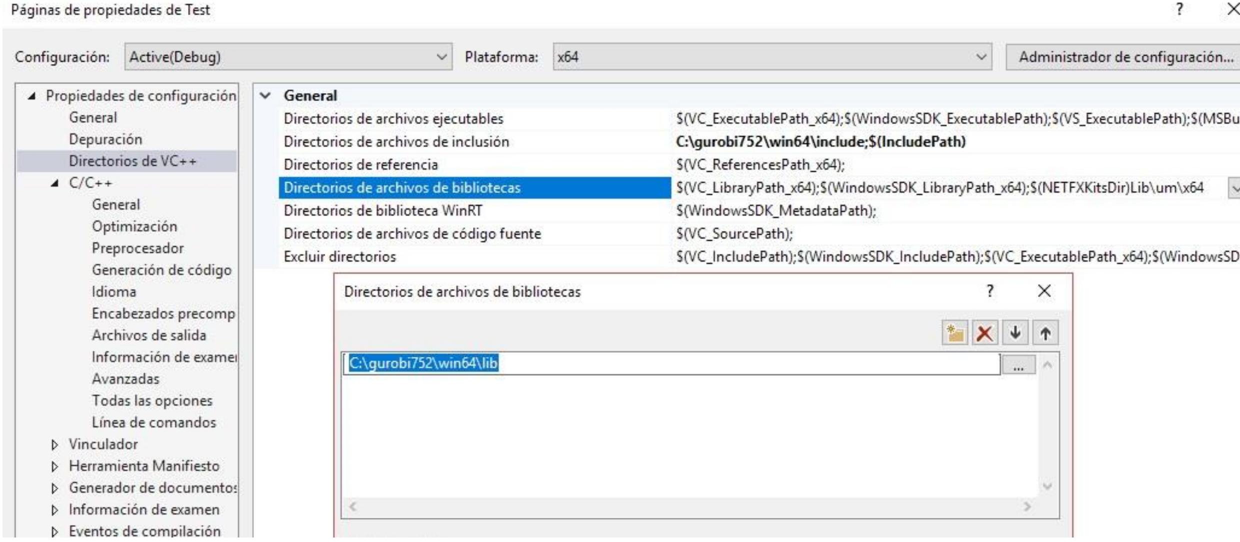


Figura 1-18. Agregar directorio de la carpeta “lib”.

3.4.3.3 Adición de directorios de inclusión:

Seleccionando el siguiente menú desplegable, "C/C++”, y el primer apartado, "General". Se modificará la primera opción "Directorios de inclusión adicionales", y de la misma forma que las anteriores modificaciones, a través de la opción "Editar" se añadirá la ruta de la carpeta "include" de los archivos de instalación de Gurobi.

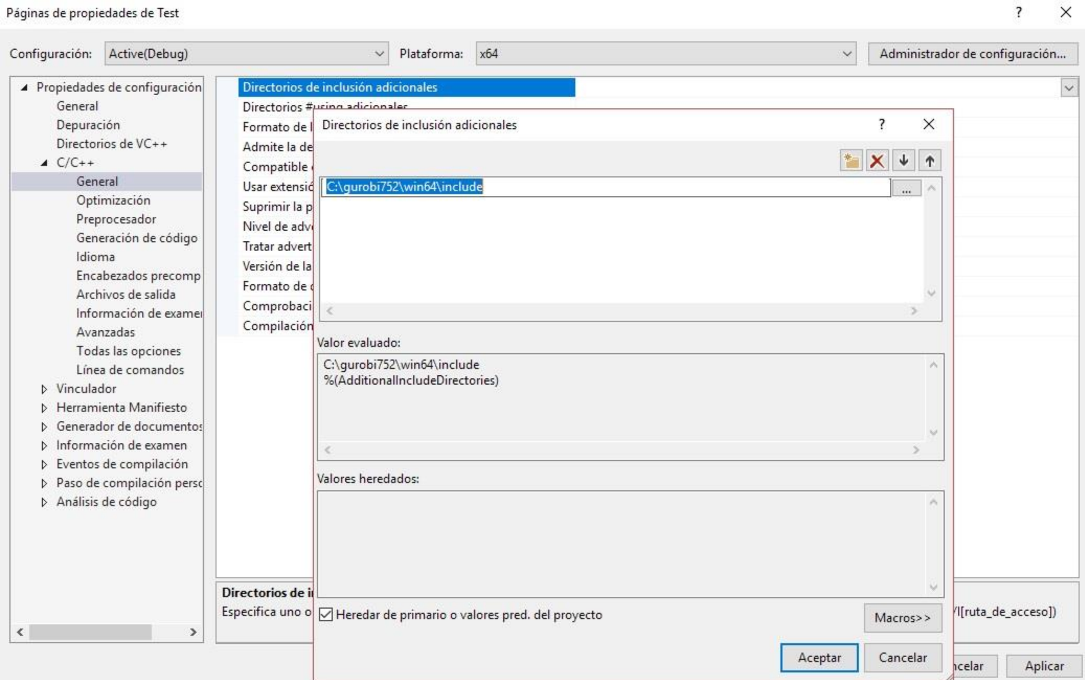


Figura 1-19. Propiedades proyecto, Directorios de inclusión adicionales.

3.4.3.4 Modificación de la entrada del vinculador:

Por último en cuanto a las modificaciones de las propiedades del proyecto, seleccionando la pestaña "Vinculador" y dentro de ésta la opción "Entrada", se añadirá en la primera opción, "Dependencias adicionales", el nombre del archivo de librería estática que se quiera vincular, en este caso específico se incluirá únicamente el archivo para las librerías en C, "*gurobi75.lib*".

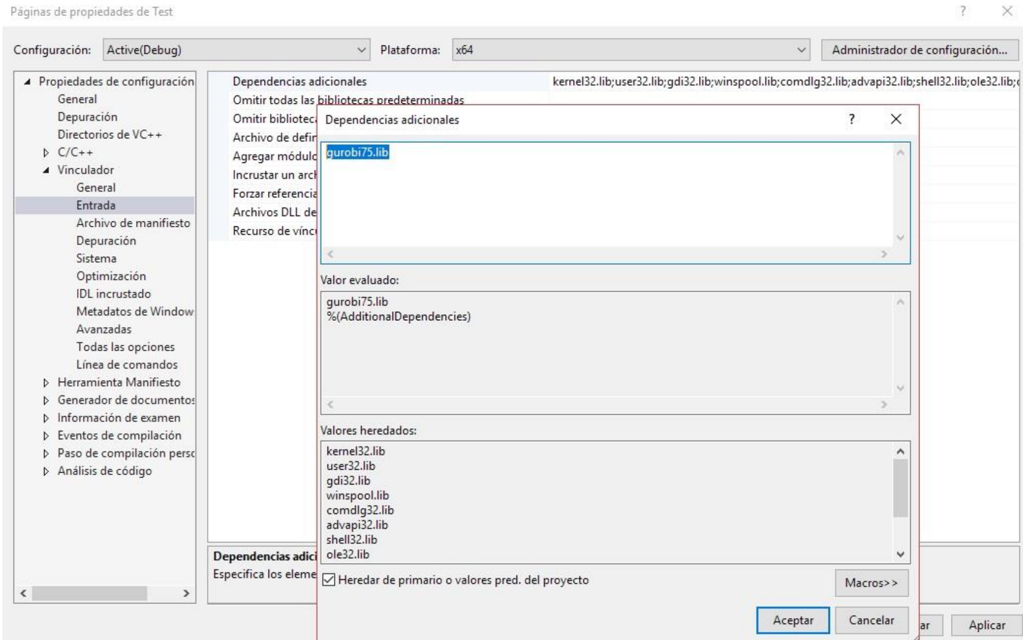


Figura 1-20. Agregar librería estática de Gurobi al Vinculador.

3.4.4 Cambio de las opciones del proyecto y del compilador a "x64":

Para la compilación y ejecución de todos los proyectos que se hagan con las librerías Gurobi es recomendable que se cambie esta opción para evitar que haya problemas de compatibilidades, en la siguiente figura se verán los dos cambios que se deben hacer,

- Primero, como todos los cambios que se han realizado en el apartado anterior, dentro de la configuración del propio proyecto, en el apartado "General", en la pestaña superior se modificará el apartado llamado "Plataforma" a x64.

- En segundo lugar, en la pantalla principal del proyecto, se modificará la opción del compilador, que viene por defecto "x86", a "x64".

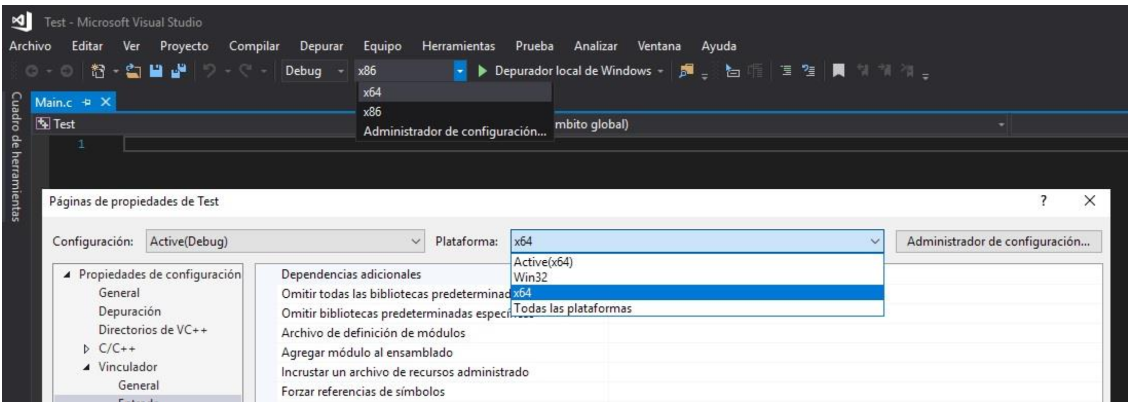


Figura 1-21. Modificación plataforma a “x64”.

Realmente lo que se acaba de realizar es algo redundante al añadir en varias opciones los mismos archivos de inclusión y librerías, pero de este modo se podrá garantizar que las librerías sean agregadas al proyecto y no den problemas de vinculación a la hora de compilar cualquier proyecto.

Además se proporcionarán en la documentación de este proyecto, un archivo comprimido con nombre "Plantilla Gurobi Final", la cual como se comentó anteriormente es una plantilla preparada con un código en C inicializado, las librerías de Gurobi incluidas en él, pero sólo es ejecutable si las instalación de las librerías se ha hecho en el disco duro principal del ordenador, del mismo modo que harían las librerías Gurobi al completar una instalación por defecto. Esta plantilla deberá copiarse en el directorio donde Visual Studio almacena las plantillas propias "Visual Studio 2017\My Exported Templates".

3.5 Análisis de códigos y funciones de mayor utilidad:

En este apartado se mostrarán y analizarán las funciones básicas e imprescindibles que serán necesarias a la hora de realizar cualquier tipo de optimización con Gurobi, además de funciones más específicas para la tarea que se tendrá que realizar en un futuro, resolver problemas cuadráticos. Se dividirá la explicación en varios apartados separando las diferentes utilidades de cada grupo de funciones:

3.5.1 Funciones de inicialización y función de error:

Si se optó por realizar la instalación de Gurobi y utilizar como primera toma de contacto algún ejemplo de optimización a través de la consola de comandos se sabrá de antemano la forma en la que trabaja Gurobi, en primer lugar es necesaria la creación de un entorno en el que se incluirá el modelo (siempre en primera instancia vacío), pero para realizar esto primero se debe conocer otra función, la función de error, ya que siempre que se llame a cualquier función de Gurobi esta va a estar igualada a una variable entera de error, la cual si es distinta de cero se obtendrá uno de los diferentes códigos de error.

3.5.1.1 Función de error "GRBgeterrormsg":

Esta realmente es una función que traduce un tipo de error y lo devuelve a través de un "printf", poniendo como variable que correspondiente a un string la función "GRBgeterrormsg", dentro del paréntesis se deberá especificar el nombre del entorno creado previamente y en el que se va a trabajando. A esta comprobación sólo se deberá acudir cuando el valor de la variable entera "error", que siempre ha de ser declarada al principio de un código, sea distinto de cero.

A partir de ahora se obviará la explicación de que todas las funciones llamadas que formen parte de Gurobi, estarán igualadas a la variable de tipo entero, error, y después de la llamada a la función se deberá llamar a la función de error para realizar la comprobación correspondiente.

3.5.1.2 Funciones para la creación de un entorno:

- "GRBenv":

Para poder llamar a esta función y crear un entorno en primer lugar se deberá declarar la variable que lo va a contener y esto se hará de la siguiente forma: "GRBenv *(variable del entorno) = NULL;".

- "GRBloadenv":

Posteriormente se llamará a la función "GRBloadenv ()" en la que dentro del paréntesis se pondrán los siguientes ocho parámetros en orden:

- Dirección de memoria del entorno: se escribirá de la siguiente manera "& (variable del entorno)", y aquí se incluirá la variable que justo se creó en el punto anterior.
- Nombre del archivo de registro del entorno (logfile): Nombre del archivo de texto con formato ".log", es decir, "(nombre del archivo).log".

3.5.1.3 Funciones para la creación de un modelo:

De igual manera que para la creación de un entorno, ahora se necesitará la declaración de una variable para el modelo, previamente a la declaración de la función para la creación del mismo:

- "GRBmodel":

La creación de la variable de modelo será, "GRBmodel *(variable del modelo) = NULL".

- "GRBnewmodel":

En este caso la función para la creación del modelo tendrá bastantes más argumentos que la anterior, pero por defecto lo que se hará siempre será inicializar el entorno vacío y con todos los parámetros por defecto (que en el caso de las librerías Gurobi éstos serán así cuando sean definidos como "NULL"). La función será "GRBnewmodel ()" tiene ocho parámetros:

- Variable correspondiente al entorno previamente creado.
- Dirección de memoria de la variable del modelo: Declarada previamente se llamará a la dirección de memoria de la siguiente forma, "& (variable del modelo)".
- Nombre del modelo: Puesto entre comillas, nombre que se le dará al modelo el cual puede ser distinto a la variable asignada al mismo.
- Número de variables del modelo: Número entero que indique el número de variables del modelo, éste parámetro por defecto se inicializará en cero y posteriormente se añadirán las variables pertinentes.

Los últimos cinco parámetros de esta función los se inicializarán con el valor de "NULL", asignando a cada uno el valor por defecto correspondiente.

- Coeficientes de las variables: No se han inicializado variables así que el valor por defecto será cero.
- Límite inferior: Límites inferiores para las variables, será cero por defecto.
- Límite superior: Límites superiores para las variables, será infinito por defecto.
- Tipo de las variables: Se podrá declarar en un vector el tipo de variables que se han inicializado; continuas, binarias, enteras, semicontinuas o semienteras. Por defecto se inicializarán como variables continuas (lo que será lo más habitual).
- Nombres de variables: Vector con todos los nombres que se le quieran asignar a las variables inicializadas, por defecto se le pondrán nombres de manera automática creados por Gurobi (se elegirá esta opción generalmente)

3.5.2 Adición de términos a la función objetivo:

En este apartado se deberán diferenciar tres tipos de términos diferentes que será interesante añadir a la función a optimizar:

3.5.2.1 Adición de términos lineales "GRBaddvars":

A través de una única función se añadirán el número de variables lineales y un vector con los valores de cada una de las variables en orden. Esta función tiene once parámetros de los cuales la mayoría se inicializarán por defecto, algunos serán parámetros repetidos ya vistos en la anterior declaración de la creación del modelo:

- Variable referente al modelo. En prácticamente la totalidad de las funciones que se utilicen a través de las librerías Gurobi en primera instancia se deberá hacer referencia al entorno, al modelo o a ambos, dependiendo los parámetros que requiera la función.
- Número de nuevas variables que añadir: Justamente aquí se deberá introducir el número entero de variables que se quieran añadir al modelo.
- Número total de coeficientes no nulos en las nuevas columnas: En todos los casos este valor será cero ya que los coeficientes se agregarán en otro de los parámetros siguientes.

Los siguientes tres parámetros no serán tomados en cuenta en ninguna de las optimizaciones que se realicen, ya que se refieren a restricciones las cuales serán añadidas posteriormente, por ello se pasará como parámetro "NULL".

- Coeficientes para las nuevas variables: Contenidas en un vector con valores tipo double con el coeficiente que acompaña a cada una de las variables en orden, siendo la primera variable a la cual le acompaña el coeficiente 0, en el caso de haber optado por aceptar la nomenclatura por defecto de Gurobi.

Por último quedan las cuatro variables que se repetirán de la inicialización del modelo realizada anteriormente, y serán inicializadas, a menos que sea necesario, en sus variables por defecto con el parámetro "NULL".

- Límite inferior: Límites inferiores para las variables, será cero por defecto.
- Límite superior: Límites superiores para las variables, será infinito por defecto.
- Tipo de las variables: Se podrá declarar en un vector el tipo de variables que se han inicializado; continuas, binarias, enteras, semicontinuas o semienteras. Por defecto se inicializarán como variables continuas (lo que será lo más habitual).
- Nombres de variables: Vector con todos los nombres que se le quieran asignar a las variables inicializadas, por defecto se le pondrán nombres de manera automática creados por Gurobi (se elegirá esta opción generalmente).
- Vector de valores para cada una de las variables cuadráticas definidas: Vector de valores tipo double correspondiente a cada una de las variables cuadráticas en el orden definido previamente.

3.5.2.2 Adición de términos cuadráticos "GRBaddqpterms":

Función que servirá para añadir variables cuadráticas al modelo previamente creado, las variables cuadráticas que se tendrá a disposición crear a través de esta función sólo podrán ser combinación de las variables lineales definidas. Por defecto las variables lineales tendrán índices definidos por Gurobi, cosa que se recomienda, que irán en orden desde la variable cero hasta el número de variables máximo menos uno. A la hora de darle parámetros a la función se tiene que conocer previamente cómo funciona, ya que se le tendrá que pasar los índices de las variables cuadráticas, éstos divididos en dos vectores que al juntarse darán la combinatoria del número máximo de variables sin importar el orden (ya que el orden de la multiplicación de dos variables diferentes dará como resultado la misma variable cuadrática):

- Coeficientes para las nuevas variables: Contenidas en un vector con valores tipo double con el coeficiente que acompaña a cada una de las variables en orden, siendo la primera variable a la cual le acompaña el coeficiente 0, en el caso de haber optado por aceptar la nomenclatura por defecto de Gurobi.
- Número de variables cuadráticas total.
- Primer vector de índices de las variables cuadráticas "qfil": Vector de variables enteras cuyos valores irán desde cero hasta el número de variables lineales menos uno y tendrán un orden predefinido, por ejemplo, si se define un modelo con tres variables el vector tendrá seis variables cuadráticas y será: "qfil = [0,0,0,1,1,2]".
- Segundo vector de índices de las variables cuadráticas "qcol": Al igual que en el caso anterior es un vector de enteros con índices como máximo el número de variables lineales menos uno con un orden específico que al combinarlo con el vector anterior dé como resultado los índices de las variables cuadráticas, y será "qcol = [0,1,2,1,2,2]".

3.5.2.3 Adición del término independiente:

Este apartado no es abordado dentro de ninguno de los ejemplos de optimización de Gurobi, pero será de vital importancia a la hora de realizar la optimización de la función objetivo de un control predictivo, ya que siempre habrá un término que será independiente. Esta función no es como todas las anteriores, ésta se basa únicamente en la modificación de uno de los "Atributos" [10] que define Gurobi.

Dependiendo del atributo que se vaya a modificar se debe especificar en la llamada:

- Operación a realizar: dentro del trato de atributos se pueden realizar dos tareas, la toma de valores del modelo añadiendo al principio de la función "get", o la declaración de valores añadiendo al principio de la función "set".
- Tipo de variable: En el caso del término independiente será una variable double, para Gurobi se utilizará la abreviación "dbl".

Por ello la llamada a la función que se deberá hacer será, "GRBsetdblattr ()", y dentro de los parámetros de ésta función:

- Variable correspondiente al modelo al que se pretenda modificar este atributo.
- Nombre específico del atributo: En este caso específico, "GRB_DBL_ATTR_OBJCON", se debe elegir el tipo de variable que se debe pasar, definir que va a ser un "Atributo" lo que se modifique y por último el nombre específico del atributo.
- Valor/Variable tipo double que se quiere agregar al atributo que se está modificando.

3.5.3 Modificación del sentido de la optimización "MODELSENSE":

Al igual que el caso anterior con el término independiente, ahora se modificará otro de los "Atributos" definidos por Gurobi, que será el sentido de la optimización. En este proyecto generalmente lo que se necesitará (y en torno a lo que se basa el control predictivo) será la minimización de una función objetivo, por esto siempre se elegirá este sentido para la optimización, que será el sentido que Gurobi elige por defecto, pero si se realiza cualquier tipo de problema de optimización esta opción es importante.

3.5.3.1 "GRBsetintattr ()":

Dentro de los atributos se tendrá algo muy similar a lo visto anteriormente:

- Variable correspondiente al modelo.
- Nombre específico del atributo: En este caso con lo definido anteriormente será, "GRB_INT_ATTR_MODELSENSE".
- Elección del sentido de optimización: Elegir entre minimización o maximización de la función objetivo, esto se hace a través de las funciones de Gurobi, "GRB_MINIMIZE" y GRB_MAXIMIZE" respectivamente.

3.5.4 Adición de restricciones al modelo:

De la misma forma que en el caso de la adición de términos en la función objetivo se puede dividir la adición de restricciones al modelo en dos tipos, lineales y cuadráticas, aunque realmente las que se aplicarán en este trabajo principalmente serán lineales, se explicarán ambas al ser similares entre ellas y tener bases exactamente en el caso de las cuadráticas.

3.5.4.1 Adición de restricciones lineales "GRBaddconstr ():

Función que servirá para añadir restricciones al modelo de forma individual, se deberán especificar dentro de los parámetros en orden:

- Variable del modelo al que añadir la restricción.
- Número entero de variables que contiene la restricción.
- Vector con el índice de las variables: Básicamente es un vector de valores enteros que representará al índice de las variables que se quiera incluir en la restricción, si se quieren incluir todas las variables el vector irá desde cero hasta el número de variables menos uno.
- Vector de valores correspondiente a cada una de las variables de la restricción: Vector compuesto por valores de tipo double correspondiente a cada una de las variables. De forma genérica se podría inicializar todas las variables por defecto dentro de la declaración de la restricción, y asignar valores nulos a las que no existan en las correspondientes restricciones.
- Sentido de la igualdad de la restricción: Símbolo de la igualdad de la restricción, el cual puede ser, menor o igual, igual, o mayor o igual, traducido a parámetros de Gurobi respectivamente,"GRB_LESS_EQUAL","GRB_EQUAL","GRB_GREATER_EQUAL" .
- Valor del lado derecho de la igualdad de la restricción: Valor de tipo double al que está igualada el lado derecho de la ecuación de la restricción.
- Nombre que se le asigne a la restricción: Generalmente se tomará el valor por defecto que asigna Gurobi al darle un parámetro "NULL", pero se podría darle el nombre que se quiera.

3.5.4.2 Adición de restricciones cuadráticas al modelo "GRBaddqconstr ()":

Esta función se basa en lo mismo que se explicó en el apartado referente a la adición de variables cuadráticas, se tienen que crear de igual manera dos vectores que al combinarse den como resultado los índices de las variables cuadráticas que participan en la restricción, aunque de igual manera que en el punto anterior, se podrá utilizar de manera convencional los mismos vector que se crearon para la definición, inicializando todas las variables cuadráticas posibles y asignando valores nulos a las que no intervengan en la restricción. Al margen de lo mencionado, esta función tiene una característica adicional, ya que una restricción cuadrática también puede tener términos lineales, por este hecho Gurobi también permite añadir términos lineales junto a los cuadráticos, aunque en caso de no ser necesarios siempre se podrán declarar como "NULL", en los parámetros de llamada se tendrá:

- Variable referente al modelo.
- Número entero de variables lineales que contiene la restricción.
- Vector de índices de tipo entero para las variables lineales.
- Vector de valores de tipo double para cada una de las variables lineales.
- Número de variables cuadráticas.
- Primer vector de índices de las variables cuadráticas "qfil".
- Segundo vector de índices de las variables cuadráticas "qcol".

- Vector de valores correspondiente a cada una de las variables cuadráticas de la restricción. Si se utiliza la convención mencionada en la que se inicializarán por defecto todas las variables cuadráticas posibles, en este vector de valores de tipo double se le asignarán valores nulos a las variables que no existan dentro de la restricción.
- Sentido de la igualdad de la restricción, designado del mismo modo que el mencionado con anterioridad.
- Valor del lado derecho de la igualdad de la restricción.
- Nombre que se le asigne a la restricción: Generalmente se tomará el valor por defecto que asigna Gurobi al darle un parámetro "NULL", pero se podría darle el nombre que se quiera.

3.5.5 Optimización y escritura del modelo:

En este apartado se abordarán dos funciones relativamente simples, la función que realiza la optimización del modelo, la cual será llamada cuando se tenga la función objetivo definida al completo, y la función para la escritura del modelo, ésta se realizará en un archivo de texto y dejará reflejado las variables, sus tipos, la función objetivo y las restricciones.

3.5.5.1 Función de optimización "GRBoptimize ()":

Función que realizará el proceso de optimización de la, ya definida, función objetivo con las restricciones añadidas anteriormente, únicamente habría que añadir como parámetro la variable del modelo que se haya creado para la optimización.

3.5.5.2 Función para escritura de modelo "GRBwrite ()":

Función que escribirá los datos más relevantes del modelo que se optimice dentro de un archivo de texto, este archivo tendrá formato ".lp", como parámetros a ésta función se le tendrá que proporcionar:

- Variable que contenga el modelo que se quiera guardar.
- Nombre del archivo en el que se guardará el modelo: Entre comillas se le dará nombre al documento con extensión ".lp" para que guarde el modelo de una optimización y pueda ser utilizado posteriormente si se quiere.

3.5.6 Captura de valores:

En este apartado se verán dos funciones que serán relevantes para el objetivo final de este trabajo, en primer lugar se presentará una función para comprobar el estado de la optimización y comprobar si se ha encontrado un resultado o es irrealizable, y en segundo lugar una función que permita extraer los resultados con los que ha dado el optimizador para cada una de las variables. Como se mencionó antes, en este caso también se tratará con funciones que van a extraer "Atributos" definidos por Gurobi dentro del modelo, por ello seguirán la misma estructura que los vistos anteriormente.

3.5.6.1 Obtención del estado de la optimización "STATUS":

El Atributo que me permite extraer el estado de una optimización ya realizada será esta vez de tipo entero, se deberá inicializar una variable para guardar el atributo para su posterior comprobación. La llamada a la función entonces será, "GRBgetintattr ()", y como parámetros se le proporcionará:

- Variable referente al modelo.
- Nombre específico del atributo: Con lo definido anteriormente se puede deducir que será, "GRB_INT_ATTR_STATUS".
- Dirección de memoria de la variable donde se va a guardar el estado óptimo.

3.5.6.2 Obtención de los resultados de las variables tras la optimización:

Por último en la obtención de resultados se va a tratar la extracción de los valores resultado de las variables de la función objetivo, generalmente se tendrán varias variables y éstas se agruparán dentro de un vector de variables tipo double, por ello a la hora de la declaración de la función que extraiga estos valores, además de lo ya comentado, se tendrá que especificar al final que se está extrayendo un "array", por ella la función quedará de la siguiente manera, "GRBgetdblattrarray ()", donde los parámetros que deben ser proporcionados serán:

- Variable referente al modelo.
- Nombre del atributo: Al contrario que justo antes donde se debía especificar que la función devolverá un vector de variables tipo double, ahora no es necesario que se especifique que será un vector, el nombre del atributo específico será, "GRB_DBL_ATTR_X".
- Valor inicial del vector: Por defecto se tomará este valor como cero.
- Número de variables total de las que se quieran obtener el resultado de la optimización.
- Variable donde se guardará el vector solución.

3.5.7 Liberación del sistema:

Posteriormente de realizar todo el trabajo de optimización dentro del modelo y el entorno que se crearon al principio de cualquier proyecto en el que se utilizarán las librerías Gurobi (y de manera similar a lo que se haría al utilizar memoria dinámica), se deberá liberar los modelos que se hayan inicializado, si existen varios, y el entorno que los contiene. Las funciones de liberación de modelos y entorno son las únicas en las que no es necesario realizar comprobación de errores, pero sí existe una única restricción a la hora de utilizarse, siempre se debe liberar previamente los modelos al entorno que los contenía.

3.5.7.1 Liberación de modelos "GRBfreemodel ()":

Como parámetro únicamente es necesario pasar el nombre de las variables de los modelos que se quieran liberar, eso sí, únicamente se puede liberar un modelo por cada llamada a la función.

3.5.7.2 Liberación de entornos "GRBfreeenv ()":

Exactamente igual que en la función anterior, pero ahora con la variable que se haya definido para el entorno, se deberá pasar como parámetro el nombre de esta variable para liberar el entorno.

Todas las funciones que se han comentado están referenciadas en el documento "*refman.pdf*" [9] por si se quiere buscar más información o la información original explicada por los desarrolladores.

3.6 Ejemplo de resolución de un QP con las librerías de optimización Gurobi:

Después de haber analizado todas las funciones básicas, que deberán incluirse en la mayoría de códigos que se realicen en los que exista un proceso de optimización, se analizará un código de ejemplo de resolución de un problema de programación cuadrática (QP, Quadratic Programming) con restricciones lineales. El código completo está incluido en el *Anexo A*, además de incluirse con este trabajo los archivos ejecutables del proyecto completo para Visual Studio con el nombre, "QPejemplo".

Los datos del sistema se pueden observar a continuación, donde se tendrán únicamente dos variables lineales y

sus tres combinaciones posibles de variables cuadráticas además de varias restricciones lineales.

```
/*
SIENDO x,y:

Minimizar:        0.5x^2   +    y^2   -   x*y   -  2x   -   6y

Sujeto a:            x     +    y         <=   2
                    -x     +   2y         <=   2
                    2x     +   y          <=   3
                     x                    >=   0
                     y                    >=   0

*/
```

Figura 1-22. Ejemplo ecuaciones de un QP.

Como primer paso, siempre imprescindible dentro de cualquier código, más aún en el caso que se trata utilizando unas librerías externas, se deben incluir todas las cabeceras para las librerías que se van a requerir para que el programa sea ejecutable. En el caso de la programación en C las más típicas suelen ser "*stdio.h*", "*stdlib.h*" o "*math.h*", en caso de que se trabajen con strings o cadenas de caracteres puede ser de utilidad la librería "*string.h*".

```
#include <stdio.h>
#include <stdlib.h>
#include <gurobi_c.h>
#include <math.h>
```

Figura 1-23. Inclusión de librerías en C.

3.6.1 Función de error:

Previamente a cualquier punto que vaya a ser analizado, en el caso de un código que incluya la utilización de librerías de optimización Gurobi, lo primero sería analizar la estructura de la función de error a la que se debe llamar después de la utilización de cualquiera de las demás funciones de Gurobi que se utilicen, exceptuando las declaraciones de variables exclusivas de Gurobi (como pueden ser las referentes a la declaración de entorno y modelo).

La función, estrictamente hablando, al ser llamarla a través del código sería únicamente "GRBgeterrormsg", la cual se refiere a un *string* o cadena de caracteres donde se indica el tipo de error que se ha producido. Como condición para comprobar que ha habido cualquier tipo de error en cada función, siempre se igualará el uso de una función a una variable entera creada previamente, la cual, al dar un valor no nulo a posteriori de la ejecución de una función, se remitirá a la parte del código en el que se realiza la, propiamente dicha, comprobación de errores. En los propios códigos de ejemplo de Gurobi se representa de una forma similar a la que se puede ver a continuación:

```
if (error != 0) {

QUIT://COMPROBACION DE ERRORES\\

    printf("ERROR = %s\n", GRBgeterrormsg(env));
    printf("Se detecto un error y se procede a cerrar la aplicacion\n");
    system("pause");
    exit(1);

}
```

Figura 1-24. Función de error de Gurobi.

Después de cada función se ejecutará una estructura *if* que comprobará si el valor entero referente al error es un valor no nulo, y si esto es así la ejecución se desplazará a la línea donde se ha escrito *QUIT*, se detendrá la ejecución y se pasará por pantalla el tipo de error que ha ocurrido durante la ejecución.

Posteriormente, por modularidad del código, esta forma de comprobación de errores que se acaba de explicar será reformada en una propia función dentro del código a la se llamará después de la ejecución de cada una de las otras funciones de Gurobi a las que se requiera y, en vez de desplazarse en el propio código para realizar la comprobación, se entrará en una función como se ve a continuación:

```
void WINAPI ErrorComp(GRBenv *env) {

    printf("ERROR = %s\n", GRBgeterrormsg(env));

    system("pause");
    exit(1);
}
```

Figura 1-25. Función externa para comprobación de errores de Gurobi.

3.6.2 Declaración de variables:

Posteriormente a la declaración de archivos de inclusión vendrá la función principal del código, la cual típicamente suele ser de tipo entero y llamarse *main*, con la declaración de variables que se van a utilizar dentro del código. A continuación se puede observar un ejemplo de las típicas declaraciones de variables ante la resolución de un QP (Figura *1-25*):

```
int error = 0;
int estadoptimo;

int inc[2];
double val[2];

int qfil[3];
int qcol[3];
double qval[3];

GRBenv *env = NULL;
GRBmodel *model = NULL;
```

Figura 1-26. Declaración de variables para Gurobi.

En orden de aparición, la primera, como ya se comentó en el apartado anterior, es el entero para la comprobación de errores de las funciones, seguido de las variables que se quieran extraer después de la realización de la optimización (en este caso solo está el estado de la optimización, pero se pueden añadir tantos como sean necesarios y Gurobi permita extraer). Los dos siguientes grupos serán los índices y valores asociados a, en primer lugar, las variables lineales de la función objetivo y en segundo lugar las relacionadas con las variables cuadráticas de la misma.

Por último, como primer uso de comandos exclusivos de Gurobi, la inicialización de variables esenciales para la optimización, inicializando en primer lugar el entorno y en segundo lugar el modelo.

3.6.3 Creación de entorno y modelo:

Posteriormente a la declaración de las variables para el entorno y el modelo Gurobi, se deben crear el propio entorno y después de esto, crear el modelo vinculado al entorno ya existente. Se utilizarán las funciones de las cuales ya se explicó el funcionamiento en el apartado 3.4.1.2 y apartado 3.4.1.3, respectivamente referidos al entorno y al modelo. Quedando las funciones de la siguiente manera declaradas:

```
GRBenv *env = NULL;
GRBmodel *model = NULL;

//CREACION DEL ENTORNO\\

error = GRBloadenv(&env, "QPmatlab.log");
if (error != 0) { goto QUIT; }

//CREACION DE MODELO (siempre vacío)\\

error = GRBnewmodel(env, &model, "QPmatlab", 0, NULL, NULL, NULL, NULL, NULL);
if (error != 0) { goto QUIT; }
```

Figura 1-27. Creación del modelo y entorno en Gurobi.

Lo más importante a destacar además de lo mencionado en todos los apartados anteriores sería en primer lugar la declaración del entorno, donde se debe dar nombre al archivo ".log" donde quedarán reflejadas todas las acciones realizadas por el optimizador en las llamadas a funciones que vendrán a continuación y en la inicialización del propio entorno. La creación de entorno se realizará siempre sobre uno vacío, sin añadir ninguna variable, ya que se añadirán con una función en el apartado que viene a continuación.

Por último después de la declaración de cada una de las funciones viene la llamada a la función de comprobación de errores, bajo la condición de que la variable de error sea un valor no nulo.

3.6.4 Adición de variables al modelo:

Al haberse inicializado la variable y creado el propio modelo, el paso inmediatamente siguiente que debe realizarse es la adición de variables, ya que el modelo está en principio vacío. Dependiendo de las características del modelo se pueden tener diferentes tipos de variables, en este caso se podrá elegir si añadir términos lineales, términos cuadráticos o ambos simultáneamente.

A la vista del modelo del ejemplo se deberán añadir ambos tipos de variables, en la imagen a continuación se puede observar las variables necesarias y las funciones utilizadas para este proceso:

```
//ADICION DE VARIABLES\\

//Aqui añadimos la parte lineal de la funcion en la CREACION DE VARIABLES
double obj[] = { -2,-6 };

error = GRBaddvars(model, 2, 0, NULL, NULL, NULL, obj, NULL, NULL, NULL, NULL);
if (error != 0) { goto QUIT; }

//Ahora añadimos los términos cuadráticos a parte
qcol[0] = 0; qfil[0] = 0; qval[0] = 0.5;
qcol[1] = 1; qfil[1] = 1; qval[1] = 1;
qcol[2] = 0; qfil[2] = 1; qval[2] = -1;

error = GRBaddqpterms(model, 3, qfil, qcol, qval);
if (error != 0) { goto QUIT; }
```

Figura 1-28. Adición de variables lineales y cuadráticas en Gurobi.

En las primeras tres líneas de código no comentadas se añaden las variables lineales del modelo, que en este caso son únicamente dos, previamente inicializando un vector con los valores (siempre de tipo double) que acompañan a cada una de las variables. El desarrollo sobre la utilización de los demás términos, que se incluyen con valores por defecto (al tener una entrada de tipo "NULL"), fueron explicados dentro del apartado *3.4.2.1*.

El resto del código tiene como objetivo el añadido de las variables cuadráticas, las cuales serán la combinatoria de las variables lineales que existan sin importar el orden (por lo que la variable cuadrática que sea de la forma "X*Y", será considerada la misma que la variable "Y*X"). En el caso que se está tratando al tener dos variables lineales, existirán como máximo tres variables cuadráticas, las cuales están totalmente definidas a través de los tres vectores, "qcol", "qfil", "qval", en los cuales, repitiendo lo que ya se ha explicado en el apartado *3.4.2.2.*, juntando los valores de las componentes homólogas de cada uno de los dos primeros vectores, se almacenan los índices de las variables cuadráticas (en este caso la variable *número 0* al cuadrado, la *número 1* al cuadrado y el producto de la variable *número 0* y la *número 1*). El tercer vector le asigna valores a cada una de las variables cuadráticas, en el orden en el que se han añadido.

Introduciendo estos términos a través de las dos funciones que se conocen de adición de términos completaría la declaración de la función objetivo del modelo.

3.6.5 Cambio del sentido de la optimización:

Dentro de las librerías de optimización Gurobi se ofrece la posibilidad de resolver multitud variedad de problemas de optimización, entre ellas lo más básico podría ser la elección del sentido de la optimización del modelo, si se quiere una minimización o una maximización del mismo. Si esta función no se llama en ningún momento en el código el modelo tomará por defecto la opción de minimización, pero en caso de que se quiera modificar, a continuación se puede ver cómo sería la llamada a esta función y la modificación del atributo dentro de Gurobi:

```
//ELEGIMOS EL SENTIDO DE LA OPTIMIZACION (depues de la declaración de la funcion objetivo)
error = GRBsetintattr(model, GRB_INT_ATTR_MODELSENSE, GRB_MINIMIZE);
if (error != 0) { goto QUIT; }
```

Figura 1-29. Función de cambio de sentido de optimización.

Como posteriormente se verá, a la hora de resolver cualquier QP dentro de un GPC siempre será necesario tomar la opción de minimización, ya que la base teórica es la minimización de una función objetivo, por lo que se puede prescindir ésta, aunque por seguridad y claridad del código se utilizará igualmente.

3.6.6 Adición de restricciones:

Posteriormente a la declaración de variables del modelo, como proceso adicional de adición de datos del modelo, en caso de que existe se deberá añadir las restricciones que posea el modelo. Pudiendo ser, de igual manera que sus variables, lineales o cuadráticas. En el caso que se está tratando las restricciones deben ser inexistentes o como máximo lineales, ya que si además existiesen restricciones cuadráticas no se estaría tratando un QP, si no que sería un problema más complicado, QCQP (Quadratic Constrained Quadratic Programming), temática que no se abordará en este proyecto.

```
//ADICION DE RESTRICCIONES\\
//Solo tenemos restricciones lineales

inc[0] = 0;inc[1] = 1;//Todas las restriciones hasta las dos ultimas utilizan las dos variables

val[0] = 1;val[1] = 1;
error = GRBaddconstr(model, 2, inc, val, GRB_LESS_EQUAL, 2, "lc0");
if (error != 0) { goto QUIT; }

val[0] = -1;val[1] = 2;
error = GRBaddconstr(model, 2, inc, val, GRB_LESS_EQUAL, 2, "lc1");
if (error != 0) { goto QUIT; }

val[0] = 2;val[1] = 1;
error = GRBaddconstr(model, 2, inc, val, GRB_LESS_EQUAL, 3, "lc2");
if (error != 0) { goto QUIT; }



inc[0] = 0; val[0] = 1;
error = GRBaddconstr(model, 1, inc, val, GRB_GREATER_EQUAL, 0, "lc3");
if (error != 0) { goto QUIT; }

inc[0] = 1; val[0] = 1;
error = GRBaddconstr(model, 1, inc, val, GRB_GREATER_EQUAL, 0, "lc4");
if (error != 0) { goto QUIT; }
```

Figura 1-30. Adición de restricciones en Gurobi.

La declaración de restricciones lineales se ha realizado en dos bloques, las primeras tres restricciones en las que las dos variables lineales están acompañadas por coeficientes no nulos y las restricciones que dependen únicamente de una de las dos variables.

En primer lugar antes de la llamada a la función de Gurobi que añade las restricciones de forma individual se debe inicializar un vector con los índices de las variables que contiene, en formato de entero “int”, en las primeras tres serán ambas por lo se inicializará el vector “inc” con los valores “0” y “1”, y en las siguientes dos serán para la primera el valor “0” y para la segunda sólo el valor “1”, correspondientes a cada una de las variables lineales. Además antes de la llamada de la función para cada una de las restricciones se irá sobrescribiendo en un vector de tipo double, los dos coeficientes que acompañan a cada variable dentro de cada una de las restricciones. Y de este modo como se ha mostrado en la imagen se van añadiendo todas las restricciones que sean necesarias al modelo.

3.6.7 Optimización y escritura:

Ya realizadas todas las operaciones de inicialización y adición de componentes al modelo se puede tomar la decisión, indistintamente en cuestión del orden, de realizar la escritura del modelo en un archivo o la optimización del mismo. La escritura del modelo no es necesaria para todo el proceso de optimización pero permite comprobar si éste se ha implementado correctamente a través de las funciones de Gurobi. El código de llamada a ambas funciones será el siguiente:

```
//OPTIMIZACION DEL MODELO\\

error = GRBoptimize(model);
if (error != 0) { goto QUIT; }

printf("\n\nOPTIMIZACION COMPLETADA\n\n");



//ESCRITURA DEL MODELO COMPLETO\\

error = GRBwrite(model, "QPmatlab.lp");
if (error != 0) { goto QUIT; }
```

Figura 1-31. Funciones de optimización y escritura de Gurobi.

En cuanto a la llamada de las funciones no existe demasiada complejidad, únicamente destacar sobre la función de escritura, donde se debe incluir el nombre y el formato del archivo de salida para la escritura del modelo, en el cual se elegirá un archivo tipo LP que tendrá representadas las variables de la función objetivo con todas las restricciones añadidas, de la forma que se puede ver a continuación:

```
\ Model QPmatlab
\ LP format - for model browsing. Use MPS format to capture full model detail.
Minimize
  - 2 C0 - 6 C1 + 9 Constant + [ C0 ^2 - 2 C0 * C1 + 2 C1 ^2 ] / 2
Subject To
 lc0: C0 + C1 <= 2
 lc1: - C0 + 2 C1 <= 2
 lc2: 2 C0 + C1 <= 3
 lc3: C0 >= 0
 lc4: C1 >= 0
Bounds
 Constant = 1
End
```

Figura 1-32. Formato de un archivo LP para escritura del modelo en Gurobi.

Por parte de la función de optimización la salida que se obtendrá será la misma a través de dos canales diferentes, el primero será en la propia consola de comandos que se debería abrir en la propia ejecución del código, y además dentro del archivo LOG que se inicializó con la creación del entorno. La salida tendrá diferentes formatos según el tipo de modelo que se esté optimizando, todas, eso sí, contarán con una estructura similar entre sí aunque con variaciones específicas (todas explicadas dentro del apartado "*Logging*", en el documento "*refman.pdf*", a partir de la página 669), en caso de la resolución de un QP la salida tendrá la siguiente estructura:

```
Gurobi 7.5.2 (win64) logging started 11/15/18 17:13:40

Academic license - for non-commercial use only
Optimize a model with 5 rows, 2 columns and 8 nonzeros
Model has 3 quadratic objective terms
Coefficient statistics:
  Matrix range     [1e+00, 2e+00]
  Objective range  [2e+00, 6e+00]
  QObjective range [1e+00, 2e+00]
  Bounds range     [0e+00, 0e+00]
  RHS range        [2e+00, 3e+00]
Presolve removed 2 rows and 0 columns
Presolve time: 0.00s
Presolved: 3 rows, 2 columns, 6 nonzeros
Presolved model has 3 quadratic objective terms
Ordering time: 0.00s

Barrier statistics:
 Free vars  : 1
 AA' NZ     : 6.000e+00
 Factor NZ  : 1.000e+01
 Factor Ops : 3.000e+01 (less than 1 second per iteration)
 Threads    : 1

                  Objective                Residual
Iter       Primal          Dual         Primal    Dual     Compl     Time
   0   4.92797080e+05 -5.00785433e+05  2.00e+03 2.00e+00  9.98e+05     0s
   1   5.05355583e+03 -8.03503102e+03  1.03e+02 4.09e-08  1.13e+05     0s
   2   3.31107593e+00 -3.28612146e+03  4.46e-14 2.82e-13  6.58e+02     0s
   3   3.29800706e+00 -3.94769712e+00  3.82e-17 9.19e-14  1.45e+00     0s
   4   1.01067660e+00  5.11117612e-01  1.67e-16 0.00e+00  9.99e-02     0s
   5   8.08706447e-01  7.71630242e-01  1.55e-15 1.11e-15  7.42e-03     0s
   6   7.77916314e-01  7.77759898e-01  1.22e-15 0.00e+00  3.13e-05     0s
   7   7.77777916e-01  7.77777760e-01  5.55e-16 6.24e-16  3.13e-08     0s
   8   7.77777778e-01  7.77777778e-01  0.00e+00 0.00e+00  3.13e-11     0s

Barrier solved model in 8 iterations and 0.01 seconds
Optimal objective 7.77777778e-01
```

Figura 1-33. Archivo resultado de una optimización en Gurobi.

3.6.8 Captura de solución:

Penúltima parte del proceso dentro del código para la optimización del modelo será la captura de solución, tomando valores asociados a atributos del modelo que ofrece Gurobi, de los cuales se tendrán una gran variedad dentro de la que elegir, Los más interesantes a la hora de operar posteriormente con ellos o tener información sobre la optimización, serán, en primer lugar en orden de importancia, la solución que Gurobi da a cada una de las variables del modelo, y en segundo lugar el estado de la optimización que, de la misma forma que la función de error, a través de un código de estado, especifica el estado del modelo, si ha sido cargado, si se ha obtenido el óptimo, en el caso que sea irrealizable…etc (todos los códigos se pueden ver en el apartado "*Optimization Status Codes*", en el documento "*refman.pdf*", a partir de la página 645). La llamada a ambas funciones se realizará de la siguiente manera:

```
//CAPTURA DE SOLUCION\\
//Primero INT DEL ESTADO OPTIMO; Segundo ARRAY X (solucion)
error = GRBgetintattr(model, GRB_INT_ATTR_STATUS, &estadoptimo);
if (error != 0) { goto QUIT; }


double sol[2];
error = GRBgetdblattrarray(model, GRB_DBL_ATTR_X, 0, 2, sol);
if (error != 0) { goto QUIT; }
```

Figura 1-34. Captura de solución a través de Gurobi.

En orden de llamada se guardará con la primera el estado de la optimización, en una variables de tipo entero, y con la segunda el vector con los valores tipo double referentes a la resolución de la optimización.

3.6.9 Liberación del sistema:

El último paso a seguir será la liberación del modelo y el entorno que se han inicializado para la optimización, proceso de seguridad para que no se produzcan errores en el caso de que se llame a funciones de Gurobi de forma recursiva o se ejecuten diferentes códigos de optimización, asegurando así que existan la menor cantidad de errores. La llamada a estas funciones serán simplemente las dos líneas de código que se pueden ver a continuación:

```
//LIBERACION DEL SISTEMA\\

GRBfreemodel(model);

GRBfreeenv(env);
```

Figura 1-35. Liberación del sistema.

Únicamente destacable el orden en el que se llaman a ambas funciones, ya que nunca se puede liberar el espacio del entorno si es que existe un modelo dentro del mismo, por lo que se debe liberar el espacio de todos los modelos que se hayan inicializado dentro del entorno, y posteriormente el del entorno en cuestión.

4 CONTROL PREDICTIVO BASADO EN MODELO

4.1 Planteamiento de las bases sobre estrategias de control predictivo:

El control predictivo es un amplio campo en el que se engloban diferentes métodos de control los cuales giran en torno a una serie de ideas comunes, no siendo al contrario una estrategia de control específica para todos. Las ideas que de forma general aparecen en la mayoría de controladores predictivos son:

- Utilización de un modelo del sistema real con el fin de la predicción de la salida del proceso en los futuros instantes de tiempo hasta un límite, autoimpuesto por el diseñador, denominado horizonte.

- Optimización para la obtención de las señales de control, a través de la minimización de cierta función objetivo, la cual puede diferir entre las diferentes estrategias de control.

- Horizontes deslizantes. En cada instante de tiempo mientras se está ejecutando cualquier tipo de estrategia de control, la predicción que se realiza sobre las salidas del modelo se irá desplazado hacia el futuro de igual manera. Esto daría como resultado que en cada instante de tiempo se tienen una serie de predicciones de las que se debe escoger una y desechar el resto.

Teniendo en cuenta que estas características se reproducirán en las diferentes estrategias de control predictivo, sólo será de forma conceptualmente similar, por ejemplo, el cálculo de un modelo para el sistema se puede abordar de diferentes formas (modelos ante respuesta impulsional o ante escalón), o la función objetivo que se utilice para la optimización puede penalizar de manera distinta cada uno de los términos, lo que resultará en diferencias ciertamente pequeñas, pero que pueden provocar comportamientos muy distintos del control.

En los siguientes apartados se abordarán estos conceptos teóricos base que se acaban de mencionar, además de algunos otros, que será de utilidad conocer en tareas más avanzadas, así como, la función objetivo, respuesta libre y respuesta forzada del sistema o las restricciones del proceso.

4.1.1 Estrategia general de los controladores predictivos:

La metodología seguida por los controladores predictivos basados en modelo se caracteriza por tener una representación como la que se puede observar a continuación (Figura *2-1*). Tomando como base el modelo del sistema real, y de la misma forma que se enunció como primera características de un control predictivo, se predicen las salidas futuras para un horizonte de predicción preestablecido (N).

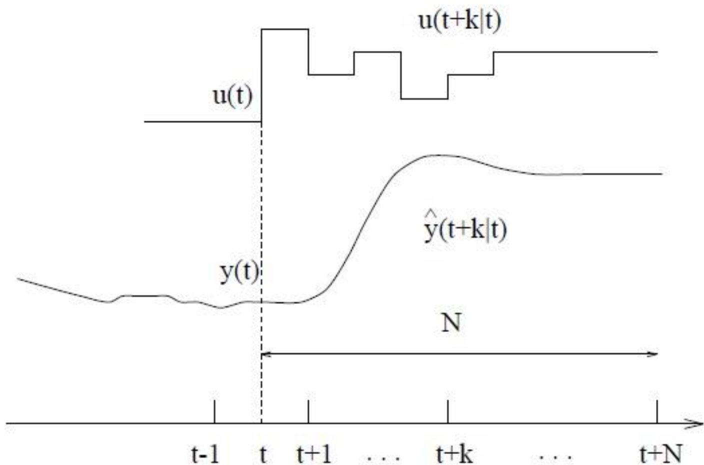


Figura 2-1. Metodología control predictivo.

Partiendo de las salidas futuras predichas, $\hat{y}(t+k|t)$ siendo $k = 1 \ldots N$, las cuales dependen de los valores conocidos hasta el instante de tiempo actual t, entradas y salidas pasadas del sistema, y de las señales de control futuras, $u(t+k|t)$ siendo $k = 0 \ldots N-1$, que son las que se calculará y mandará al sistema cerrando así el bucle.

La obtención del este conjunto de señales de control futuras se calcula a través de la minimización de una función objetivo, logrando así minimizar el error de la salida del proceso con relación a la trayectoria de referencia, $w(t+k)$, será lo que se llame obtención de la ley de control. Esta función que se pretende minimizar suele tomar forma de una función cuadrática de los errores entre la salida predicha y la trayectoria de referencia, incluyendo en muchos casos el esfuerzo de control. En el caso de que la función objetivo tome forma de función cuadrática, el modelo sea lineal y no existan restricciones, se puede obtener una solución explícita, si se dan otras condiciones se deberá optar por el uso de un método iterativo de optimización.

Con el conjunto de señales de control obtenidas se toma como válida la primera de las mismas, $u(t|t)$, y se desechan las señales de control restantes. Ésta es enviada al proceso obteniendo así la señal de salida para el instante de muestreo siguiente, $y(t+1)$, y se vuelve a repetir el proceso completo desde el primer paso partiendo de este nuevo valor con el objetivo de calcular la señal de control en el siguiente instante de muestreo, $u(t+1|t+1)$.

La elección de este tipo de estrategia de control tiene una serie de ventajas deducibles por lo ya expuesto:

- Posibilidad de utilización en gran variedad de procesos, desde procesos dinámicamente simples hasta sistemas con grandes retardos, de fase no mínima o inestable.
- No requiere un conocimiento muy profundo en teoría de control, al basarse en conceptos relativamente intuitivos como los mencionados.
- El conocimiento sobre las futuras referencias y las futuras predicciones puede ser de mucha utilidad.
- Permite ampliar el control al caso multivariable de manera relativamente sencilla.
- La inclusión de restricciones es simple.
- La metodología de control predictivo está totalmente abierta a nuevas versiones, lo que la convierte en un campo con, potencialmente, indefinidas extensiones a futuro al únicamente estar basada en torno a unos principios básicos sencillos sin limitaciones.

4.1.2 Modelo del proceso:

La base, que a la vez da nombre a este tipo de controladores, es el modelo del proceso que se pretende controlar. El objetivo debe ser recrear todas las dinámicas del proceso que sean posibles, para calcular un modelo lo más similar al mismo. Se elegirá un método de modelado u otro en cuestión de las necesidades que se tengan sobre el cálculo de la salida predicha en los instantes de tiempo futuros. Dependiendo de la estrategia que se utilice se pueden utilizar diferentes modelos para representar la relación de las salidas con las entradas medibles (algunas de las cuales serán variables manipulables y otras se pueden considerar como perturbaciones medibles). Además se deberá tener en cuenta un modelo de perturbaciones, el cual englobará las dinámicas que no aparecen reflejadas en el modelo del proceso, como las entradas no medibles, el ruido o los errores de modelado. Lo que indica que se necesita realizar dos modelos, uno principal para el proceso propiamente dicho y otro para las perturbaciones.

- Modelo de respuesta impulsional y ante escalón:

Modelos en los que lo que se busca es analizar la respuesta del sistema real a cierto tipo de entradas, siendo éstas impulsos o escalones. Si se somete al proceso a un impulso unitario (Figura *2-2*, izquierda)la salida estará relacionada con la suma de los sucesivos valores de h hasta un valor N, siendo éste último valor relativamente grande pudiendo resultar un inconveniente a la hora de elegir éste método.

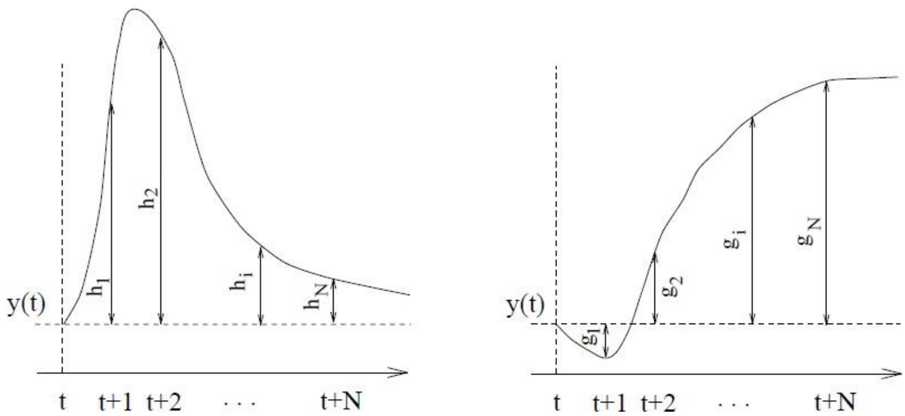


Figura 2-2. Modelo de respuesta impulsional y ante escalón.

El modelo de respuesta ante escalón es muy similar al anterior exceptuando porque en este caso la señal de entrada al proceso será un escalón y la salida estará en función de las sucesivas g y la variación de la señal de entrada (Figura *2-2*, derecha). Este proceso contempla las mismas ventajas e inconvenientes que el primero.

- Función de transferencia:

Se utiliza el concepto de función de transferencia, la cual es el cociente entre dos polinomios, B y A, de la siguiente forma:

$$A(z^{-1})y(t) = B(z^{-1})u(t)$$

$$\begin{aligned} A(z^{-1}) &= 1 + a_1 z^{-1} + a_2 z^{-2} + \cdots + a_{na} z^{-na} \\ B(z^{-1}) &= b_1 z^{-1} + b_2 z^{-2} + \cdots + b_{nb} z^{-nb} \end{aligned} \Longrightarrow \hat{y}(t+k \mid t) = \frac{B(z^{-1})}{A(z^{-1})} u(t+k \mid k)$$

Figura 2-3. Modelo basado en función de transferencia.

Este tipo de modelado posee la ventaja de necesitar pocos parámetros aunque es importante tener cierto conocimiento del proceso, ya que el orden de los polinomios definirá la fiabilidad del modelo, y en el caso de no ser los correctos es probable que haya dinámicas del proceso que no sean modeladas.

- Espacio de estados:

La representación de un sistema en espacio de estados tiene de forma genérica la siguiente forma:

$$\begin{aligned} x(t) &= Ax(t-1) + Bu(t-1) \\ y(t) &= Cx(t) \end{aligned}$$

Figura 2-4. Modelo en espacio de estados.

Posee la ventaja de servir para sistemas multivariables a la vez que permite analizar la estructura interna del proceso, aunque el uso de este tipo de modelos puede suponer una complejidad añadida para el proceso de cálculo.

4.1.3 Modelo de perturbaciones:

Del mismo modo que se elige un tipo de modelo para el proceso, se puede elegir un modelo para el modelado que mejor represente las perturbaciones del mismo. Modelos CARIMA (*Controlled Auto-Regressive and Integrated Moving Average*, Autorregresivo e Integrado de Media Móvil) puede ser un ejemplo muy utilizado para este tipo de modelos, mostrando las perturbaciones (diferencias entre la salida medida y la calculada por el modelo) representadas como el cociente de la izquierda (Figura *2-5*, izquierda).

$$n(t) = \frac{C(z^{-1})e(t)}{D(z^{-1})} \implies n(t) = \frac{e(t)}{1 - z^{-1}}$$

Figura 2-5. Modelo de perturbaciones.

El factor $e(t)$ representa el ruido, de media cero, el polinomio D incluye explícitamente el término integrador y por último, normalmente, el polinomio C se suele considerar igual a la unidad, obteniendo el resultado para un caso particular en el que se permite incluir la perturbación como constate, representada en el cociente de la derecha (Figura *2-5*, derecha).

4.1.4 Función objetivo:

Como uno de los pilares base en los conceptos genéricos del control predictivo se busca diseñar una función específica de costes, a la que se llamará función objetivo de forma general, la cual tendrá diferentes matices dependiendo de la estrategia de control predictivo que se elija, pero de forma general buscará que la salida futura, en el horizonte definido, siga una determinada trayectoria de referencia y a la vez se puedan penalizar los esfuerzos de control requeridos, lo que deriva en una función general (Figura *2-6*) con la siguiente forma:

$$J(N_1, N_2, Nu) = \sum_{j=N_1}^{N_2} \delta(j)[\hat{y}(t+j \mid t) - w(t+j)]^2 + \sum_{j=1}^{Nu} \lambda(j)[\triangle u(t+j-1)]^2$$

Figura 2-6. Función objetivo genérica.

En la función existen dos sumandos claramente definidos, el primero será la penalización de los errores cuadráticos futuros (diferencia entre las salidas predichas y la trayectoria de referencia), y el segundo será la penalización del esfuerzo de control, el cual puede no tenerse en cuenta según la estrategia de control predictivo elegida. Igualmente también según la estrategia se pueden diferenciar las que utilizan, para este último cálculo en la función objetivo, los valores de la señal de control en vez de sus incrementos.

4.1.4.1 Análisis de los parámetros de la función objetivo:

Después de haberse presentado la forma general de la función objetivo, en ella aparecen parámetros que no se ha mencionado previamente o no se han explicado en su totalidad, por lo que con el fin de entenderlos, se ven representados los siguientes grupos de parámetros:

- Horizontes:

No sólo existe un valor que define el horizonte del control predictivo, como se ha mencionado de forma general con anterioridad, sino que existen tres diferentes valores para definir este parámetro:

- N1 u horizonte inicial de predicción
- N2 u horizonte final de predicción
- Nu u horizonte de control

En conjunto, N1 y N2 serán los límites de los instantes de muestreo en los que se decide utilizar el control (y por ende, que la salida siga a la referencia). En caso de que N1 no corresponda al primer instante de muestreo, que por lo general será lo que ocurra excepto que se presente un proceso con retardo en el que un valor de N1 menor que el tiempo muerto no tendría sentido, significa que los primeros instantes hasta N1 los errores no son relevantes.

El valor N2, en el caso general en que N1 coincida con el primer instante de muestreo y desde el inicio actúe el controlador, representará el horizonte de predicción del que se ha hablado previamente, éste no tiene por qué coincidir con el valor del horizonte de control Nu, pero siempre se debe de cumplir que el horizonte de control sea menor o igual al horizonte de predicción:

$$N_2 \geq N_U$$

- Coeficientes de ponderación:

Se puede apreciar delante de cada uno de los sumandos de la expresión de la función objetivo (Figura *2-6*) la existencia de dos coeficientes que multiplican a cada uno de los términos, éstos son coeficientes de ponderación que sirven para penalizar más o menos el término de error o el esfuerzo de control.

- δ (t): coeficiente de ponderación que penaliza el término de error.
- λ (t): coeficiente de ponderación que penaliza el esfuerzo de control.

Finalmente estos términos responderán generalmente a valores constantes o secuencias exponenciales y servirán como parámetros de sintonización del controlador, siendo estos los que permitan definir el tipo de control que se quiera, más suave penalizando los valores más alejados de cada instante de tiempo, o más agresivo penalizando los primero valores.

- Trayectoria de referencias futuras:

Representada en el primer sumando de la función objetivo por el vector $w(t+j)$. Conociendo los valores de este vector, se conocen los valores que se quiere que tome la salida en el futuro, poniendo a la vista una de las claras ventajas que ofrece el control predictivo, si se conocen la evolución futura de la referencia, se puede hacer que el sistema empiece a reaccionar antes de que el cambio se haya hecho efectivo, lo que resultará en mejoras de prestaciones y de comportamiento del proceso.

En caso de que las referencias futuras no sean conocidas se pueden aplicar diferentes criterios, por ejemplo, la utilización de un vector de referencias futuras que no tenga por qué coincidir exactamente con el real pero que sea una aproximación y facilite el trabajo del controlador, o también el uso de referencias constante a lo largo de cada uno de los instantes de muestreo, lo que no permite que se realice una reacción preventiva del proceso ante el conocimiento de la referencia futura, pero sí permite la realización del cálculo del control.

4.1.5 Respuesta libre y respuesta forzada:

Existen dos conceptos que serán repetidos de manera constante dentro de los controladores predictivos basados en modelo, la respuesta libre y la respuesta forzada del sistema. Expresando la secuencia de acciones de control como la suma de dos señales, $u_f(t)$ y $u_c(t)$, las cuales tendrán la siguiente forma (Figura *2-7*, superior), la respuesta libre $y_f(t)$ y la respuesta forzada $y_c(t)$ (Figura *2-7*, inferior), serán las respuestas del sistema ante cada una de las dos señales de control divididas.

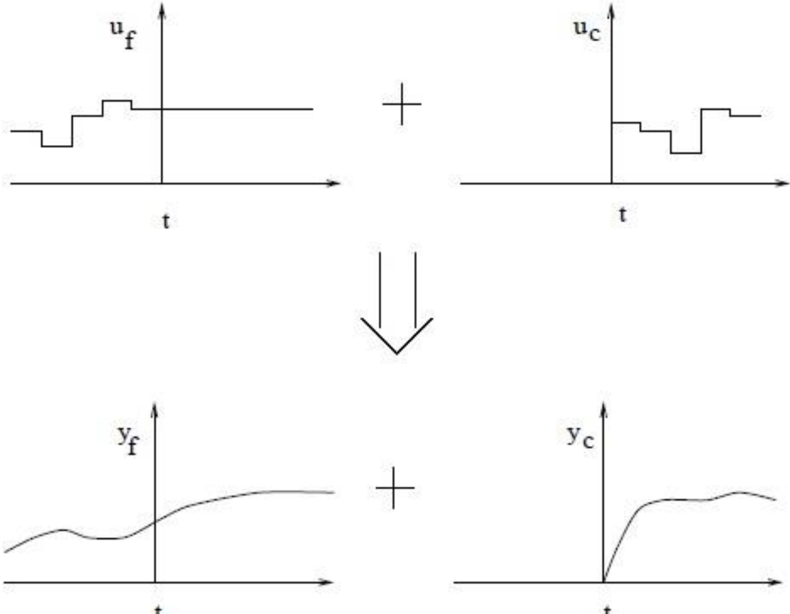


Figura 2-7. Señales de entrada y salida para la respuesta libre y forzada respectivamente.

En definitiva, se llamará respuesta libre a la señal del sistema anterior al instante “t”, en el que entraría el efecto del control predictivo, únicamente dependiendo de las señales pasadas del sistema manteniendo la señal constante e igual al último valor de la salida, sin que el controlador le influya.

En cambio, se llamará respuesta forzada a la señal del sistema que corresponde a la señal de control en los instantes futuros al instante t, donde ya se tiene en cuenta el efecto del control predictivo.

4.1.6 Restricciones:

A la hora de ejecutar un control en un sistema real, en primer lugar se debe de tener en cuenta que todos los procesos tienen limitaciones, por ejemplo, los actuadores están limitados en cuanto al rango de acción que poseen, al igual que a la velocidad en la que pueden variar, o puede ocurrir que los actuadores tengan limitaciones en las posiciones que pueden tomar (como podría ser una válvula que sólo puede estar totalmente abierta o cerrada), o incluso limitaciones del alcance de los sensores, limitando alguna variable a un máximo o mínimo diferente del real. Estas limitaciones además, pueden ser autoimpuestas a la hora del diseño, ya que por motivos económicos puede ser más rentable que el sistema trabaje en torno a unos límites. Todo este tipo de situaciones dentro del modelado de un control predictivo para cualquier proceso se llamarán restricciones.

De forma habitual en multitud de algoritmos de control predictivo se suelen tomar restricciones en la amplitud de la salida, la velocidad de cambio de la señal de control (*slew rate*) y en la amplitud de la salida (Figura *2-8*).

$$\begin{array}{ccccc} \underline{U} & \leq & u(t) & \leq & \overline{U} \\ \underline{u} & \leq & u(t)-u(t-1) & \leq & \overline{u} \\ \underline{y} & \leq & y(t) & \leq & \overline{y} \end{array}$$

Figura 2-8. Ecuaciones de restricciones base.

Este tipo de restricciones se podrá igualmente expresar para cualquier tipo de sistema, incluso si se tiene el caso de un proceso multivariable con m entradas, n salidas y restricciones en el horizonte N. Aunque, en cualquier caso, siempre que se añadan restricciones a la función objetivo, la minimización no permitirá un proceso de solución analítico, al contrario que en el caso sin restringir.

Además otra forma de clasificar el tipo de restricciones que se quiere imponer, más allá del punto de aplicaciones de las mismas a la entrada o a la salida del proceso, se pueden clasificar ateniendo a su relevancia de cumplimiento durante el proceso, pudiendo diferenciar una restricciones “duras”, las cuales serán de estricto cumplimiento durante la completa ejecución del control, y otras restricciones “blandas”, las cuales al no ser cruciales para la seguridad o eficiencia del proceso, pueden no cumplirse en algún momento, eso sí, acompañadas de la correspondiente penalización dentro de la función objetivo si eso ocurre.

4.2 Presentación de estrategias de control predictivo:

A continuación se presentarán, de forma superficial, diferentes estrategias de control predictivo conocidas dentro de la industria, en ellas se podrán observar las diferencias en cuanto al planteamiento, que se ha comentado anteriormente, en cuanto a elección de modelo de predicción, función objetivo o cálculo de la ley del control, con el fin de clarificar la elección de una estrategia frente a otra para su implementación.

4.2.1 *Dynamic Matrix Control* (DMC, Control de Matriz Dinámica):

Desarrollado por *Cutler y Ramaker*, pertenecientes a *Shell Oil Co*, extendiéndose su uso en la industrial, principalmente petroquímica, al destacar por su aplicación en grandes sistemas multivariables con consideración de restricciones. Método que utiliza como proceso de modelado la respuesta temporal ante escalón (Figura *2-2*, derecha), de la cual se partirá para la obtención de la predicción de la salida, considerando las perturbaciones constantes:

$$\hat{y}(t+k \mid t) = \sum_{i=1}^{k} g_i \triangle u(t+k-i) + f(t+k)$$

Figura 2-9. Ecuación de predicción DMC.

Donde $f(t+k)$ será la respuesta libre del proceso, al no depender de las acciones de control futuras. En el caso de que el proceso sea asintóticamente estable, se podrá calcular el sumatorio de la respuesta libre ya que los coeficientes g_i de la respuesta ante escalón tienden a un valor constante después de N períodos de muestreo (en caso de no ser estable no existe un valor para N haciendo que no sea posible calcular la respuesta libre al margen de realizar alguna concesión), por lo que la respuesta libre se podría calcular con la siguiente expresión:

$$f(t+k) = y_m(t) + \sum_{i=1}^{N} (g_{k+i} - g_i) \triangle u(t-i)$$

Figura 2-10. Respuesta libre DMC.

Ahora que la predicción de las salidas se puede calcular a lo largo del horizonte, definidos en función de los coeficientes de la respuesta ante escalón, los incrementos en la señal de control y la respuesta libre (Figura *2-11*, izquierda), pudiéndose además representar en forma matricial (Figura *2-11*, derecha):

$$\begin{aligned} \hat{y}(t+1 \mid t) &= g_1 \triangle u(t) + f(t+1) \\ \hat{y}(t+2 \mid t) &= g_2 \triangle u(t) + g_1 \triangle u(t+1) + f(t+2) \\ &\vdots \\ \hat{y}(t+p \mid t) &= \sum_{i=p-m+1}^{p} g_i \triangle u(t+p-i) + f(t+p) \end{aligned} \quad \Longrightarrow \quad \hat{\mathbf{y}} = \begin{bmatrix} g_1 & 0 & \cdots & 0 \\ g_2 & g_1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ g_m & g_{m-1} & \cdots & g_1 \\ \vdots & \vdots & \ddots & \vdots \\ g_p & g_{p-1} & \cdots & g_{p-m+1} \end{bmatrix} \mathbf{u} + \mathbf{f}$$

Figura 2-11. Ecuación de predicción matricial DMC.

A continuación se analizará este algoritmo de control para el caso monovariable sin restricciones, comentando adicionalmente al final los puntos claves para la extensión del algoritmo al caso multivariable.

El objetivo siempre será llevar al proceso lo más cerca posible al valor de referencia que se requiera, a lo que además, ya que se trata de controladores basados en predicción, se le puede añadir una penalización a los esfuerzos de control. Por ello, como función objetivo se tendrá la posibilidad de seleccionar minimizar sólo los errores futuros al cuadrado (Figura *2-12*, término izquierda), o además añadir el esfuerzo de control, dando así una representación más genérica (Figura *2-12*, término derecha).

$$J = \sum_{j=1}^{p} [\hat{y}(t+j \mid t) - w(t+j)]^2 + \sum_{j=1}^{m} \lambda [\triangle u(t+j-1)]^2$$

Figura 2-12. Función objetivo DMC.

Como se dijo anteriormente, si se toma el caso sin restricciones, se puede obtener el resultado de forma analítica calculando la deriva de J e igualándola a cero, obteniendo el siguiente resultado general:

$$\mathbf{u} = (G^T G + \lambda I)^{-1} G^T (\mathbf{w} - \mathbf{f})$$

Figura 2-13. Obtención del vector de señales de control.

Donde se obtendrá el vector $\boldsymbol{u}$ (las señales futuras de control para el proceso en forma de variación, de la cual en cada tiempo de muestreo se tomará únicamente la primera de ellas), igualado al producto de una matriz (obtenida al resolver la expresión, $(G^T G + \lambda I)^{-1} G^T (w - f)$) por la diferencia del vector de referencias futuras y la respuesta libre del proceso, lo cual sería el error del proceso si no se realizase ninguna acción de control externa.

Para casos más complejos, como podría ser en el que se tengan varias variables de salida y de entrada, se deberá extender el problema a todas las variables que se añadan por encima del caso monovariable ya tratado. Esto significará que las ecuaciones generales siguen siendo completamente válidas pero con la adición de, en caso de los vectores de predicción de salidas, variaciones de control, respuesta libre, por superposición al tratarse de modelos lineales, se les debe añadir los valores correspondientes a las salidas adicionales que se tengan. Para el caso de la matriz dinámica G, se ampliará a una matriz formada por submatrices en las que se representará ordenadamente, en cada una de ellas, las respuestas ante escalón correspondiente a cada una de las diferentes entradas. Esta ampliación se deberá realizar análogamente para los factores de ponderación en el proceso de minimización, tanto para los errores o también para los esfuerzos de control, en el caso que se haya optado por la utilización de la ecuación genérica.

4.2.2 *Model Algorithmic Control* (MAC):

Método conocido también como *Model Predictive Heuristic Control*, muy similar a la anterior, es, entre las diferentes estrategias de control predictivo, una de las posibilidades más intuitivas y simples, utiliza un modelo de respuesta impulsional del proceso e introduce una modificación al concepto de la trayectoria de referencias futuras, al implementarla como un sistema de primer orden que va evolucionando, desde la salida en el instante de tiempo actual hasta el punto de equilibrio, con una constante de tiempo determinada.

Partiendo de la ecuación del modelo de respuesta impulsional (Figura *2-2*, izquierda), los coeficientes h_j serán truncados hasta el elemento N y asumiendo el sistema estable y causal, la predicción tendrá la siguiente forma (Figura *2-14*, izquierda), cuyo sumatorio se podrá dividir en dos partes como se puede ver a continuación (Figura *2-14*, derecha).

$$\hat{y}(t+k \mid t) = \sum_{j=1}^{N} h_j u(t+k-j) + \hat{n}(t+k \mid t) \quad \Longrightarrow \quad f_r(t+k) = \sum_{j=k+1}^{N} h_j u(t+k-j), \qquad f_o(t+k) = \sum_{j=1}^{k} h_j u(t+k-j)$$

Figura 2-14. Ecuación de predicción MAC.

En cada una de las partes en las que se divide la ecuación anterior se tienen las dos respuestas posibles para el sistema, en primer lugar la respuesta libre (predicción de la respuesta del sistema asumiendo una nula acción del controlador), y consiguientemente la respuesta forzada (componente que contempla la salida del proceso teniendo en cuenta las acciones de control calculadas). A continuación se tiene la ecuación de predicción considerando las perturbaciones, $\hat{n}$, constantes en el instante de tiempo t:

$$\hat{y}(t+k \mid t) = f_r + f_o + \hat{n}(t \mid t)$$

Figura 2-15. Suma completa de la ecuación de predicción MAC.

Partiendo de los horizontes definidos M y N, se pueden definir dos vectores para las acciones de control, u+ como vector de acciones futuras (vector no incrementar de acciones de control desde $u(t) \dots u(t+M-1)$) y u- como vector de acciones de control pasadas (vector de acciones pasadas de control desde $u(t-N+1) \dots u(t-1)$). Colocando, por último, los coeficientes de la respuesta impulsional de la siguiente manera (Figura *2-16*, superior) se tendrá la expresión de la predicción escrita en forma matricial (Figura *2-16*, inferior).

$$\begin{bmatrix} \hat{y}(t+1) \\ \hat{y}(t+2) \\ \vdots \\ \hat{y}(t+M) \end{bmatrix} = \begin{bmatrix} h_1 & 0 & \cdots & 0 \\ h_2 & h_1 & \cdots & 0 \\ \dots & \dots & \ddots & \dots \\ h_M & h_{M-1} & \cdots & h_1 \end{bmatrix} \mathbf{u}_+ + \begin{bmatrix} h_N & \cdots & h_i & \cdots & h_2 \\ 0 & \cdots & h_j & \cdots & h_3 \\ \dots & \ddots & \dots & \dots & \dots \\ 0 & \cdots & h_N & \cdots & h_{M+1} \end{bmatrix} \mathbf{u}_- + \begin{bmatrix} \hat{n}(t+1) \\ \hat{n}(t+2) \\ \vdots \\ \hat{n}(t+M) \end{bmatrix}$$

$$\Downarrow$$

$$\mathbf{y} = \mathbf{H}_1 \, \mathbf{u}_+ + \mathbf{H}_2 \, \mathbf{u}_- + \mathbf{n}$$

Figura 2-16. Predicción en forma matricial MAC.

La trayectoria de referencia, como se ha comentado al principio, será la de un sistema de primer orden que se aproxime desde la salida del proceso hasta el valor de la referencia conocida, la forma de la trayectoria dependerá de un coeficiente α, que determinará la velocidad con la que se quiere llegar a la referencia deseada, permitiendo así controlar la agresividad del controlador. De manera homóloga a lo que se comentó sobre los fundamentos del control predictivo, se busca minimizar una función objetivo que contenga el error y el esfuerzo de control, por ello si se representan los errores futuros como la resta de la referencia y la salida calculada antes en la ecuación de predicción (Figura *2-17*, superior), además se pueden agrupar los valores que se conocen como, entradas pasadas, salida actual, referencia, dentro del vector $\boldsymbol{f}$, obteniendo la siguiente función objetivo genérica (Figura *2-17*, centro):

$$\mathbf{e} = \mathbf{w} - \mathbf{H_2 u_-} - \mathbf{n} - \mathbf{H_1 u_+} = \mathbf{w} - \mathbf{f} - \mathbf{H_1 u_+}$$

$$\Downarrow$$

$$J = \mathbf{e}^T \mathbf{e} + \lambda \mathbf{u_+}^T \mathbf{u_+}$$

$$\Downarrow$$

$$\mathbf{u_+} = (\mathbf{H}_1^T \mathbf{H}_1 + \lambda \mathbf{I})^{-1} \mathbf{H_1}^T (\mathbf{w} - \mathbf{f})$$

Figura 2-17. Función objetivo MAC.

En el caso de que se tome esta consideración pero sin la existencia de restricciones, se obtiene una solución explícita (Figura *2-17*, inferior) para la minimización de la función.

4.2.3 *Predictive Functional Control* (PFC):

Controlador que emplea, como excepción a las estrategias anteriores, un modelo en espacio de estados (Figura *2-4*), además de dos características que combinadas darán como resultado la señal de control, ciertas *funciones base* predeterminadas y *Puntos de coincidencia* que servirán para evaluar la función de coste a lo largo del horizonte.

En primer lugar el concepto de puntos de coincidencia, emplea un conjunto de puntos seleccionados en el horizonte de predicción en lugar de la función completa con el fin de, simplificar los cálculos y la comparación de la predicción de la salida y la salida buscada, haciendo coincidir únicamente los puntos de coincidencia en lugar de todo el horizonte.

El segundo concepto introducido en esta estrategia son las funciones base. Buscan la parametrización de la señal de control a través de la combinación lineal de ciertas funciones predeterminadas, de escasa complejidad, las cuales se elegirán en base a la naturaleza del proceso y a la referencia, tomando siempre las más apropiadas. Estas funciones suelen ser de tipo polinómico y usualmente se realizan combinaciones entre los siguientes tipos de funciones:

- Función de escalón ($B_1(k) = 1$).
- Función de rampa ($B_2(k) = k$).
- Función de parábola ($B_3(k) = k^2$).

A través de este tipo de funciones se debería poder expresar la mayoría de referencias como una combinación de estas. Obteniendo la señal de controles futuros representada de la siguiente forma:

$$u(t+k) = \sum_{i=1}^{n_B} \mu_i(t) B_i(k)$$

Figura 218-18. Señales de control futuras PFC.

La elección de estas funciones base puede asegurar un comportamiento predeterminado con una señal suave de entrada al proceso, lo que se puede traducir en un punto muy positivo a la hora de controlar sistemas no lineales.

La función objetivo que se busca minimizar tendrá la siguiente forma:

$$J = \sum_{j=1}^{n_H} [\hat{y}(t+h_j) - w(t+h_j)]^2$$

Figura 2-19. Función objetivo PFC.

Donde el vector de referencias futuras, $w(t+j)$, podría ser, como se ha comentado en estrategias anteriores, una aproximación a un comportamiento de primer orden. El valor h_j es la forma de representar los puntos de coincidencia en los que se considerará el error de predicción, $j = 1 \dots n_H$. El número de puntos de coincidencia elegidos deben ser al menos iguales al número de funciones base utilizadas, además, debe tomarse en consideración la elección de los puntos que tengan mayor información sobre la estabilidad y robustez del proceso, ya que estos puntos servirán como parámetros de sintonización del controlador.

El cálculo de la ley de control para esta estrategia implica la computación de los valores de $\mu_i(t)$, representados en la ecuación de la figura (Figura *2-18*). Para el caso sencillo de un proceso con una única salida y una única entrada (SISO, *Single Input single Output*) sin restricciones se obtendrá la ley de control, en primer lugar separando la respuesta libre y forzada, donde la función objetivo quedará reescrita de la siguiente forma (Figura *2-20*, izquierda):

$$J = \sum_{j=1}^{n_H} [\mathbf{y}_B(h_j)\,\mu - d(t+h_j)]^2 \quad \Longrightarrow \quad J = (\mathbf{Y}_B\mu - \mathbf{d})^T(\mathbf{Y}_B\mu - \mathbf{d})$$

Figura 2-20. Función objetivo PFC.

Donde el vector μ será los sucesivos valores de $\mu_i(t) \dots \mu_{nB}(t)$, $\boldsymbol{Y}_B(h_j)$ será un vector que englobe las sucesivas respuesta del proceso antes cada una de las funciones base (Bi) utilizadas en los puntos de coincidencia h_j, $Y_B(h_j) \dots Y_{BnB}(h_j)$. Si se define $\boldsymbol{d}$ como un vector en el que se agrupan los sucesivos $d(t+h1) \dots d(t+hnH)$, la función objetivo se podrá representar en forma de vector (Figura *2-20*, derecha).

Minimizando la función J con respecto a los coeficientes μ, y a continuación si se toma el vector de ponderación de las funciones base como la solución de $\boldsymbol{Y}_{B\mu} = \boldsymbol{d}$. La señal de control será dada por:

$$u(t) = \sum_{i=1}^{n_B} \mu_i(t) B_i(0)$$

Figura 2-21. Señal de control minimizada PFC.

4.2.4 *Generalized Predictive Control* (GPC, Control Predictivo Generalizado):

El control predictivo generalizado, desarrollado por *Clarke et al*, es uno de las estrategias de control predictivo más utilizadas, no solo en la industria sino también con objetivos académicos, pudiendo dar soporte para diferentes problemas de control en multitud de variedad de procesos diferentes, mostrando cierto nivel de robustez en sus aplicaciones industriales. Como características destacables, previas al análisis completo:

- Estrategia de control basada en la utilización de un modelo CARIMA, donde la perturbación viene dada por un ruido blanco coloreado por un polinomio, en este caso C(z-1):

$$A(z^{-1})y(t) = B(z^{-1})z^{-d}\ u(t-1) + C(z^{-1})\frac{e(t)}{\triangle} \qquad \text{con} \quad \triangle = 1 - z^{-1}$$

Figura 2-22. Modelo CARIMA.

- La ecuación de predicción deriva de la resolución de una ecuación diofántica, la cual se puede calcular recursivamente de manera muy eficiente, detalle muy relevante a la hora de realizar la elección del algoritmo que se va a usar para la aplicación en este trabajo. Esta estrategia de control utiliza, la ya mencionada con anterioridad, función de coste cuadrática (Apartado.*4.1.4*).

La idea básica de esta estrategia tiene muchas similitudes a las anteriores, se busca calcular una secuencia de acciones de control futuras que minimice la función objetivo cuadrática que se acaba de mencionar, la cual por un lado contempla la discrepancia entre la salida predicha y la salida del sistema, y por otro lado el esfuerzo de control para obtener dicha salida para el proceso. Esta idea básica es muy similar a lo comentado en las estrategias de control anteriores, a continuación se analizará la formulación de un GPC en profundidad.

4.2.4.1 Planteamiento general de un GPC para un sistema SISO:

Tratándose de un proceso SISO (*Single Input Single Output*), pueden ser descritos, si se considera un punto de trabajo y posteriormente de ser linealizado, por:

$$A(z^{-1})y(t) = z^{-d}B(z^{-1})u(t-1) + C(z^{-1})e(t)$$

Figura 2-23. Modelo GPC.

Donde están representadas la señal de control y la salida del proceso ($u(t)$ e $y(t)$ respectivamente), el retardo del proceso d, y donde los polinomios A, B y C tendrán la siguiente forma:

$$\begin{aligned} A(z^{-1}) &= 1 + a_1 z^{-1} + a_2 z^{-2} + \ldots + a_{na} z^{-na} \\ B(z^{-1}) &= b_0 + b_1 z^{-1} + b_2 z^{-2} + \ldots + b_{nb} z^{-nb} \\ C(z^{-1}) &= 1 + c_1 z^{-1} + a_2 z^{-2} + \ldots + c_{nc} z^{-nc} \end{aligned}$$

Figura 2-24. Modelo basado en función de transferencia.

El modelo es conocido como CARMA (*Controlled Auto-Regresive Moving-Average*, Autorregresivo de Media Móvil), aunque en algunas aplicaciones industriales, puede ser más conveniente el uso de un modelo CARMA integrado, o también llamada, modelo CARIMA (*Controlled Auto-Regresive Integrated Moving-Average*), el cual simplemente añadiría el término de un integrador a la formula anterior (Figura *2-22*).

Como se comentó anteriormente, la idea básica del algoritmo es la obtención de una serie de señales de control a través de la minimización de una función de coste, la cual de forma general tendrá la siguiente forma:

$$J(N_1, N_2, Nu) = \sum_{j=N_1}^{N_2} \delta(j)[\hat{y}(t+j \mid t) - w(t+j)]^2 + \sum_{j=1}^{Nu} \lambda(j)[\triangle u(t+j-1)]^2$$

Figura 2-25. Función objetivo general GPC.

Donde se tiene como primer término, el error cuadrático (diferencia entre la predicción de la salida del proceso, calculada para el instante de tiempo t, y el vector de referencias futuras) a lo largo del horizonte de predicción, desde N1 hasta N2, partiendo del instante conocido t. El esfuerzo de control sería el segundo término de esta función, calculado a lo largo del horizonte de control, Nu. Ambos términos están ponderados por dos secuencias, $\delta(j)$ y $\lambda(j)$, las cuales ponderan el error cuadrático y el esfuerzo de control respectivamente. Estos parámetros servirán a la hora de sintonizar el controlador, aunque en un gran número de situaciones se suelen considerar valores de δ igual a la unidad y unos valores de λ constantes.

4.2.4.2 Obtención de la predicción óptima:

En la presentación de esta estrategia de control ya se comentó que la obtención de la ley de control se basa en la resolución de cierta ecuación diofántica, esta ecuación se considera de la siguiente forma:

$$\begin{aligned} 1 &= E_j(z^{-1}) \triangle A + z^{-j} F_j(z^{-1}) \\ 1 &= E_j(z^{-1}) \tilde{A} + z^{-j} F_j(z^{-1}) \end{aligned}$$

Figura 2-26. Ecuación diofántica.

El primer paso será la obtención de los polinomios E y F, definidos con los grados $(j-1)$ y n_A (grado del polinomio A del modelo. Pudiendo ser calculados a través de la división de 1 entre la convolución de un polinomio integrador y el polinomio $A(z^{-1})$ (al que se denominará $\tilde{A}(z^{-1})$ (Figura *2-26*), a través del resto se podrá obtener el polinomio F, y el polinomio E serán los sucesivos cocientes de la división.

A partir de este punto se tomará la convención de asignar el valor 1 al polinomio $C(z^{-1})$, por simplicidad. Partiendo entonces de la ecuación con la simplificación que se acaba de mencionar, se van a multiplicar ambos miembros por el término $E_j(z^{-1})z^j\Delta$.

$$\tilde{A}(z^{-1})E_j(z^{-1})y(t+j) = E_j(z^{-1})B(z^{-1}) \triangle u(t+j-d-1) + E_j(z^{-1})e(t+j)$$

$$\Downarrow$$

$$(1 - z^{-j}F_j(z^{-1}))y(t+j) = E_j(z^{-1})B(z^{-1}) \triangle u(t+j-d-1) + E_j(z^{-1})e(t+j)$$

Figura 2-27. Multiplicación de la ecuación diofántica desarrollo.

Si se despeja a través de la ecuación diofántica (Figura *2-26*) y se sustituye (Figura *2-27*, inferior). Se puede reescribir la ecuación de la siguiente forma:

$$y(t+j) = F_j(z^{-1})y(t) + E_j(z^{-1})B(z^{-1}) \triangle u(t+j-d-1) + E_j(z^{-1})e(t+j)$$

Figura 2-28. Salida despejada en la ecuación diofántica.

En esta ecuación se observa que los términos referentes al ruido están todos en el futuro, por lo que la mejor predicción para la salida basándose en el instante t (si se nombra una variable $G_j(z^{-1}) = E_j(z^{-1})B(z^{-1})$) será:

$$\hat{y}(t+j \mid t) = G_j(z^{-1}) \triangle u(t+j-d-1) + F_j(z^{-1})y(t)$$

Figura 2-29. Ecuación de predicción GPC.

4.2.4.3 Obtención de los polinomios E, F y G de forma recursiva:

La demostración de que los polinomios E_j y F_j se pueden obtener recursivamente pasa por corroborar que, los valores de los polinomios en el paso $j+1$ (es decir los polinomios E_{j+1} y F_{j+1}) sean función de los polinomios del paso anterior, j.

Partiendo de que los polinomios E_j y F_j se han obtenido dividiendo 1 entre $\tilde{A}(z^{-1})$ hasta que el resto haya sido factorizado como $z^{-j}F_j(z^{-1})$, con:

$$\begin{aligned} F_j(z^{-1}) &= f_{j,0} + f_{j,1}z^{-1} + \cdots + f_{j,na}z^{-na} \\ E_j(z^{-1}) &= e_{j,0} + e_{j,1}z^{-1} + \cdots + e_{j,j-1}z^{-(j-1)} \end{aligned}$$

Figura 2-30. Formato vectores ecuación diofántica.

Continuando con este procedimiento con el objetivo de obtener los siguiente valores, E_{j+1} y F_{j+1}, dividiendo de igual manera 1 entre $\tilde{A}(z^{-1})$ hasta que el resto se pueda factorizar, en este caso, $z^{-(j+1)}F_{j+1}(z^{-1})$. Obteniendo un resultado muy similar al paso anterior para F_j (Figura *2-31*, superior). Por lo que queda claro que para obtener los sucesivos polinomios se deberá dar esa misma cantidad de pasos en la división. Después de dar el siguiente paso, E_{j+1} será el nuevo cociente de la división, el cual será igual al cociente que había en el anterior paso más un nuevo término (Figura *2-31*, inferior).

$$F_{j+1}(z^{-1}) = f_{j+1,0} + f_{j+1,1}z^{-1} + \cdots + f_{j+1,na}z^{-na}$$

$$E_{j+1}(z^{-1}) = E_j(z^{-1}) + e_{j+1,j}z^{-j} \qquad \text{con} \quad e_{j+1,j} = f_{j,0}$$

Figura 2-31. Obtención sucesiva polinomios ecuación diofántica.

Y por último, teniendo en cuenta que el nuevo resto será el resto anterior menos el producto del cociente por el divisor, los coeficientes del polinomio se pueden expresar de la siguiente forma para su cálculo recursivo:

$$f_{j+1,i} = f_{j,i+1} - f_{j,0}\,\tilde{a}_{i+1} \quad i = 0 \cdots na$$

Figura 2-32. Obtención sucesiva F.

Por ello, como se anticipó en la breve introducción que se hizo sobre los polinomios E y F, los sucesivos resultados para los polinomios resultarán a través de los cocientes y los restos sucesivos de la división:

- Partiendo de los elementos, $E_1 = 1; F_1 = z(1 - \tilde{A})$.
- Añadir término al nuevo E_j, correspondiendo al primer valor del anterior polinomio F. $e_{j+1,j} = f_{j,0}$ (Lo cual sería homólogo a realizar la división entre E_1 y F_1).
- Calcular el F_j correspondiente, a través de la anterior expresión (Figura *2-32*).

Así se demuestra la recursividad del cálculo de la ecuación diofántica, aunque es interesante destacar que existen otras formulaciones del GPC que no están basadas en la recursividad de esta ecuación, aunque no se tratarán en este supuesto

Partiendo del conocimiento sobre la obtención del polinomio G_j (Figura *2-29*) y con lo expuesto en el apartado anterior, el cálculo del polinomio G_{j+1} se podrá expresar como:

$$G_{j+1} = E_{j+1}B = (E_j + f_{j,0}z^{-j})B = G_j + f_{j,0}z^{-j}B$$

Figura 2-33. Obtención sucesiva G (1).

A la vista de esto, los primeros coeficientes de G_{j+1} serán los mismos que G_j, mientras que los demás serán (Figura *2-34*) los correspondientes valores de g del polinomio anterior más el primer valor de F por cada valor de $B(z^{-1})$) hasta completar el polinomio.

$$g_{j+1,j+i} = g_{j,j+i} + f_{j,0}\ b_i \qquad \text{para } i = 0 \cdots nb$$

Figura 2-34. Obtención recursiva G (2).

4.2.4.4 Obtención de la ley de control:

La ecuación de predicción (Figura *2-29*) puede ser agrupada en conjuntos de cada uno de los parámetros dentro de la ecuación de predicciones óptimas (además se puede generalizar la expresión añadiendo un retardo al proceso de d períodos de muestreo, haciendo que la salida se empiece a ver únicamente afectada a partir del instante $d + 1$).

$$\begin{bmatrix} \hat{y}(t+d+1 \mid t) \\ \hat{y}(t+d+2 \mid t) \\ \vdots \\ \hat{y}(t+d+N \mid t) \end{bmatrix} = \mathbf{G} \begin{bmatrix} \triangle u(t) \\ \triangle u(t+1) \\ \vdots \\ \triangle u(t+N-1) \end{bmatrix} + \begin{bmatrix} F_{d+1}(z^{-1}) \\ F_{d+2}(z^{-1}) \\ \vdots \\ F_{d+N}(z^{-1}) \end{bmatrix} y(t) + \mathbf{G}'(z^{-1}) \triangle u(t-1)$$

Para:

$$\mathbf{G} = \begin{bmatrix} g_0 & 0 & \dots & 0 \\ g_1 & g_0 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ g_{N-1} & g_{N-2} & \dots & g_0 \end{bmatrix} \quad \mathbf{G}'(z^{-1}) = \begin{bmatrix} z(G_{d+1}(z^{-1}) - g_0) \\ z^2(G_{d+2}(z^{-1}) - g_0 - g_1 z^{-1}) \\ \vdots \\ z^N(G_{d+N}(z^{-1}) - g_0 - g_1 z^{-1} - \cdots - g_{N-1}z^{-(N-1)}) \end{bmatrix}$$

Figura 2-35. Ecuación de predicción matricial GPC.

Pudiendo escribir la ecuación de forma más simplificada, dividiendo los grupos de términos que dependen sólo del pasado (a lo que se denominará respuesta libre, f), agrupándolos en una matriz de la siguiente forma:

$$\mathbf{y} = \mathbf{G}\mathbf{u} + \mathbf{F}(z^{-1})y(t) + \mathbf{G}'(z^{-1}) \triangle u(t-1)$$

$$\Downarrow$$

$$\mathbf{y} = \mathbf{G}\mathbf{u} + \mathbf{f}$$

Figura 2-36. Ecuación de predicción GPC simplificada.

A continuación se reescribirá la función objetivo sustituyendo en ella la ecuación general de predicción en forma matricial que se acaba de calcular, obteniéndose el siguiente resultado:

$$J = (\mathbf{G}\mathbf{u} + \mathbf{f} - \mathbf{w})^T(\mathbf{G}\mathbf{u} + \mathbf{f} - \mathbf{w}) + \lambda\mathbf{u}^T\mathbf{u}$$

Figura 2-37. Función objetivo matricial GPC.

Desarrollando el término que está multiplicando dentro de la ecuación, y agrupando los términos diferenciando en el caso en que dependan del vector de acciones de control cuadráticas, del vector acciones de control pero de forma lineal o no dependan de ninguna de las anteriores, se obtiene la siguiente función objetivo, donde los términos agrupados por simplicidad (H, b, f_0) serán los siguientes:

$$J = \frac{1}{2}\mathbf{u}^T\mathbf{H}\mathbf{u} + \mathbf{b}\mathbf{u} + \mathbf{f}_0$$

para

$$\begin{aligned} \mathbf{H} &= 2(\mathbf{G}^T\mathbf{G} + \lambda\mathbf{I}) \\ \mathbf{b} &= 2(\mathbf{f} - \mathbf{w})^T\mathbf{G} \\ \mathbf{f}_0 &= (\mathbf{f} - \mathbf{w})^T(\mathbf{f} - \mathbf{w}) \end{aligned}$$

Figura 2-38. Desarrollo función objetivo GPC.

Como anteriormente, si se plantea la convención para un modelo sin restricciones en las señales de control, la minimización de la función J tendrá como resultado:

$$\mathbf{u} = -\mathbf{H}^{-1}\mathbf{b}^T = (\mathbf{G}^T\mathbf{G} + \lambda\mathbf{I})^{-1}\ \mathbf{G}^T(\mathbf{w} - \mathbf{f})$$

Figura 2-39. Minimización función objetivo GPC sin restricciones.

Con las ecuaciones más relevantes para la creación de un GPC ya expuestas, ya se podría dar algunas pinceladas sobre lo que sería la ampliación de lo anterior para el caso multivariable. De forma muy similar a lo que se presentó para un DMC multivariable, en este caso se partirá de un modelo CARIMA (Figura *2-22*) para un sistema con un número m de entradas y n de salidas, donde los anteriores polinomios $A(z^{-1})$, $B(z^{-1})$) y $C(z^{-1})$, ahora serán matrices polinómicas de dimensión NxN, para el caso de $A(z^{-1})$ y $C(z^{-1})$, y de dimensión n x m para $B(z^{-1})$:

$$
\begin{aligned}
\mathbf{A}(z^{-1}) &= I_{n\times n} + A_1 z^{-1} + A_2 z^{-2} + \cdots + A_{n_a} z^{-n_a} \\
\mathbf{B}(z^{-1}) &= B_0 + B_1 z^{-1} + B_2 z^{-2} + \cdots + B_{n_b} z^{-n_b} \\
\mathbf{C}(z^{-1}) &= I_{n\times n} + C_1 z^{-1} + C_2 z^{-2} + \cdots + C_{n_c} z^{-n_c}
\end{aligned}
$$

Figura 2-40. Modelo CARIMA para sistema multivariable.

La ecuación de predicción parte de la resolución de una ecuación diofántica, de manera homóloga al caso monovariable, pero ésta tendrá forma matricial, aunque sigue permitiendo el cálculo recursivo para su resolución. La ecuación objetivo de forma general tendrá la siguiente forma:

$$
J(N_1, N_2, N_3) = \sum_{j=N_1}^{N_2} \|\hat{y}(t+j \mid t) - w(t+j)\|_R^2 + \sum_{j=1}^{N_3} \| \triangle u(t+j-1)\|_Q^2
$$

Figura 2-41. Función objetivo GPC multivariable.

Donde las variables nuevas que aparecen, R y Q, serán los coeficientes de ponderación del caso multivariable, pero en este caso convertidos en matrices de ponderación definidas positivas, las cuales de forma habitual se toman como matrices diagonales.

4.3 Elección del control predictivo generalizado (GPC, Generalized Predictive Control):

El objetivo de toda la anterior exposición sobre las diferentes estrategias de control predictivo, además de dar contexto sobre las posibilidades que se tienen a disposición, y las diferencias que existen para abordar un tipo de control basado en ideas generales concisas, es principalmente la búsqueda de un tipo de estrategia que cumpla las necesidades que deben ser cubiertas por el controlador en cuestión:

- Posibilidad de utilización para gran variedad de procesos.

El objetivo del controlador que se busca desarrollar en este trabajo no va dirigido a un tipo de proceso concreto, sino que lo que se busca es la posibilidad de implementación en diferentes tipos de procesos, por lo que la adaptabilidad del mismo a la mayor cantidad de sistemas reales posibles es una de las características principales.

- Tipo de modelo sencillo y extensible a diversidad de procesos.

La mayoría de procesos SISO, cuando se considera su espectro de trabajo en torno a un punto de equilibrio, pueden ser descritas por un modelo general relativamente sencillo (Figura *2-24*). Lo cual permite la extensión de la aplicación que se busca desarrollar, y su realización de manera más sencilla, que los diferentes modelos que se puedan hacer con las demás estrategias expuestas anteriormente, como pueden ser modelos de respuesta impulsional o ante escalón.

- Facilidad para un completo cálculo recursivo.

Partiendo del hecho de que la programación completa del controlador para el proceso va a realizarse de forma recursiva, es totalmente imprescindible la adaptación de la estrategia teórica, de entre las representadas anteriormente, ofrezca las mayores facilidades ante la implementación de esta dentro de un código ejecutado consecutivamente. En primer lugar el cálculo de la predicción de la salida, realizándose recursivamente a través del modelo y de la resolución de una ecuación diofántica ya definida, promueve a la estrategia de Control Predictivo Generalizado, una de las opciones que mejor cumple este objetivo. Y en segundo lugar, la utilización más eficiente para el cálculo en formato matricial de las estrategias DMC y GPC, lo cual ayudará en gran medida a la creación del propio algoritmo de control, son las que se promueven como mejores alternativas para facilitar este cálculo.

- Sencillo cálculo matricial y extensión al caso multivariable.

De forma similar en varias de las estrategias de control expuestas, se busca obtener una ecuación para la predicción de la salida lo más simplificada posible (Figuras *2-11*, y *2-36*), dividiendo la ecuación únicamente en una parte referente a la respuesta forzada y otra con la respuesta libre, permitiendo un sencillo cálculo matricial, lo que se puede traducir en, primer lugar, no demasiada dificultad para la realización del propio cálculo recursivo del algoritmo, y además la sencilla ampliación al caso multivariable si se necesitase, sin dejar de lado que esta ampliación en el cálculo tiene intrínsecamente una importante complejidad.

A la vista de las necesidades que se acaban de exponer, es necesario tomar la decisión sobre el tipo de estrategia de control que cumpla, o potencialmente pueda cumplir, la mayoría de requerimientos, para este caso se tomará el Control Predictivo Generalizado (GPC), como la estrategia de control a implementar.

4.4 Formulación práctica del cálculo recursivo de un GPC:

En este apartado, se abordarán todos los pasos necesarios que hay que dar para desarrollar un código que implemente el algoritmo de un Control Predictivo Generalizado programado en lenguaje C. Para esto se puede proceder de varias formas:

- Realizar directamente el desarrollo del algoritmo en C.

Si se opta por desarrollar directamente el código en el lenguaje final, se tiene el problema de que la ejecución de los códigos en C no es tan rápida e intuitiva como podrían ser otros lenguajes o plataformas, al necesitar el uso de funciones exclusivas y bucles completos sólo para la visualización de valores por pantalla, lo que puede hacer bastante más tedioso el desarrollo, y más lento el proceso de prueba. Además de no tener implementadas más tipos de operaciones que las básicas, haciendo, por ejemplo, que el cálculo matricial necesite de algoritmos creados desde cero para realizar operaciones tan sencillas como el producto entre matrices.

Aunque, como contraparte, si se comienza el desarrollo en el lenguaje final que va a poseer el trabajo no habría que realizar portabilidades entre lenguajes, que en el caso en el que no se conozcan exactamente las diferencias entre los diferentes lenguajes que se utilicen, o no se tenga soltura trabajando con los mismos, puede resultar en multitud de errores de programación y, por ello, en un gran aumento de tiempo de desarrollo.

- Utilizar una plataforma de apoyo para el desarrollo del algoritmo.

Si se decide utilizar una plataforma auxiliar para el desarrollo de cualquier algoritmo con la necesidad, posteriormente al desarrollo, de realizar la portabilidad del código creado a un lenguaje de programación diferente, sería muy recomendable tener buenos conocimientos sobre las diferencias principales entre ambos lenguajes, con el objetivo de optimizar el tiempo de depuración del código durante la portabilidad de lenguajes, ya que en el caso de que no se conozcan en exceso, el resultado, en cuanto a tiempo invertido, podría ser similar a desarrollar dos algoritmos iguales para dos lenguajes de programación diferentes, haciendo esta alternativa mucho más ineficiente que la anterior.

Expuestas las alternativas a la hora de enfocar la programación del algoritmo completo, es imprescindible mencionar que el primer paso, previo a cualquier cálculo referente a la estrategia de control predictivo que se elija, será siempre el cálculo del modelo del proceso a través del método que sea necesario, en el caso del GPC se utilizará un modelo en forma de función de transferencia, para los cálculos que se hagan a continuación se tomará un modelo sencillo de ejemplo, utilizando una función de transferencia de primer orden en los puntos del cálculo en el que haga falta, aunque para el desarrollo del propio algoritmo no es necesario un modelo de ejemplo ya que todo estará planteado de forma genérica, por este motivo se abordará el apartado sobre el cálculo del modelo del proceso cuando se llegue a la implementación en el propio sistema físico, donde sí será totalmente imprescindible su cálculo.

4.4.1 Resolución de la ecuación diofántica:

Con el fin de generalizar las diferentes posibilidades de programación, únicamente en este y sólo por ser el primero de los apartados de desarrollo, se procederá realizando la resolución de la ecuación diofántica en primer lugar, utilizando un entorno de programación auxiliar, para el cual se elegirá Matlab, por su facilidad a la hora de ejecutar los diferentes códigos y la obtención de resultados por pantalla de forma casi directa, lo que permitirá que las pruebas y la posterior depuración del código sean mucho más eficientes. Y a continuación se realizará la portabilidad del código resultante anterior a lenguaje C, dentro del entorno Visual Studio al que ya se ha tenido cierto acercamiento en apartados anteriores.

4.4.1.1 Resolución de la ecuación diofántica en Matlab:

En primer lugar si se cree necesaria la existencia de un modelo de ejemplo para realizar las pruebas y tener resultados para cada parte del código que se quiera probar, si se parte del caso más primario, en el que se tienen los datos sobre un modelo temporal de primer orden (K y tau del sistema), se debe introducir la función de transferencia del sistema en tiempo discreto y con factores autorregresivos. Para ello en Matlab se definirá la función de transferencia en tiempo continuo a través del comando “tf”, introduciendo los valores para el numerador y el denominador, a continuación se pasará la función a tiempo discreto con el comando “c2d”, introduciendo en el mismo el tiempo de muestreo que se haya elegido y pueda representar debidamente las dinámicas del proceso. Por último se deben cambiar la función de transferencia a términos autorregresivos, utilizando el comando “filt”, introduciendo como términos el numerador y el denominador de la función de transferencia en tiempo discreto que se acaba de calcular y otra vez más el tiempo de muestreo elegido.

Matlab trata, en pos de la facilidad de cálculo en cuestiones de control, el denominador y el numerador de las funciones de transferencia como vectores con únicamente los valores de los coeficientes, lo que ayuda al desarrollo del cálculo, ya que es lo único que se necesita para realizar todos los cálculos posteriores, por ello con que se tengan los datos de estos coeficientes todo el cálculo que se acaba de comentar sería omisible.

- Inicialización de las variables de control conocidas:

Obviando el cálculo previamente explicado, se tomarán directamente unos polinomios de ejemplo para el modelo del proceso, y además se definirán en este apartado los valores necesarios para el cálculo, que se deben imponer en base al conocimiento que se tenga del sistema y las necesidades que se quiere que cumpla el control, como serán los diferentes horizontes y los coeficientes de ponderación necesarios (Figura *2-42*,).

```
%% Cálculo realizado conociendo los polinomios de G(z-1)

A = [1 -0.8];
B = [0.4 0.6];

%Ventana de prediccion
Nu = 3;      %Horizonte de control
N1 = 1;      %Horizonte Inicial
N2= 3;       %Horizonte Final

%Coeficientes de ponderación
lambda=0.5;
```

Figura 2-42. Inicialización cálculo recursivo ecuación diofántica en Matlab.

- Cálculo de las matrices E y F:

Partiendo de la anteriormente expuesta ecuación diofántica y su proceso de resolución recursivo, se debe comenzar realizando la convolución entre el polinomio A y un polinomio integrador en discreto (resultado al que se nombrará como AD). A partir de aquí se podría proceder construyendo una división entre la unidad y el resultado de la convolución anterior, los sucesivos cocientes de la división irán conformando la matriz E y los sucesivos restos la matriz F.

Para la construcción de la matriz F, previamente a la realización de ningún cálculo, se inicializará una matriz con las dimensiones conocidas de la matriz F completa (en función del horizonte final y la dimensión de A) rellena de ceros. Como paso inicial la primera fila siempre se completará con los valores, desde el segundo hasta el final, del polinomio AD cambiados de signo, lo que coincide con el resto de la primera división entre la unidad y el polinomio AD. En segundo lugar se irán rellenando los siguientes valores de la matriz, a través del bucle presentado a continuación (Figura *2-43*), de uno en uno completando las filas de izquierda a derecha.

```
%% Cálculo de la ecuación diofántica 1 = E*AD+(z^-j)*F
delta = [1 -1];
AD = conv(A,delta);

%Los polinomios E y F pueden ser obtenidos dividiendo 1 y AD

nA = size(AD);nA = nA(2);
nB = size(B);nB = nB(2);

%% Matriz F
F = zeros(N2,nA-1);
F(1,:) = -AD(2:nA);

for i = 1:N2-1
    for j = 1:nA-2
        F(i+1,j) = F(i,j+1) - F(i,1)*AD(j+1);
    end
    F(i+1,2) = -F(i,1)*AD(nA);
end
```

Figura 2-43. Cáculo recursivo matriz F en Matlab.

Para la construcción de la matriz E, de manera homóloga al caso anterior, se inicializará una matriz con las dimensiones conocidas de la matriz resultante E (únicamente en función del valor del horizonte final, formando así una matriz cuadrada), inicializando toda la primera columna con el valor 1, ya que el resultado de la matriz E serán los cocientes sucesivos de la división antes presentada, y el primero siempre será la unidad. A continuación para rellenar los términos de las siguientes filas de la matriz (ya que la primera queda completada con un solo valor), se rellenarán las siguientes columnas con los valores de la primera columna de la matriz F, hasta completar la matriz E, resultando una matriz triangular inferior.

```
%% Matriz E
E = zeros(N2,N2);E(:,1) = 1;
for i = 2: N2
    E(i:N2,i)=F(i-1,1);
end
```

Figura 2-44. Cálculo recursivo matriz E en Matlab.

- Cálculo de la matriz G perteneciente a la respuesta forzada

A continuación se tratará la construcción de la matriz G representada en la ecuación final de la función objetivo simplificada (Figura *2-36*), aunque este apartado no está dentro de la propia resolución de la ecuación diofántica, se va a incluir esta parte del desarrollo del algoritmo al ser una consecuencia directa de los cálculos anteriores, y para de forma posterior, tener un código completo que proporcione el resultado de las principales matrices del cálculo de un GPC, ante unos valores de entrada de un modelo en función de transferencia genérico.

En primer lugar se inicializará una matriz auxiliar, llamada "*Gaux*", que albergará la convolución entre la matriz E y B, lo que se conoce como los valores que tendrá la matriz G (Figura *2-45*), aunque habrá valores que no se usarán dentro de la matriz y los restantes se deberán recolocar para obtener el resultado final de la matriz G.

```
Gaux=zeros(N2,N2);
for i=0:N2-1
 Gaux(i+1,1:nB+i) = conv(E(i+1,1:nB+i-1),B);
end
```

Figura 2-45. Convolución preparatoria para calcular G en Matlab.

Para la obtención final de la matriz, en primer lugar se separarán los valores de la matriz Gaux que no pertenecen a la matriz G, correspondientes al último valor de cada una de las filas, guardando los mismos dentro de un vector al que se llamará "*Gf*", al cual se le dará uso a la hora de realizar el cálculo de los términos de la función objetivo.

```
%Extracción de los valores no contenidos en G para la matriz Gf
Gf=zeros(N2,1);
for i=1:N2+1
    for j=1:N2+1
        if (j>i) && (j-i==1)
            Gf(i,1)=Gaux(i,j);
            Gaux(i,j)=0;
        end
    end
end
Gaux = Gaux(:,1:N2);
```

Figura 2-46. Cálculo previo matrices GPC en Matlab.

Por último, la matriz resultado de la extracción anterior debe ser reorganizada para ser la matriz G final, ya que la forma anterior está organiza los valores por filas y deberán estar organizados en las diagonales inferiores de la matriz, a través del siguiente bucle:

```
%Se reorganizan los términos de G
G=zeros(N2,N2);
for i=1:N2
  k=1;
  for j=i:-1:1
  G(i,k)=Gaux(i,j);
  k=k+1;
  end
end
```

Figura 2-47. Reorganización términos G definitiva en Matlab.

- Cálculo de la matriz G perteneciente a la respuesta libre:

Esta matriz para ser diferenciada de la anterior será denominada "*Gf*", la cual no está representada en las ecuaciones que se han desarrollado hasta ahora, por lo que esta variable será una variable auxiliar únicamente utilizada para el cálculo de la respuesta libre dentro de la ecuación final del GPC (variable f), donde para formar esta variable se realizará a través de la matriz F y Gf, por ello se ha añadido este cálculo dentro del apartado de la resolución de la ecuación diofántica, a pesar de no pertenecer a la propia resolución.

Esta matriz, buscando la optimización de código, fue calculada en el apartado anterior, a la vez que la propia matriz G, a través de la extracción de los valores que no correspondía a ésta, gracias al resultado de la convolución de las matrices E y B (Figura *2-46*).

4.4.1.2 Requisitos para la portabilidad del código del cálculo de la ecuación diofántica a lenguaje C:

Previamente a la realización de la portabilidad del código desde el algoritmo de resolución creado en el anterior apartado, es totalmente necesaria la creación de tres funciones:

- Función de reserva de memoria dinámica para vectores.
- Función de reserva de memoria dinámica para matrices.

Estas dos funciones serán las más recurrentes en el código, ya que en ningún caso en el que se inicialice una variable que contenga un vector o una matriz, se hará conociendo el valor numérico exacto de filas y columnas de esta variable, si no que se recibirá una variable que contenga ese valor numérico, lo cual hace que sea imprescindible la utilización de memoria dinámica para la inicialización de todas estas variables.

Por esto y por el hecho de que sea necesario el uso de inicialización de variables una gran cantidad de veces a lo largo del código, se crearán dos funciones explícitamente para esta tarea llamadas, "*reservarVEC*" y "*reservarMAT*", para reserva de memoria dinámica para vectores y matrices respectivamente, no son funciones necesarias para el funcionamiento del código, pero evitan la repetición de bloques de código un gran número de veces y hará el código más modular y eficiente en su ejecución.

```c
//FUNCIONES DE RESERVA DE ESPACIO EN MEMORIA PARA MATRICES Y VECTORES RESPECTIVAMENTE

double** reservarMAT(int filas, int columnas) {

    double **mat;
    int i;

    mat = (double**)calloc(filas, sizeof(double*));
    if (mat == NULL) {
        printf("No se ha podido reservar memoria.");
        exit(1);
    }
    for (i = 0; i < filas; i++) {
        mat[i] = (double*)calloc(columnas, sizeof(double));
        if (mat[i] == NULL) {
            printf("No se ha podido reservar memoria.");
            exit(1);
        }
    }

    return mat;
}

double* reservarVEC(int filas) {

    double *vec;

    vec = (double*)calloc(filas, sizeof(double));
    if (vec == NULL) {
        printf("No se ha podido reservar memoria.");
        exit(1);
    }


    return vec;
}
```

Figura 2-48. Funciones de reserva de memoria dinámica en C.

- Función para el cálculo de la operación convolución.

En este caso, previamente expuesto en el cálculo anterior, la función que sí será necesario crear expresamente para la posible realización del código, será la función de convolución entre dos vectores, ya que el lenguaje C no da la posibilidad más que de realizar las operaciones matemáticas básicas. A continuación se puede observar el código para la creación de esta función, al contrario que en las dos funciones anteriores, donde únicamente se le tenían que pasar los valores del número de filas o columnas según fuera necesario, en este caso se tienen una mayor cantidad de variables a introducir en la llamada de la función.

```
//FUNCIÓN DE CONVOLUCIÓN DE VECTORES

void convolucion(int NA, int NB, double A[], double B[], double *Res) {

    int Naux = NA + NB - 1;
    int i, j, k;

    double **Maux;
    Maux = reservarMAT(NB + 1, Naux);


    k = 0;
    for (i = 0;i < NB;i++) {
        for (j = 0;j < NA;j++) {
            Maux[i][j + k] = B[i] * A[j];
        }
        k += 1;
    }


    for (i = 0;i < NB;i++) {
        for (j = 0;j < Naux;j++) {
            Maux[i + 1][j] = Maux[i + 1][j] + Maux[i][j];
        }
    }


    for (i = 0;i < Naux;i++) {
        Res[i] = Maux[NB][i];
    }

}
```

Figura 2-49. Función de convolución de vectores en C.

Como las funciones son porciones de código separadas unas de otras, es necesario darle a conocer a la función de llamada todos los parámetros que se tienen en la función principal del código, y le serán necesarios para la ejecución como, las longitudes de los dos vectores que se van a operar, los dos vectores propiamente inicializados y el valor del vector resultado, el cual, a menos que sea un caso especial, en todas las funciones que se realicen se pasarán por referencia.

En definitiva, las primeras dos funciones no son necesarias para realizar un código funcional, pero ayudará a crear un código modular, ordenado y más eficiente, además de simplificar el mismo, al añadir una función que realizará una tarea que será necesario que se repita un gran número de veces en una sola ejecución del código. En cambio la tercera función en cuestión será una de las varias que se tengan que crear para suplir las carencias del lenguaje C en cuestión de, operaciones entre vectores y matrices y sus combinaciones.

4.4.1.3 Portabilidad del código de resolución de la ecuación diofántica a lenguaje C:

Con las funciones necesarias para el código en C es importante conocer las diferencias entre los lenguajes utilizados, ya que en caso contrario sería lo mismo que re hacer el código desde cero otra vez. Principalmente ambos lenguajes difieren en, la forma de declaración de variables, el uso general de los bucles, las opciones de Matlab para cálculo matricial, los índices de inicio de una variable y por último se puede destacar la forma de trabajar con funciones, ya que Matlab utiliza funciones que son códigos externos al propio código que se está desarrollando, en cambio en C, al margen de que también se pueden compilar códigos externos, se tiene la opción de agregar funciones que realicen diferentes procesos dentro de un mismo código, por lo que esto será lo que se realice a continuación, una función cuyo fin sea la resolución de la ecuación diofántica y la obtención de las matrices G y Gf anteriormente calculadas.

- Inicialización de las variables de control conocidas:

La inicialización de las variables se realizará en una función principal en el que se simulará la obtención de las variables del modelo (vectores A y B), y las dimensiones de los horizontes de control y horizonte final, extrayendo de esos datos las dimensiones de ambos vectores, y por último inicializando las variables que contendrán a G y Gf, posteriormente llamando a la función que realizará todos los cálculos.

```
//Horizonte final y de control
int N2 = 3; int Nu = 3;

//Valores del modelo del proceso en forma de función de transferencia
double A[] = { 1, -0.8 }; double B[] = { 0.4, 0.6 };
int nA = 2; int nB = 2;

//Inicialización de las matrices resultado
double **G;
G = reservarMAT(N2, Nu);

double **Gf;
Gf = reservarMAT(N2, (nA + nB - 1));
```

Figura 2-50. Inicialización valores de entrada y matrices de salida.

- Cálculo de las matrices F y E:

De manera homóloga se abordará el cálculo de las diferentes matrices en orden, en primer lugar se definirán los valores previos para la realización de la convolución que inicia la resolución de la ecuación diofántica (Figura *2-51*), para de forma seguida realizar el cálculo de la matriz F de forma exactamente igual a la programada en Matlab, excepto que los valores de los índices numéricos serán un valor una unidad menor, además de la forma completamente distinta de inicialización utilizando la función "*reservarMAT*".

```
int i, j, k;
int nA = na + 1;

double delta[] = { 1, -1 };

double *AD;
AD = reservarVEC(nA);

convolucion(na, 2, AA, delta, AD);


//CALCULO DE LA MATRIZ F

double **F;
F = reservarMAT(N2, nA - 1);

//Calculo del valor de la primera fila de F (obtenido de dividir AD entre 1)

for (i = 1;i < nA;i++) {
    F[0][i - 1] = -AD[i];
}


for (i = 0;i < N2 - 1;i++) {
    for (j = 0;j <= nA - 2;j++) {
        F[i + 1][j] = F[i][j + 1] - F[i][j] * AD[j + 1];

    }
    F[i + 1][1] = -F[i][0] * AD[nA - 1];
}
```

Figura 2-51. Inicialización y cálculo recursivo F en C.

Como segunda parte se calculará la matriz E, procediendo de la misma manera que la matriz anterior, aunque como se ha podido observar en ambas figuras (Figuras *2-51* y *2-52*), se ha tenido que añadir un bucle extra a cada uno de los cálculos de las matrices, esto es debido a lo que ya se ha comentado sobre las facilidades que otros entornos de desarrollo, como Matlab, pueden ofrecer a la hora de realizar ciertos cálculos. En este caso en concreto, la asignación de los valores iniciales a las matrices, de una simple igualdad en la programación anterior (Figuras *2-43* y *2-44*), se convierten en dos bucles, que no por ello hace más compleja la programación, pero sí hace sobresalir las diferencias entre entornos.

```
//CALCULO DE LA MATRIZ E

double **E;
E = reservarMAT(N2, Nu);

//Se inicializa el primer valor del vector que coincide con toda la primera columna
for (i = 0;i < N2;i++) {
    E[i][0] = 1;
}

k = 0;
for (i = 1;i < N2;i++) {
    for (j = 1 + k;j < Nu;j++) {
        E[j][i] = F[k][0];
    }
    k += 1;
}
```

Figura 2-52. Cálculo recursivo matriz E en C.

- Cálculo de las matrices G y Gf:

En primer lugar se inicializará un vector auxiliar, llamada "*convG*", que albergará la convolución entre el vector completo de E y B, lo que se conoce como los valores que tendrá la matriz G (Figura *2-33*), aunque habrá valores que no se usarán dentro de la matriz, por lo que se recolocarán los términos agrupados en el vector que contiene la convolución para obtener el resultado final de la matriz G (Figura *2-53*).

```
//CALCULO DE LA MATRIZ G

//Se reserva espacio
double *convG;//Vector que contendrá la convolucion entre E y B
convG = reservarVEC(Nu + nB - 1);

double *Eaux;//En este vector se guardarán la última componente de E
Eaux = reservarVEC(Nu);

//Extraccion de la componente con todos los valores que tendrá E en todo el horizonte
for (i = 0;i < Nu;i++) {
    Eaux[i] = E[Nu - 1][i];
}


//Calculo de la convolucion de E y B
convolucion(Nu, nB, Eaux, BB, convG);

//Se recolocan los términos de la convolución en las diagonales para obtener G
for (i = 0;i < N2;i++) {
    for (j = 0;j < Nu;j++) {
        if (i - j >= 0) {
            G[i][j] = convG[i - j];
        }
    }
}
```

Figura 2-53. Cálculo recursivo de la matriz G en C.

Como se dijo en el cálculo que se acaba de realizar, existen unos valores de la convolución entre E y B que no serán utilizados para la matriz G, en cambio, serán valores que formarán parte de la respuesta libre, Gf. Estos valores se extraerán realizando las sucesivas convoluciones entre los diferentes valores de E (cada una de sus filas) y B, extrayendo los valores extremos de las mismas y guardándolos en una matriz auxiliar, "*Gfaux*" (Figura 2-54).

```
//CALCULO COMPLETO DE Gf

double *aux;//En este vector guardaremos el valor de la convolución sucesivamente
aux = reservarVEC(Nu + (nB - 1));

double **Gfaux;
Gfaux = reservarMAT(N2, (nB - 1));

for (i = 0;i < N2;i++) {
    for (j = 0;j < Nu;j++) {
        //Se extrae en cada bucle la componente de E correspondiente
        Eaux[j] = E[i][j];
    }
    convolucion(i + nB, nB, Eaux, BB, aux);

    for (j = 0;j < (nB - 1);j++) {
        Gfaux[i][j] = aux[i + 1];
    }
}
```

Figura 2-54. Obtención de los valores de la respuesta libre dentro de G.

Por último se debe colocar la matriz definitiva correspondiente a la respuesta libre, formándose a través del resultado anterior y la matriz F. Sencillamente se debe colocar como bloques de matrices adyacentes, ya que el número de filas será el mismo que el horizonte final de predicción, la matriz Gfaux y la matriz F (Figura *2-55*).

```
//Se colocan en la matriz definitiva para Gf primero Gfaux y luego F

for (i = 0;i < N2;i++) {
    for (j = 0;j < (nB - 1);j++) {
        Gf[i][j] = Gfaux[i][j];
    }


}

//Se completa el vector con la respuesta libre entera
for (i = 0;i < N2;i++) {
    for (j = nB - 1, k = 0;j < (nA + nB - 2), k < (nA - 1);j++, k++) {
        Gf[i][j] = F[i][k];
    }
}
```

Figura 2-55. Colocación definitiva matriz Gf.

El código resultado en Matlab y en C, sobre la resolución de la ecuación diofántica y el cálculo de las matrices principales del GPC, se puede encontrar en los archivos adjuntos a este proyecto, llamados "*DiofanEQmatlab*" y "*GPCscript*", además de poder recurrir al proyecto transcrito en los *Anexos B* y *C* respectivamente.

4.4.2 Desarrollo de funciones para cálculo vectorial y matricial en C:

Realizada toda la preparación para el cálculo del GPC, el proceso que se deberá realizar corresponde al cálculo completo de la función objetivo, pero al requerir de cálculos matriciales será necesaria la creación de ciertas funciones en lenguaje C, con el fin de realizar las operaciones que necesita la función objetivo (Figura *2-38*).

4.4.2.1 Función de trasposición de una matriz:

En primer lugar, al contrario de lo expuesto anteriormente, se realizará un algoritmo para la trasposición de matrices, un proceso diferente a cualquier operación matricial, ya que es una transformación que como resultado tendrá una matriz con sus filas y columnas intercambiadas. Como nomenclatura diferenciadora, una matriz traspuesta, con nombre, por ejemplo "*G*", será denominada a su homóloga traspuesta como, "*GT*".

```
//FUNCION TRASPUESTA DE UNA MATRIZ

void WINAPI traspMAT(double **M, int N1, int N2, double **Res) {

    int i, j;

    for (i = 0; i < N1;i++) {
        for (j = 0;j < N2;j++) {
            Res[j][i] = M[i][j];
        }
    }
}
```

Figura 2-56. Función traspuesta de una matriz en C.

La función que realizará el proceso de trasposición de una matriz tendrá la estructura mostrada en la figura anterior (Figura *2-56*), en la cual, simplemente, a una matriz resultado, definida previamente a la llamada de la función, se le asignan cada uno de los valores dentro de otra matriz con los índices de la matriz intercambiados.

4.4.2.2 Funciones para el producto entre matrices y entre vectores:

Los productos en los que intervienen matrices y vectores se pueden diferenciar en dos casos, donde en el producto sólo interviene uno de los dos o en los que se mezclan ambos elementos. Por ello, y además teniendo en cuenta que el orden del producto de matrices y vectores importa, se pueden dar cuatro tipos de productos diferentes. En este apartado se realizarán funciones para el producto entre dos matrices y entre dos vectores.

- Producto entre dos vectores:

Para realizar un producto entre dos vectores es necesario que previamente se conozca y se comprueben las dimensiones de cada uno de los vectores, ya que es necesario que tengan la misma, para que los vectores sean operables y para obtener como resultado un escalar.

```
double WINAPI multiVEC(double *V1, double *V2, int N) {

    //En esto caso únicamente se necesita una dimensión para ser multiplicables y el resultado es un escalar

    int i;
    double Res = 0;

    for (i = 0;i < N;i++) {
        Res = Res + (V1[i] * V2[i]);
    }
    return Res;
}
```

Figura 2-57. Función producto entre vectores en C.

Por ello se hará una simple función que devuelva un escalar, que sea el resultado de ir sumando los productos de cada una de las posiciones de un vector por su homóloga en el otro (Figura *2-57*)

- Producto entre dos matrices:

Para la realización de un producto entre dos matrices se debe comprobar, para que ambas matrices sean multiplicables, que el número de columnas en la primera matriz sea el mismo que el número de filas de la segunda, dando como resultado una matriz que tuviera como dimensiones, el número de filas de la primera y el número de columnas de la segunda, suponiendo el caso en que dos matrices M1 y M2 multiplicables entre sí den como resultado una matriz M3.

$$M1_{(N1,N2)} * M2_{(N2,N3)} = M3_{(N1,N3)}$$

A la vista de la siguiente función, la matriz resultado será una matriz donde cada término coincide con el sumatorio del producto de los términos de la fila de la primera matriz y columna de la segunda, lo que sería el producto entre dos vectores si se extrajeran de la matriz.

```
void WINAPI multiMAT(double **M1, double **M2, int N1, int N2, int N3, double **Res) {
    //N1 = FILAS M1-----------N2 = COLUMNAS M1-------------N3 = COLUMNAS M2

    //N2 debe ser el número compartido por las matrices para ser operable (COLUMNAS de M1, FILAS de M2)

    int i, j, k;
    double aux;

    for (i = 0; i < N1;i++) {//i para las filas de la matriz Resultado
        for (j = 0;j < N3;j++) {//j para las columnas de la matriz Resultado
            aux = 0;

            for (k = 0;k < N2;k++) {//k para apoyarnos a la hora de hacer la multiplicación
                aux = aux + (M1[i][k] * M2[k][j]);
            }
            Res[i][j] = aux;
        }
    }
}
```

Figura 2-58. Función producto entre matrices en C.

4.4.2.3 Funciones para el producto matriz-vector y vector-matriz:

Por último en este apartado se realizará una función para el producto entre matrices y vectores, pero dado el caso de que en este tipo de operaciones el orden importa se desdoblará en dos casos, cuyo resultado siempre será un vector. Este tipo de multiplicación se podría considerar como dos casos específicos del producto entre dos matrices, en el que, o la primera matriz tiene una fila o la segunda matriz únicamente tiene una columna, pero este caso no se puede aplicar al lenguaje C por la diferencia en la declaración entre matrices y vectores, creando así dos casos nuevos y la necesidad de programación de dos funciones separadas:

- Producto matriz-vector:

Como se está programando en C los vectores no tienen orientación, no es necesario especificar que se debe tener un vector columna para la que multiplicación sea realizable. Entones únicamente se debe comprobar que la segunda dimensión de la matriz (Nº de columnas) sea igual que la longitud del vector.

```
void WINAPI MATxVEC(double **M, int N1, int N2, double *V, double *Res) {

    //En este caso sólo se necesitarán las dimensiones de la matriz ya que las dimensiones del vector serán N1

    int i, j;
    double aux = 0;

    for (i = 0; i < N1;i++) {
        for (j = 0;j < N2;j++) {
            aux = aux + (M[i][j] * V[j]);
        }
        Res[i] = aux;
        aux = 0;
    }
}
```

Figura 2-59. Función producto entre una matriz y un vector en C.

En la programación se obtendrá un vector donde cada una de las posiciones sea el producto de vectores de cada fila de la matriz por el vector de entrada (Figura *2-59*)

- Producto vector-matriz:

En esta ocasión, totalmente opuesto a lo realizado en el caso anterior, se debe comprobar que la primera dimensión de la matriz (Nº de filas) sea igual a la longitud del vector. Y se programe una función exactamente igual que la anterior, a excepción del intercambio de índices y de la posición del producto que esta vez los valores del vector multiplican a los de la matriz (Figura *2-60*).

```
void WINAPI VECxMAT(double **M, int N1, int N2, double *V, double *Res) {

    //En este caso sólo se necesitarán las dimensiones de la matriz ya que las dimensiones del vector serán N2

    int i, j;
    double aux = 0;

    for (j = 0;j < N2;j++) {
        for (i = 0;i < N1;i++) {
            aux = aux + (V[i] * M[i][j]);
        }
        Res[j] = aux;
        aux = 0;
    }
}
```

Figura 2-60. Función producto entre un vector y una matriz en C.

4.4.3 Cálculo diferenciado de los diferentes términos de la función objetivo:

Dispuesta toda la preparación previa para el desarrollo, propiamente dicho, de la función objetivo y su resolución en C, se va a tomar la ecuación de la función objetivo desarrollada anteriormente (Figura *2-38*) sin la aplicación de ninguna convención, para así partir de la forma más general posible.

```
SIENDO EL VECTOR U, VECTOR CON LAS VARIACIONES DE CONTROL DEL HORIZONTE DE PREDICCION NU

Minimizar:      J   =   1/2 * UT * 2(GT * G + lambda*I(NU,NU)) * U    +   2(f - W)T * G * U    +   (f - W)T*  (f - W)

                J   =   1/2 * UT *           H                 * U    +    bT     * G * U      +        f0

                      (-----------TERMINO CUADRÁTICO------------)     (--TERMINO LINEAL--)    (--TERMINO INDEPENDIENTE--)
```

Figura 2-61. Modelo basado en función de transferencia.

Las partes que pueden ser extraídas de la función objetivo están diferenciadas por su dependencia con el vector de variaciones de control (U), teniendo un término independiente, un término lineal y un término cuadrático (Figura *2-61*). Para el cálculo se primará la optimización, abordando los bloques de datos que se desconocen en primer lugar, para luego calcular cada uno de los términos completos en orden.

4.4.3.1 Cálculo del término independiente:

Como único término que se desconoce previamente a este cálculo, el perteneciente a la respuesta libre*(f-w)*, cuya representación está a la vez en el término independiente y en el término lineal, debe ser calculado de forma especial, ya que *f*, representada en el cálculo de la ecuación diofántica por Gf, depende parcialmente de las señales de control pasadas (vector al que se llamará "*ukpast*") y de las salidas pasadas del sistema (ypast), al haberse formado con los valores restantes de G y la matriz F respectivamente.

Por ello, en primer se van a formar cada una de las partes de la matriz que depende del pasado, en el primer bucle la parte dependiente de las señales de control pasadas de la matriz G, y en segundo lugar en otro bucle la parte que depende de las salidas pasadas del sistema obtenidas por F (Figura *2-62*).

```c
//En primer lugar se va a calcular el valor del término "(f-w)" que está presente en los términos LINEAL e INDEPENDIENTE
//La matriz Gf se divide por columnas en primer lugar, (nB - 1) columnas multiplicadas por un término del vector ukpast
for (i = 0;i < (nB - 1);i++) {
    for (j = 0;j < N2;j++) {
        Gf[j][i] = Gf[j][i] * ukpast[i];
    }
}

//Las demás columnas (nA) deben ser multiplicadas por las entradas pasadas, ykpast
//en primera instancia éstas serán siempre el valor Y

for (i = (nB - 1);i < (nA + nB - 1);i++) {
    for (j = 0;j < N2;j++) {
        Gf[j][i] = Gf[j][i] * ypast[i - 1];
    }
}
```

Figura 2-62. Modelo basado en función de transferencia.

Y a continuación se sumarán todos los términos creados, guardando el valor de cada una de las filas en una variable auxiliar ("*f_reslib*"), a la que se le restará el valor correspondiente de la referencia dando como resultado el vector *(f-w)* que se buscaba en principio (Figura *2-63*), denominado en el código "*Gfaux*".

```
//Para finalizar el cálculo se debe sumar entre sí todos los términos de las filas
//Y cada uno restar el valor correspondiente de la referencia, W
double *Gfaux;
Gfaux = reservarVEC(N2);

double f_reslib;
for (i = 0;i < N2;i++) {
    f_reslib = 0;
    for (j = 0;j < (nA + nB - 1);j++) {
        f_reslib += Gf[i][j];
    }
    Gfaux[i] = f_reslib - w[i];
}
```

Figura 2-63. Cálculo término *(f-w)* de la función objetivo.

Ahora todos los elementos que aparecen en la función objetivo son conocidos, aunque algunos dependerán de la información recibida del propio sistema o que requerirán inicialización, pero los cálculos pueden ser programados en su totalidad, y en primer lugar se calculará el término independiente, el cual simplemente será el producto vectorial de *(f-w)* por sí mismo (Figura *2-64*), guardado en la variable "*f0*".

```
/*
    CALCULO DEL TÉRMINO INDEPENDIENTE
*/

double f0 = multiVEC(Gfaux, Gfaux, N2);
```

Figura 2-64. Cálculo del término independiente de la función objetivo.

4.4.3.2 Cálculo del término lineal:

El término lineal depende del vector Gfaux y de la matriz G (Figura *2-61*), dando como resultado un vector que estará en función del vector de variaciones de control a lo largo del horizonte, el cual será el resultado de la optimización, por ello se guardará el vector resultado para luego añadirse a la función de optimización de Gurobi en los términos lineales correspondientes.

```
/*
    CALCULO DE LOS TÉRMINOS LINEALES
*/

//Primero se multiplica Gfaux por 2 y luego lo multiplicaremos por G


for (i = 0;i < N2;i++) {
    Gfaux[i] = Gfaux[i] * 2;//Sobreescribir los valores en el propio vector
}

double *linval;//el vector resultado contendrá los valores que acompañan los términos lineales en orden creciente
linval = reservarVEC(Nu);

VECxMAT(G, N2, Nu, Gfaux, linval);
```

Figura 2-65. Cálculo del término lineal de la función objetivo.

En primer lugar se multiplica por dos los valores de Gfaux en un bucle extra, ya que para hacer la multiplicación siguiente se debe llamar a una función externa. Los valores que van a acompañar a los términos lineales se guardan en un nuevo vector, "*linval*" (Figura *2-65*), que será el resultado de la multiplicación entre el vector 2Gfaux y la matriz G.

4.4.3.3 Cálculo del término cuadrático:

El cálculo del término cuadrático será el más complejo de realizar, al tener que ordenar los valores correspondientes a las variables cuadráticas a través de los índices de una matriz. Para ello en primer lugar se va a calcular la matriz H, que corresponde a realizar la multiplicación entre GT (Matriz G traspuesta) y G, y posteriormente sumar, a esta matriz resultado, una matriz identidad multiplicada por el parámetro λ.

```
/*
    CALCULO DE LOS TÉRMINOS CUADRÁTICOS
*/

//En primer lugar se realizará el cálculo matricial de H

//Se crea y guarda G traspuesta
double **GT;
GT = reservarMAT(Nu, N2);

traspMAT(G, N2, Nu, GT);


//Creamos la matriz H y en ella guardamos el resultado de la multiplicación de G trapuesta por G
double **H;
H = reservarMAT(Nu, Nu);

multiMAT(GT, G, Nu, N2, Nu, H);
```

Figura 2-66. Traspuesta de G y cálculo de la matriz H.

Para la suma de esta matriz identidad, se realizará a través de un bucle que únicamente recorra los valores de la diagonal, y a estos se les sumará el valor de λ (Figura *2-66*) directamente dando el resultado final de la matriz H. A continuación se hará la recolocación de los valores de esta matriz, ya que contiene los valores que acompañarán a todas las variables cuadráticas (las cuales son indicadas a través de los índices de las posiciones de la matriz). Como para denominar las diferentes variables no importa el orden de la multiplicación (por ejemplo, $U12 = U21$, cumplen el papel de la misma variable), se debe hacer la suma de los términos que corresponden a las mismas variables cuadráticas, éstos siendo los términos homólogos colocados a ambos lados de la diagonal principal de la matriz, sumados en un simple bucle y acumulados en las posiciones superiores a la diagonal principal de la matriz H (Figura *2-67*)

```
//Como la matriz identidad multiplicada por lambda va a ser una matriz con ÚNICAMENTE LAMBDAS EN LA DIAGONAL
//Se suman directamente el valor de lambda a la diagonal de la matriz H.

for (i = 0, j = 0;i < Nu, j < Nu;i++, j++) {
    H[i][j] += lambda;
}

//Se suman los términos que ocupan el lugar por encima de la diagonal
//en los términos correspondientes de la matriz triangular inferior
//Al representar a las mismas variables cuadráticas


for (i = 0;i < Nu;i++) {
    for (j = 0;j < Nu;j++) {
        if (i < j) {
            H[i][j] += H[j][i];
        }
    }
}
```

Figura 2-67. Obtención de la matriz H con los términos cuadráticos definitivos.

Por último, y ya no como parte del cálculo de los términos cuadráticos de la función objetivo, sino como preparación para el uso de estos términos dentro de las funciones de optimización de Gurobi, es necesario colocar los términos en orden de asignación, para ello únicamente se necesitará inicializar una variable entera que guarde el número de variables cuadráticas máximas (Figura *2-68*, líneas superiores). Y la colocación se hará realizando un bucle que recorra toda la matriz H, pero únicamente guarde los valores de la matriz que se sitúen en la triangular superior (Figura *2-68*, bucle inferior).

```
/*
    CÁLCULO PREVIO DE VARIABLES CUADRÁTICAS NECESARIO PARA LA OPTIMIZACIÓN EN GUROBI
*/

//Calculo del número de variables cuadráticas
int Nqvar;
Nqvar = ((Nu*Nu - Nu) / 2) + Nu;

//Bucles de asignación de los índices para las variables cuadráticas en dos vectores
k = 0;
for (i = 0;i < Nu;i++) {
    for (j = i;j < Nu;j++) {
        qfil[k] = i;
        qcol[k] = j;
        k++;
    }
}

//Se inicializa el vector que contendrá los valores de todas las variables cuadráticas en orden
double *qval;
qval = reservarVEC(Nqvar);

k = 0;
for (i = 0;i < Nu;i++) {
    for (j = 0;j < Nu;j++) {
        if (i <= j) {
            qval[k] = H[i][j];
            k++;
        }
    }
}
```

Figura 2-68. Cálculo de los vectores de las variables cuadráticas.

Este último cálculo culmina con la preparación del desarrollo de los términos de la función objetivo, teniendo el término independiente (f0), termino lineal (linval) y término cuadrático (qval además de los índices de las variables cuadráticas). El proceso de optimización de Gurobi será el apartado que sigue a estos cálculos, utilizando las funciones analizadas en el ejemplo sobre la resolución de un QP utilizando Gurobi (Apartado *3.5*). Pero todo este proceso de cálculo no puede funcionar por sí mismo, ya que del resultado de la optimización se deben actualizar los valores de las variables de la señal de control, volviendo a ejecutar el algoritmo del GPC de forma iterativa. Para ello se necesita planificar un bucle de control que englobe todo el proceso y lo realice de forma cíclica actualizando las variables que son relevantes, lo que se realizará en el siguiente apartado.

4.4.4 Programación del bucle de control de un GPC:

Para la programación general de bucle de control de un GPC, teniendo calculada la resolución completa de la ecuación diofántica, desarrollo de todos los términos de la función objetivo y proceso de optimización (conocimiento de las funciones de Gurobi relativas a la resolución de un QP (Apartado *3.4*)), se debe establecer la forma que tendrá el bucle a la hora de realizar una resolución iterativa del problema de optimización, tomando resultados de las señales de control, actualizando los mismos y volviendo a recalcular.

- Declaración de variables estáticas:

Como paso totalmente imprescindible en todos los comienzos de una programación, se deben inicializar las variables correspondientes al proceso en cuestión. De forma excepcional, las variables dentro del bucle de control será necesario que se mantengan entre ejecuciones del propio bucle, por ello serán variables estáticas (utilizando el comando "*static*" delante del tipo de variable a inicializar) las que almacenarán los valores pasados de las salidas pasadas del sistema, incrementos de las señales de control pasadas y señales de control pasadas (Figura *2-69*). Además, para evitar problemas con la reserva de memoria de las variables en el primer bucle, se inicializará una variable estática adicional, "*startup*", para que la reserva de espacio de los tres vectores anteriores solo se realice en el primera ejecución del algoritmo.

```
/*
    INIALIZACIÓN DE VARIABLES DE CONTROL NECESARIAS
*/

static int startup = 0;//Variable para la iniciación de los valores para el primer bucle

static double *deltaUpast;//Incremento de la señal de control pasadas
static double *ukpast;//Vector de las salidas pasadas necesarios (se inicializa en el primer bucle en cero)
static double *ypast;//Vector con las entradas pasadas necesarias

//Reserva de memoria de las variables estáticas en la primera ejecución del bucle
if (startup == 0) {

    deltaUpast = reservarVEC(nB);
    ukpast = reservarVEC(nB + 1);//Nesario para calcular la actualizacion del vector anterior
    ypast = reservarVEC(nA);//si estamos en el primer bucle, el segundo valor será cero,hay que inicializarlo igualandolo al primero

    startup = 1;
}
```

Figura 2-69. Inicialización de variables del bucle de control

- Actualización de los valores de entrada de la función:

La función en la que se realice el bucle tendrá que recibir de manera constante los valores actuales para, la salida del sistema real "Y" y el valor de referencia "W". Estas variables deberán ser actualizadas cada vez que se inicie un nuevo bucle de control (Figura *2-70*), en primer lugar, desplazando los valores de las salidas pasadas en su vector una posición (descartando así el último de los valores) y asignando el nuevo valor de la salida a la primera posición del vector "*ykpast*". Y a continuación se inicializará el vector de referencias futuras "w", al cual se le aplicará la convención de tomar el mismo valor en cada uno de los bucles para todo el horizonte de predicción.

```
//Actualización de las salidas pasadas del sistema
for (i = nA - 1;i > 0;i--) {

    ypast[i] = ypast[i - 1];

}ypast[0] = Y;//Actualización del vector salida


//Inicialización del vector de referencias futuras
double *w;
w = reservarVEC(N2);
for (i = 0;i < N2;i++) {
    w[i] = W;//Se aplica la convención de referencias constantes en el horizonte
}
```

Figura 2-70. Inicialización de la salida y del vector de referencias futuras.

- Actualización de la señal de control:

Con la declaración e inicialización de las variables del bucle de control, se procederá con toda la formulación y resolución de la función objetivo con la posterior optimización y obtención del vector de variaciones de control, en el que cada una de las filas representa la variación de control que el sistema necesitará en los sucesivos tiempos de muestreo hasta el horizonte de control, por ello se tomará únicamente el primer valor de este vector, actualizando, de la misma forma que se hizo con las salidas pasadas del sistema (desplazando el vector una posición y sustituyendo el primer valor), el vector de variaciones de control pasadas *"deltaUpast"* y el vector de la salida de control pasada "*ukpast*" (Figura *2-71*).

```
/*
    ACTUALIZACION DE VALORES DEL BUCLE DE CONTROL
*/

//Se deben actualizar los dos vectores referentes a las señales de control pasadas


//En primer lugar se actualiza el vector variaciones de control pasadas
for (i = nB-1; i > 0; i--) {
    deltaUpast[i] = deltaUpast[i-1] ;
}
deltaUpast[0] = deltaU[0];

//A continuación se actualiza el vector de salidas de control pasadas
for (i = nB; i > 0;i--) {
    ukpast[i] = ukpast[i-1];
}
ukpast[0] = *Usal;
```

Figura 2-71. Actualización de las variables del bucle de control.

5 CREACIÓN E IMPLEMENTACIÓN DE DLLS PARA SU USO EN EL ENTORNO LABVIEW

5.1 Puesta a punto del entorno Visual Studio para la creación de una DLL con la inclusión de las librerías Gurobi:

Del mismo modo que se hizo anteriormente a la hora de preparar el entorno de Visual Studio para la inclusión de las librerías de Gurobi en él (Apartado *3*), para un tipo de proyecto genérico, se realizarán los mismos pasos ahora para un proyecto de una librería de vínculos dinámicos (DLL, Dynamic Link Library), aunque con alguna salvedad en el proceso en comparación con el previamente realizado, se recorrerán los pasos a realizar de forma detallada.

5.1.1 Ejecutar Visual Studio y creación un nuevo proyecto:

Posteriormente a la ejecución del programa se optará por una de las dos opciones que se explicaron antes para crear un nuevo proyecto:

- A través de la barra de herramientas entrar en "Archivo > Nuevo > Proyecto".
- O dentro de la página de inicio bajo la sección "Nuevo Proyecto" al final del todo estará la opción "Crear nuevo proyecto...".

Ahora con el menú de "Nuevo proyecto" ya abierto, esta vez se recomienda introducir el nombre del proyecto que se quiera, el directorio y elegir la opción "Asistente para escritorio de Windows", la cual por defecto debería ser la última que aparece en la lista y estar seleccionada de antemano (Figura *3-1*,).

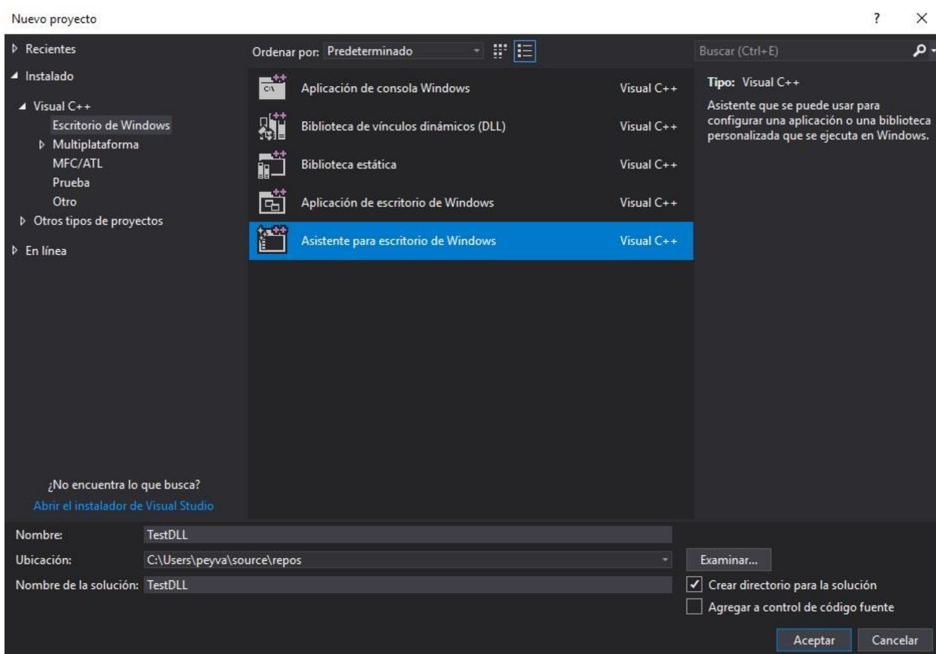


Figura 3-1. Creación de un archivo DLL en Visual Studio.

Seleccionando el nombre para el proyecto y aceptando en el cuadro de diálogo anterior, saldrá la siguiente ventana:

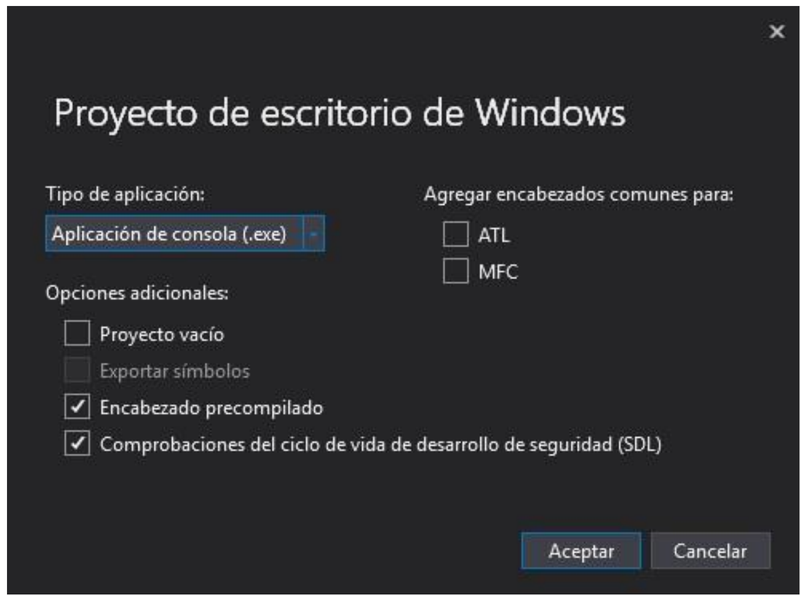


Figura 3-2. Opciones de creación de un proyecto en Visual Studio.

En esta ventana titulada "Proyecto de escritorio de Windows" se dejará todas las opciones que aparecen con los valores por defecto excepto el valor que haya escrito en el desplegable referente a "Tipo de aplicación" donde se seleccionará la opción "Biblioteca de vínculos dinámicos (.dll)" (Figura *3-3*). Después de realizar esta selección ya estará preparado el proyecto para que pueda crearse.

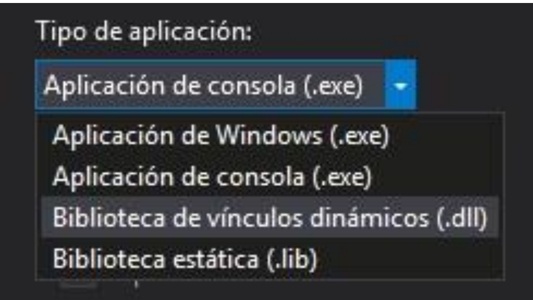


Figura 3-3. Elección del tipo de aplicación en Visual Studio.

A continuación se procederá a realizar los mismos cambios que se realizaron en el apartado de puesta a punto de las librerías de Gurobi para un código genérico en C.

5.1.2 Modificación de la plataforma en opciones del proyecto y del compilador a "x64":

Para la compilación y ejecución de todos los proyectos que se hagan con las librerías Gurobi es recomendable que se cambie esta opción para evitar que haya problemas de compatibilidades.

- Primero, como todos los cambios que se han realizado en el apartado anterior, dentro de la configuración del propio proyecto, en el apartado "General", en la pestaña superior se cambiará la "Plataforma" a x64.

- En segundo lugar, en la pantalla principal del proyecto, se modificará la opción del compilador, que viene por defecto "x86", a "x64".

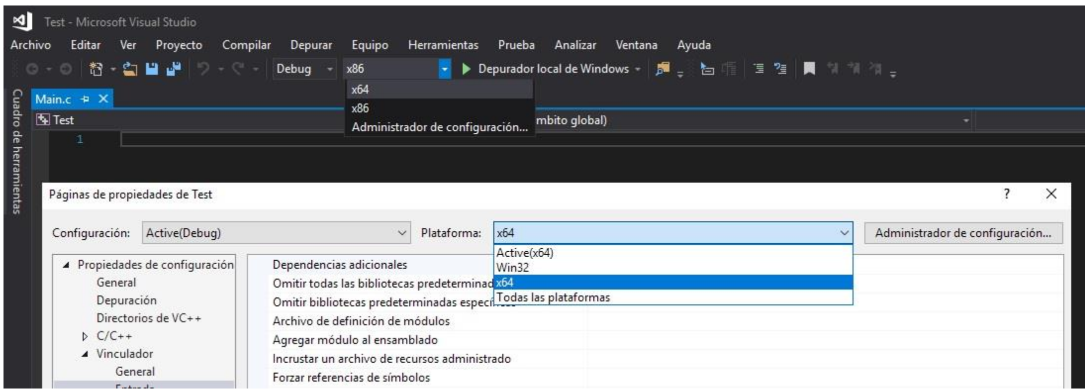


Figura 3-4. Modificación opciónes plataforma en Visual Studio.

5.1.3 Inclusión de las librerías Gurobi:

En este apartado se realizará la tarea principal para la inclusión de las librerías, ésta tendrá varias partes, todas ellas realizadas dentro del menú de "Propiedades" del proyecto, desplegándose haciendo clic derecho en el nombre del proyecto y yendo a la última opción.

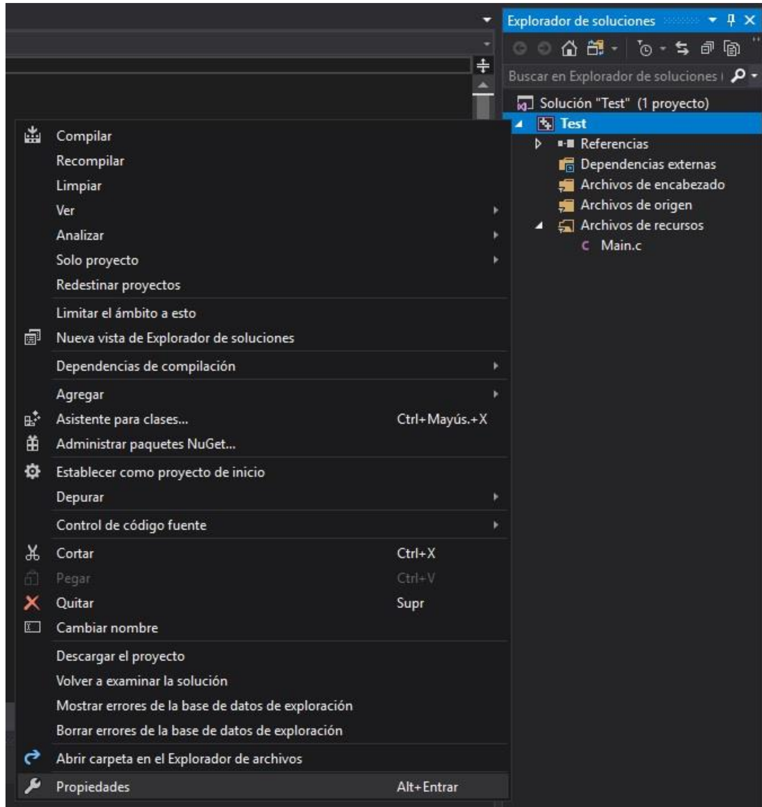


Figura 3-5. Desplegable propiedades de un proyecto DLL en Visual Studio.

A partir de la ventana que se abrirá a continuación, se abordarán en orden descendente por las opciones que aparecen en el desplegable que se muestra a la izquierda, las diferentes opciones del proyecto que se deben modificar para incluir las librerías necesarias de Gurobi, con el objetivo de crear archivos "*.dll*".

5.1.3.1 Modificación de la versión de Windows:

En el apartado que se abre por defecto al abrir las propiedades del proyecto, que deberá ser "General", se modificará la versión de "Windows" por la opción "8.1", ésta puede no estar disponible como opción en el desplegable si se ha realizado una instalación por defecto de Visual Studio, pero si se vuelve al instalador de del programa esta opción se puede agregar muy sencillamente. Como se puede ver, además del cambio realizado en estas opciones, los apartados sobre "Extensión de destino", tiene por defecto seleccionada la opción de "*.dll*" y en el apartado "Tipo de configuración", además remarcado en negrita, la opción de "Biblioteca dinámica (.dll)".

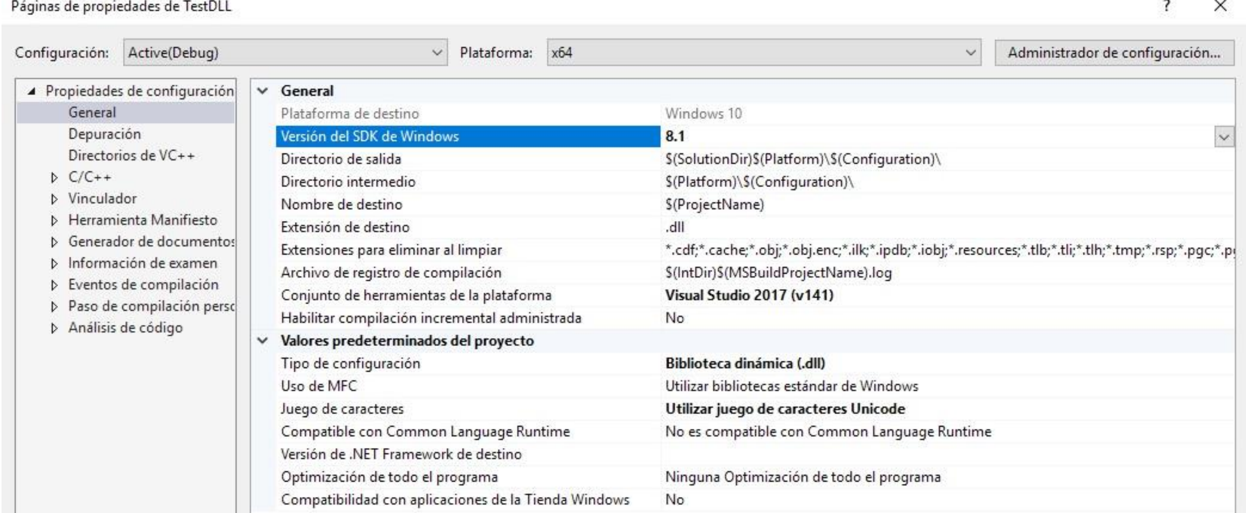


Figura 3-6. Modificación versión de Windows de una DLL en Visual Studio.

5.1.3.2 Modificación de los directorios de VC++:

Se seleccionará en el desplegable de la izquierda el menú "Directorios de VC++", y aquí se realizarán dos modificaciones:

- Primera en la opción "Directorios de archivos de inclusión", se clicará en la opción de "Editar", dentro de la ventana que aparecerá se copiará la ruta de la carpeta "include" de los archivos de la instalación de Gurobi.

- En segundo lugar debe hacer un cambio similar, pero copiando la ruta de la carpeta "lib", de los archivos de instalación de Gurobi, dentro de la opción "Directorios de archivos de bibliotecas", de la misma forma que antes seleccionando la opción "Editar".

5.1.3.3 Adición de directorios de inclusión:

Seleccionando el siguiente menú desplegable, "C/C++", y a continuación el primer apartado, "General". Se modificará la primera opción "Directorios de inclusión adicionales", y de la misma forma que las anteriores modificaciones, a través de la opción "Editar" se añadirá la ruta de la carpeta "include" de los archivos donde se haya realizado la instalación de Gurobi.

Estos dos pasos anteriormente definidos serían exactamente los mismos dos pasos que se realizaron en el apartado de preparación de Visual Studio anterior (Apartado *3.3.3.2*), ya que únicamente se están definiendo los directorios en los que se deben buscar los archivos de cabecera y las librerías que serán incluidas en un futuro en el proyecto, y estos directorios no deberían cambiar en ningún caso.

5.1.3.4 Modificación de la entrada del vinculador:

Por último en cuanto a las modificaciones de las propiedades del proyecto, seleccionando la pestaña "Vinculador" y dentro de ésta la opción "Entrada", se tendrá que añadir en este caso dos documentos, como se mencionó previamente, es necesario también añadir el archivo de librerías de Gurobi en lenguaje C++, por el hecho de que el archivo en el que se va a compilar la librería es un archivo "*.cpp*", lo que significará que se deberá de realizar una modificación a cierto archivo de la librería para que se pueda ejecutar código en C, pero esto será el tema principal del siguiente apartado.

Y por último, dentro del apartado de "Dependencias adicionales" se deberán añadir los archivos de las librerías, esto se realizará a través del nombre de los archivos correspondientes a cada una de las librerías estáticas que se quieran vincular, las librerías en C, "*gurobi75.lib*" y las librerías correspondientes en C++, "*gurobi_c++mdd2017.lib*".

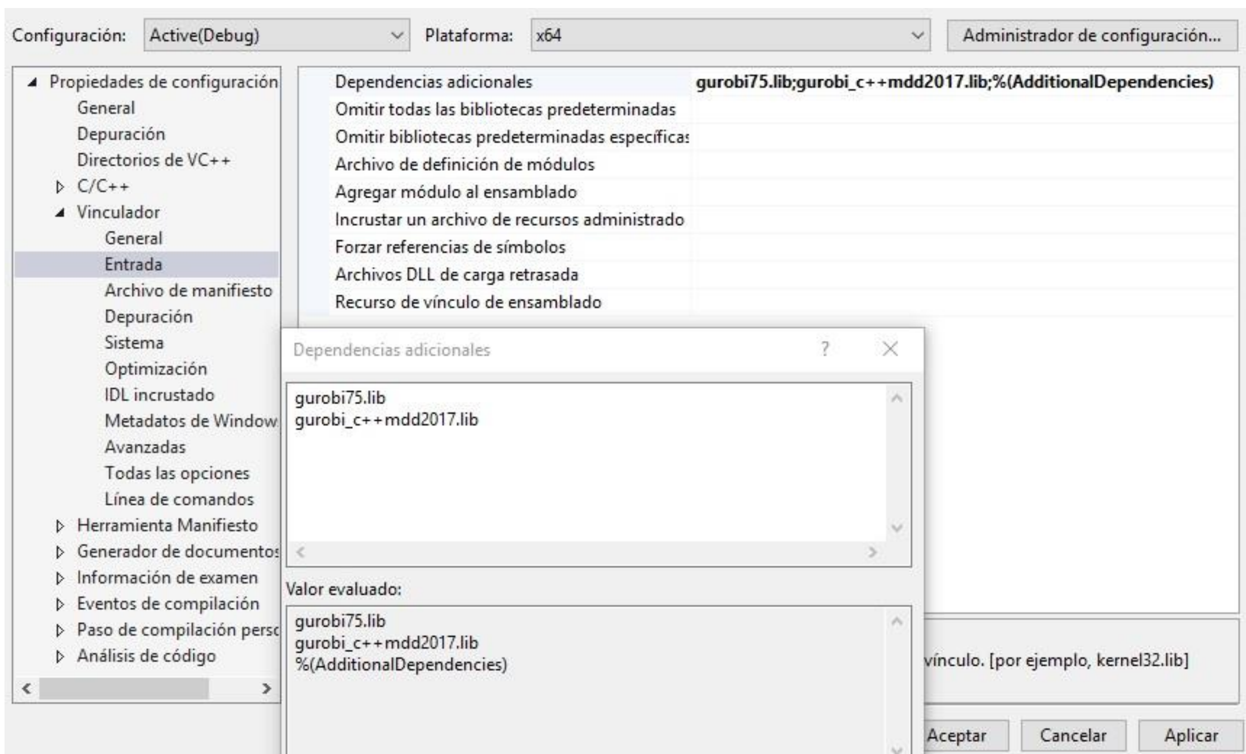


Figura 3-7. Adición de librerías de entrada en el Vinculador en Visual Studio.

5.1.4 Añadir código de encabezado:

En este caso por defecto se crea un archivo (de extensión "*.cpp*") en el que contener el código principal, y casi la totalidad del código de la DLL, no es necesario añadir ninguno más, a menos que se quiera dividir partes del proyecto en otros archivos para incluir de forma aislada éstos dentro de otros códigos de manera específica. Pero este tipo de modificaciones estarán únicamente en mano del programador, ahora se abordará la puesta a punto de un archivo DLL completo de la forma más básica posible, posteriormente todos los añadidos en complejidad que se quieran realizar serán posibles.

El archivo que se acaba de crear para la DLL deberá tener los siguientes documentos inicializados dentro del desplegable de la solución (Dentro del desplegable de "Dependencias externas" estarán todas las librerías agregadas por defecto por Visual Studio al crear un archivo DLL, si se abre se verá la cantidad de estos archivos que se incluyen por defecto, por esto mismo se comentó en un principio que los proyectos de Visual Studio no están hechos realmente para hacer cosas extremadamente simples ya que éstos ocuparán mucho espacio en el disco):

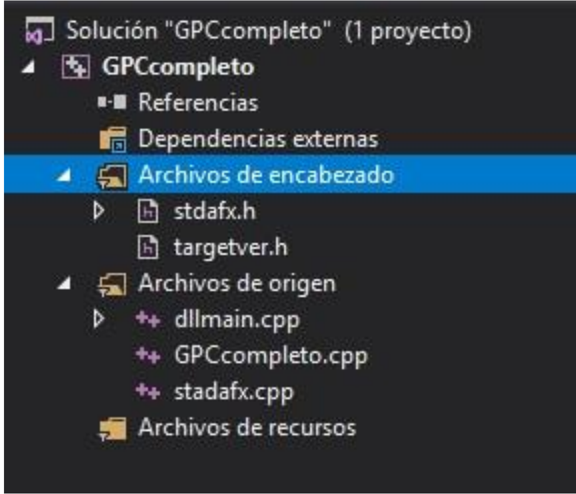


Figura 3-8. Desplegable de solución de una DLL iniciado por defecto.

Ahora se debe inicializar un documento nuevo que será la cabecera correspondiente al archivo "*.cpp*", por ello se le pondrá el mismo nombre que el archivo anterior pero con la extensión "*.h*", para crear un nuevo archivo de este formato, haciendo clic derecho en la carpeta del menú desplegable de la solución del proyecto llamada "Archivos de encabezado" y posteriormente entrando en la primera opción "Agregar" y a continuación en "Nuevo elemento", se podrá agregar el tipo de archivo requerido (Figura *3-9*).

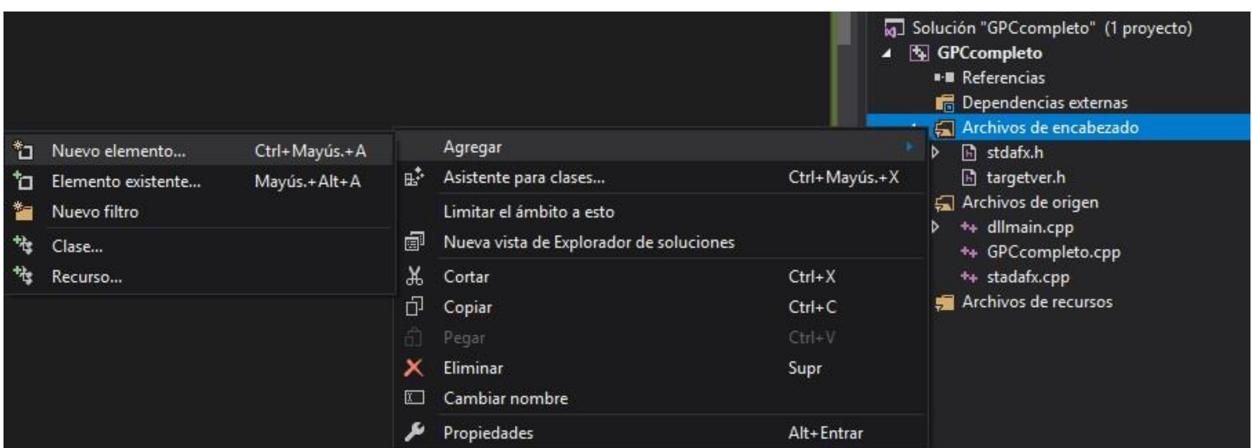


Figura 3-9. Desplegable agregar nuevo elemento de encabezado en Visual Studio.

Entrando ya en el menú de creación se tendrá la opción de elegir "Archivo de encabezado (.h)", se le asignará el mismo nombre que el archivo "*.cpp*" creado por defecto al crear el proyecto (nombre que coincidirá con el que se le haya puesto al proyecto). Y ya estarán todos los archivos necesarios creados para el uso de la librería, aunque todavía no estará inicializada para su compilación y uso, en el apartado siguiente se ahondará en la forma que deben tener los archivos para que el archivo DLL sea compilable y a la vez exportable al entorno LabVIEW.

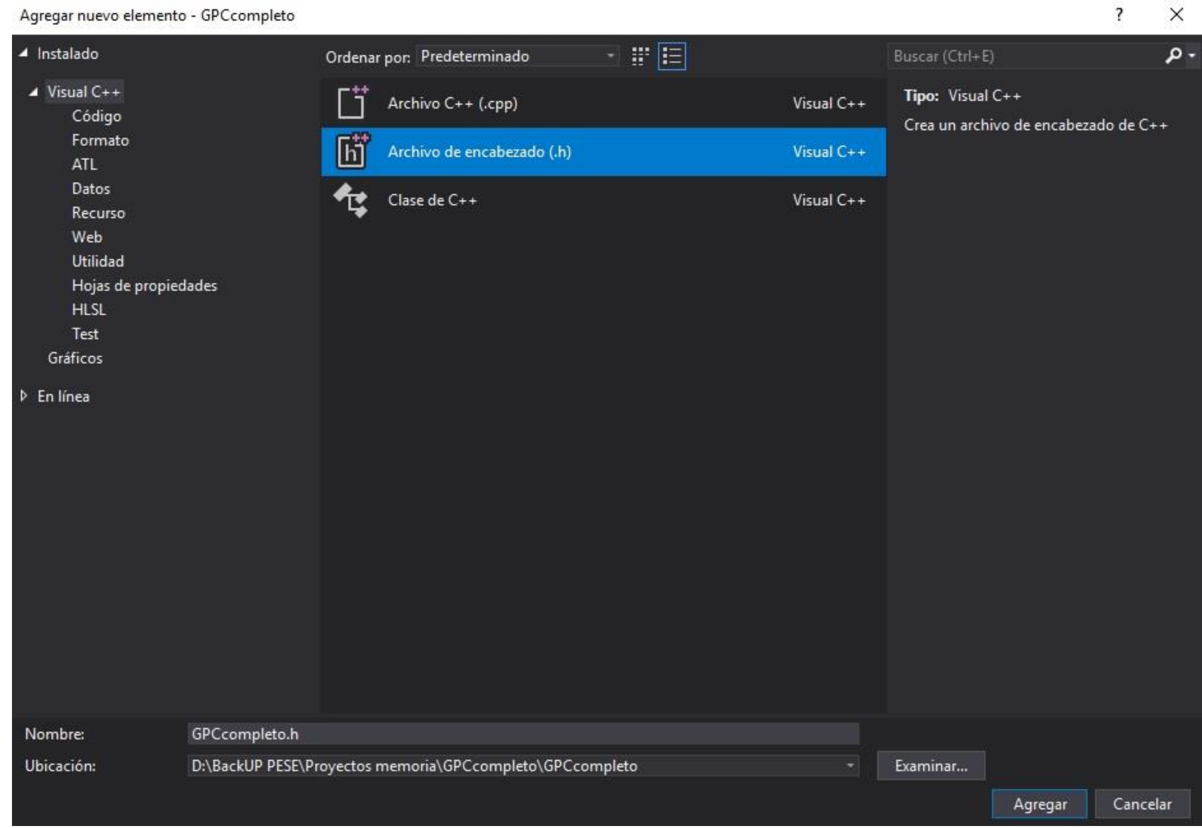


Figura 3-10. Creación de un archivo de encabezado en una DLL en Visual Studio.

Por último, de forma homóloga al apartado de puesta a punto de Visual Studio anterior, se proporcionarán en la documentación de este proyecto, un archivo comprimido con nombre "PlantillaGurobiDLL", la cual como se comentó anteriormente es una plantilla preparada con el archivo de encabezado inicializado de manera correcta para el uso posterior de LabVIEW, las librerías de Gurobi incluidas en él, las correspondientes para lenguaje C y C++. Pero sólo es ejecutable si la instalación de las librerías se ha hecho en el disco duro principal del ordenador, del mismo modo que harían las librerías Gurobi al completar una instalación por defecto, en el caso contrario en el que se haya hecho la instalación de las librerías Gurobi en otro directorio, será necesario seguir todos los pasos detallados anteriormente para incluirlas, ya sea con el objetivo de crear un simple script o una librería completa. Esta plantilla deberá copiarse en el directorio donde Visual Studio almacena sus propias plantillas, "Visual Studio 2017\My Exported Templates".

5.2 Formato de los archivos de una DLL para su uso en el entorno LabVIEW:

Dentro de un archivo estándar destinado para la compilación de una DLL en el entorno de LabVIEW se deberá tener en Visual Studio un desplegable de solución de proyecto con los siguientes archivos creados:

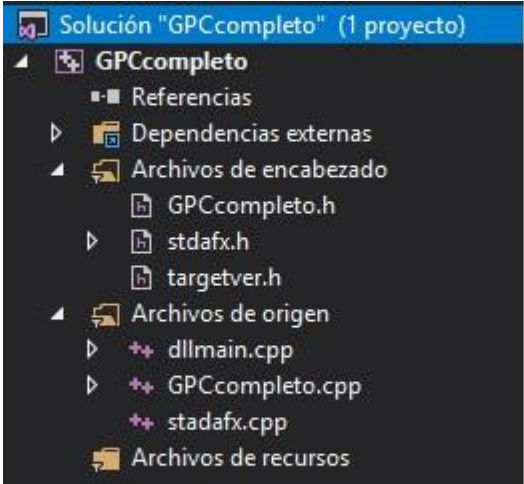


Figura 3-11. Desplegable solución de una DLL completo.

Dentro de los archivos que contiene la solución únicamente serán de interés, y por ello serán los únicos que se modificarán:

- Dentro de la carpeta "Archivos de encabezado": será imprescindible modificar el archivo de cabecera que se creó en el apartado anterior, en el cual se definirán las funciones que sean exportadas a LabVIEW.

- En la carpeta "Archivos de origen": se deben modificar los dos archivos principales que se crean de manera predeterminada, "*dllmain.cpp*" y el otro archivo principal de extensión "*.cpp*" que se crea con el nombre predefinido que se le haya puesto a la plantilla, en el caso que se use, o el nombre general del proyecto.

5.2.1 Formato del archivo "*dllmain.cpp*":

Teniendo presente que las librerías que se van a crear serán utilizadas dentro del entorno LabVIEW, desde National Instruments [19] se define una estructura específica para la definición de las funciones (en las que se deberá añadir en su prototipo el argumento "WINAPI", después del tipo de función, por ejemplo, "double WINAPI funejemplo()"). Dentro del apartado "2: Editing the source file" del documento mentado se comenta el formato recomendado para el archivo "*dllmain.cpp*" y además, se pone un ejemplo de la utilización de una librería simple en LabVIEW a través de este formato:

```
// dllmain.cpp : Define el punto de entrada de la aplicación DLL.
#include "stdafx.h"

BOOL WINAPI dllmain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpReserved)
{
    // Perform actions based on the reason for calling.
    switch (fdwReason)
    {
    case DLL_PROCESS_ATTACH:
        // Initialize once for each new process.
        // Return FALSE to fail DLL load.
        break;

    case DLL_THREAD_ATTACH:
        // Do thread-specific initialization.
        break;

    case DLL_THREAD_DETACH:
        // Do thread-specific cleanup.
        break;

    case DLL_PROCESS_DETACH:
        // Perform any necessary cleanup.
        break;
    }
    return TRUE;
}
```

Figura 3-12. Formato archivo *dllmain.cpp*.

5.2.2 Formato del archivo de encabezado:

El objetivo del archivo de encabezado de la función principal, será única y exclusivamente la exportación de las funciones que se necesiten a través del archivo "*.dll*" resultante, si no se declaran dentro de este archivo las funciones que se quieran utilizar a posteriori en otro entorno, no podrán ser utilizadas, es por este archivo debe tener un formato y una forma de declaración específica. A través de la Figura *3-1*3, se puede observar la forma de declaración que permitirá la exportación de las funciones de la librería que se quieran dentro del entorno de LabVIEW, además de permitir que se pueda introducir código programado en C dentro de los archivos "*.cpp*", que estarían en un principio definidos para la programación en C++.

```
#pragma once

#ifdef __cplusplus

extern "C" {

#endif

    __declspec(dllexport) void WINAPI DevuelveG(dou

    __declspec(dllexport) void WINAPI DevuelveGf(do

    __declspec(dllexport) void WINAPI BucleGPC(doub

#ifdef __cplusplus

}

#endif
```

Figura 3-13. Formato del archivo de encabezado de una DLL.

Dentro de la definición condicional, que será lo que permite la compilación de código externo en lenguaje C dentro de archivos "*.cpp*", se definirán las funciones que se crean necesarias, éstas serán definidas con la declaración de los prototipos, delante del cual se añadirá a modo de encabezado, "__declspec(dllexport) ", lo que permitirá que la función al que se le añada sea exportada al entorno LabVIEW si se agrega el archivo "*.dll*" que será creado al compilar el proyecto.

5.2.3 Formato general del archivo "*.cpp*" principal:

En este apartado no se ahondará al completo dentro del código general de las librerías, ya que en un apartado posterior se verá un código completo en el que se implementa una librería en LabVIEW, pero sí se comentará el formato básico que deberá tener este archivo, además conteniendo el grueso del código funcional de la aplicación que se vaya a desarrollar. En orden de aparición dentro del código:

- Inclusión de las librerías necesarias dentro del código:

Todas las inclusiones de librerías dentro de códigos se realizan utilizando el encabezado "#include" seguido del nombre del archivo de cabecera que se quiera incluir entre comillas dobles o símbolos "<" y ">".

Por defecto se debe incluir el archivo que es creado en la inicialización del archivo DLL por Visual Studio, "*stdafx.h*", con el fin de que a la hora de la compilación y creación del archivo "*.dll*" resultante no haya problemas, seguido de las librerías que sean necesarias dentro del código de manera genérica, como puedan ser "*stdio.h*", "*stdlib.h*", etc.

Y por último, se colocarán las librerías propias que se necesiten, en un caso específico dentro de este proyecto estas librerías podrían ser, las librerías de Gurobi (específicamente las librerías de Gurobi para C++, como se comentó antes es el archivo necesario de inclusión, aunque no es necesario que se incluyan las librerías en C, dentro del propio código de la cabecera de las librerías en C++ se incluye el archivo de Gurobi para C) y además se incluirá el archivo de cabecera creado previamente donde se definieron las funciones que se exportarán en un futuro en el entorno LabVIEW. En la figura siguiente se puede ver una declaración de archivos de inclusión estándar para un archivo DLL:

```
//Se incluyen todas las librerías que necesitará el código
#include "stdafx.h"
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include "GPCcompleto.h"

#include <gurobi_c++.h>
```

Figura 3-14. Formato de las librerías de inclusión típicas en una DLL con Gurobi.

- Definición de prototipos de funciones que no serán exportadas:

Es necesaria la definición de todos los prototipos de funciones internos de este código (es decir, todos excepto los cuales que se vayan a exportar fuera a través de la DLL, ya que estos serán declarados en el archivo de cabeceras que se creó previamente), la definición por defecto se hará con el tipo de función seguido de "WINAPI" y por último el nombre de la función y los parámetros requeridos por la misma entre paréntesis, se puede ver un ejemplo de funciones realizadas para el uso de otras funciones de Gurobi dentro de la librería:

```
//Funciones para el cálculo preparatorio de las matrices del control predictivo
void WINAPI calcG(double *AA, double *BB, int N2, int Nu, int na, int nB, double** G);
void WINAPI calcGF(double *AA, double *BB, int N2, int Nu, int na, int nB, double** Gf);
void WINAPI CalculosGyGf(double *AA, double *BB, int N2, int Nu, int na, int nB, double** GG, double** Gf);

//Función de cálculo de convolución entre vectores
void WINAPI convolucion(int NA, int NB, double A[], double B[], double *Res);

//Funciones de reserva de espacio en memoria para matrices y vectores
double** WINAPI reservarMAT(int filas, int columnas);
double* WINAPI reservarVEC(int filas);

//Función de linealización por filas de una matriz:
void WINAPI LinMat(double **M, int N1, int N2, double *Res);
```

Figura 3-15. Ejemplo definición de prototipos en C.

- Código completo de las funciones definidas:

A continuación generalmente se tendrá el grueso del código, en el que se irán colocando los códigos de todas las funciones declaradas, las que serán exportadas y las que no, el orden de definición no es relevante, pero si existiese un número excesivamente grande de funciones lo más correcto sería ir definiendo las funciones tomando algún tipo de orden o quía, ya sea relevancia, aparición o exportadas y no exportadas, una opción, por ejemplo, podría ser correcto colocar las funciones exportables en primer lugar y las funciones auxiliares de cálculo posteriormente, agrupando las mismas por similitud de funcionalidad.

5.3 Proceso de implementación de una DLL en LabVIEW:

La implementación de archivos DLL dentro de LabVIEW se realiza de una manera ciertamente directa, ya que únicamente a través de la utilización de un bloque de las biblioteca de LabVIEW se puede vincular un archivo DLL, siguiendo los pasos de puesta a punto de proyectos de DLLs en Visual Studio será de hecho una implementación prácticamente directa. El bloque necesario para la inclusión de librerías en LabVIEW será, "Call Library Function" (Bloque para la llamada de una función de una librería) [20], el cual tiene multitud de variables que es posible modificar, ya que a la hora de incluir código externo se podrá hacer de multitud de formas, aunque con la utilización de la herramienta Visual Studio, el propio Nacional Instruments recomienda cierto tipo de programación (Apartado *5.2*).

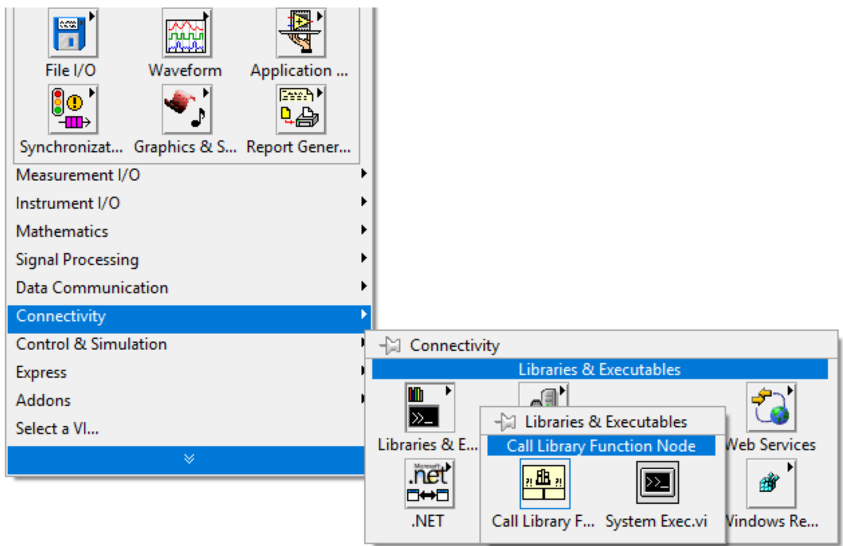


Figura 3-16. Desplegable paleta de funciones de LabVIEW.

A la hora de poder trabajar con el bloque en cuestión, será necesario acceder a través de la ventana "Block Diagram" (Diagrama de bloques) a la paleta de funciones, a la que por defecto se puede acceder haciendo clic derecho dentro de la ventana. Dentro del gran desplegable de opciones que ofrece LabVIEW dentro de su tableta se accederá a través de las pestañas "Connectivity" > "Libraries & Executables" > "Call Library Function Node" (Conectividad > Librerías y ejecutables > Nodo para llamada de librerías) (Figura *3-16*), y arrastrando desde este último desplegable ya se tendrá el bloque que se precisa. Si se hace clic derecho encima del mismo se pueden ver todas las funciones que ofrece el propio nodo (Figura *3-17*)

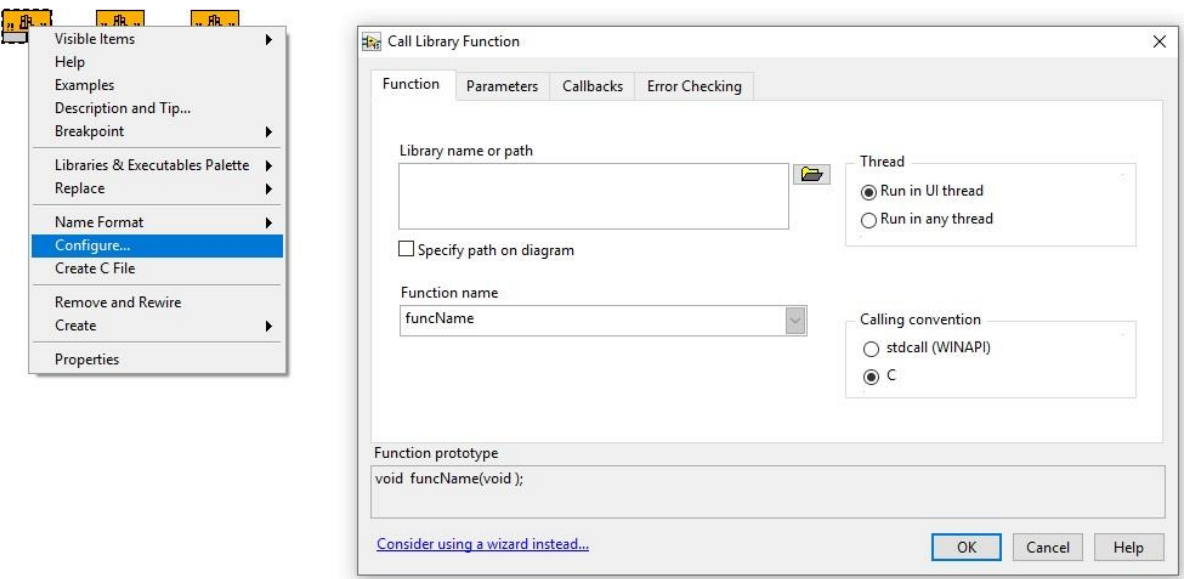


Figura 3-17. Opciones de configuración del bloque de llamada a una librería en LabVIEW.

A pesar de que la mayor parte de las opciones son irrelevantes en cuanto programación o modificación del comportamiento del propio bloque, se ofrecen opciones estéticas:

- "Visible Items".
- "Name format".

Opciones para la facilidad de la programación:

- "Breakpoint".
- "Libraries & Executables Palette".
- "Replace".
- "Create C File".
- "Remove and Rewire".
- "Create".

Opciones de ayuda:

- "Help".
- "Examples".
- "Description & Tip…".

Pero la realmente importante que sí ofrece la modificación y programación del bloque:

- “Configure”: Opción a la que se puede acceder directamente al hacer doble clic en el bloque, dando como resultado que se abra la ventana que se puede observar a la derecha de la Figura *3-17*.

A continuación dentro de las opciones configurables del bloque, se tienen cuatro pestañas diferentes a las que se puede acceder, “Function”, “Parameters”, “Callbacks” y “Error Cheking”. Las únicas que no se dejarán con valores por defecto serán las dos primeras, las cuales deberán ser modificadas cada vez que se quiera introducir un archivo de librería diferente dentro de LabVIEW.

- Pestaña “Function”.

Dentro de la primera pestaña se deberá hacer la identificación del archivo “*.dll*” que se quiera implementar y la función en cuestión, especificando la forma de llamada de la misma que se haya realizado.

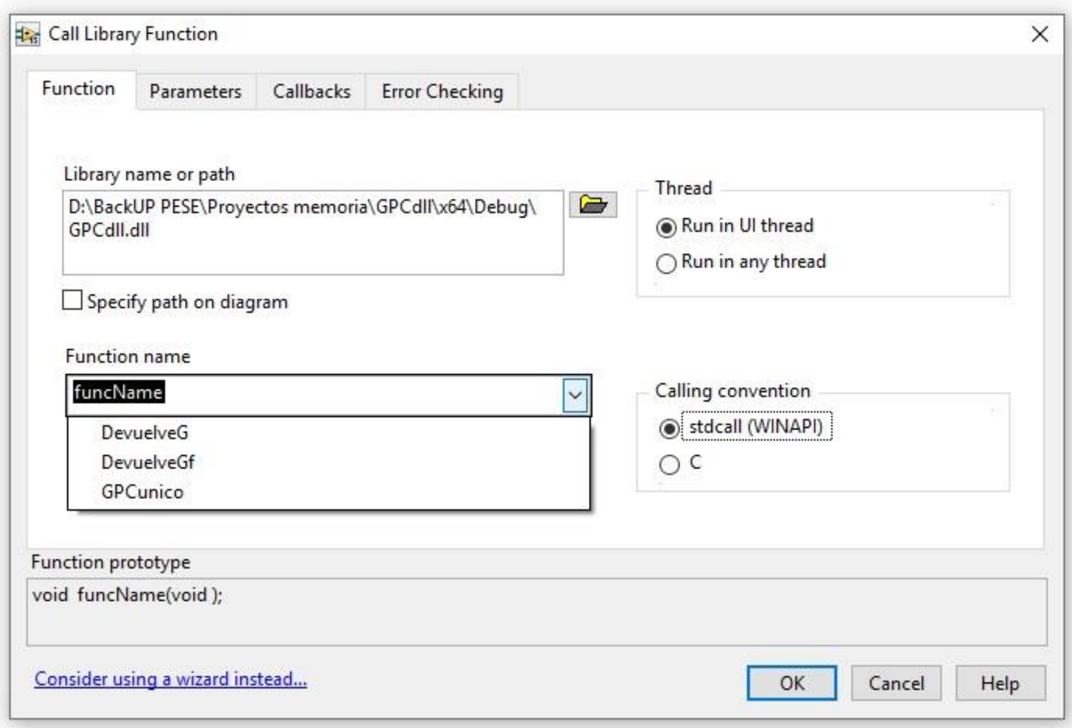


Figura 3-18. Selección de una librería externa en LabVIEW.

Dentro del primer cuadro “Library name or path”, que aparece vacío en un principio, se deberá especificar la ubicación del archivo “*.dll*” que haya salido como resultado después de la compilación del código dentro de Visual Studio, a través del símbolo de carpeta a la derecha del recuadro (Figura *3-18*). Con la compilación del código, Visual Studio creará varias carpetas donde irá almacenando los archivos resultado de las ejecuciones, en el caso de haber cambiado la plataforma como se ha explicado anteriormente (Apartado *3.3.4* y *5.1.2*), el archivo en cuestión estará guardado dentro de la carpeta “x64”, en caso contrario podrá estar dentro de la carpeta “x86” o “Debug” dependiendo de las opciones que se hayan especificado dentro de Visual Studio. Pero en general hay que tener en cuenta que para poder ejecutar una librería dentro de LabVIEW, ambos elementos tiene que utilizar la misma plataforma, si no el propio programa rechazará el archivo antes siquiera de la ejecución.

La segunda elección que debe hacerse será la función a exportar, ya que dentro de una librería pueden existir diferentes funciones exportables, sólo podrán ejecutarse de una en una en los nodos de librerías de LabVIEW. En caso de que se haya seleccionado correctamente la ruta del archivo "*.dll*", dentro del desplegable "Function name" deberán hallarse todas las funciones que pueden ser exportadas (Figura *3-18*). Y por último se debe elegir la forma de llamada que tenga la función elegida en el código, si se han seguido las recomendaciones de Nacional Instruments, se deberá seleccionar la opción "stdcall (WINAPI)" dentro del recuadro "Calling convention" situado en la esquina inferior derecha de la ventana.

- Pestaña "Parameters".

Dentro de esta segunda pestaña que será modificada, tendrá que realizarse todas las declaraciones que se hayan hecho dentro del prototipo de la función que se va a exportar, esto puede tener limitaciones, ya que el número y tipo de variables de entrada o de salida puede o ser limitado o no ser siquiera una posibilidad en algún caso, limitación que se deberá de tener en cuenta previamente a la hora de programar las funciones de la librería.

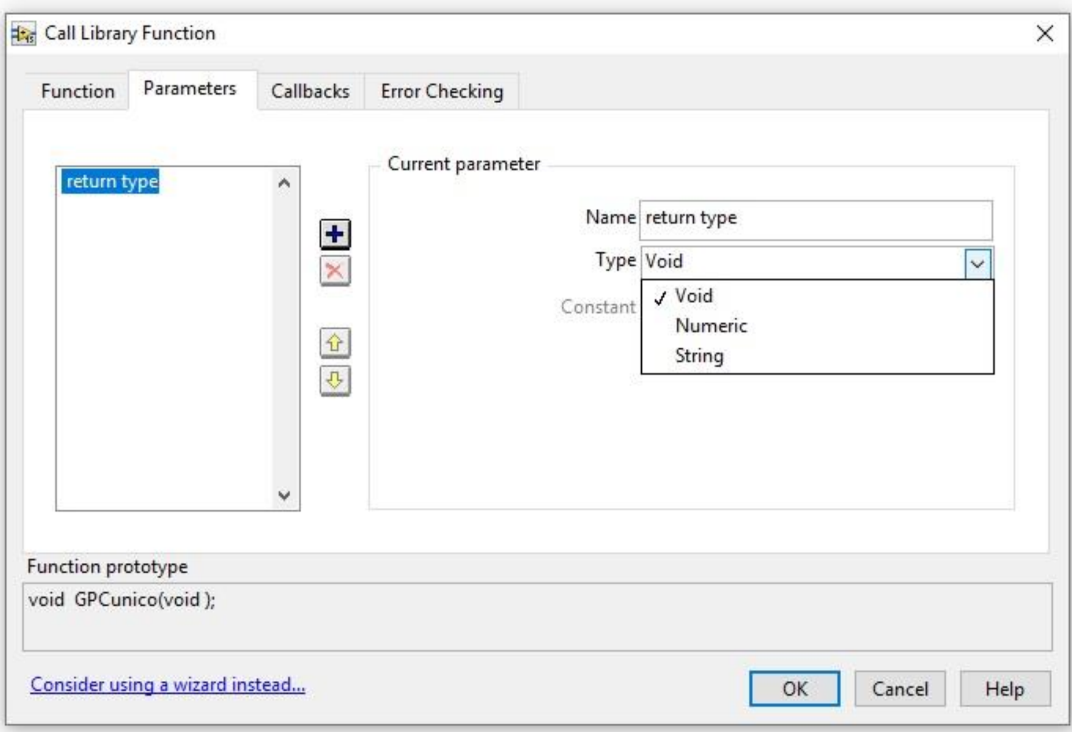


Figura 3-19. Variable de retorno de una librería externa en LabVIEW.

Por defecto el único parámetro que viene inicializado dentro de la función será el parámetro de retorno, dentro de sus opciones además de poder modificar el nombre se puede modificar el tipo entre:

- "Void": Sin parámetro de retorno para la función.
- "Numeric": Parámetro numérico de retorno de la longitud que se requiera.
- "String": Parámetro de tipo cadena de caracteres.

Dentro de los parámetros de retorno de manera general se tomará la opción "Void", ya que si se elige cualquier otro tipo de parámetro, se estará limitando a un único parámetro de salida que se pueda tomar de la función, y también se limita directamente los tipos de variables que un bloque de función puede devolver, ya que es imposible que pueda devolver variables tipo "array" de una o más dimensiones o punteros de cualquier tipo.

Seleccionado el valor de retorno, lo que se debe hacer a continuación será declarar todas las variables que contenga el prototipo de la función que se va a exportar, estos se deben tomar directamente del propio código, ya que LabVIEW no los detecta de ninguna forma. Y de manera homóloga a las anteriores opciones, como se puede observar en la imagen extendida a continuación (Figura *3-20*), se tiene la opción de elegir el tipo de variable aunque con más posibilidades de elección, el tipo de dato y la forma en la que se va a elegir realizar la comunicación a través del código, con paso por valor o por referencia ("Pointer to value").

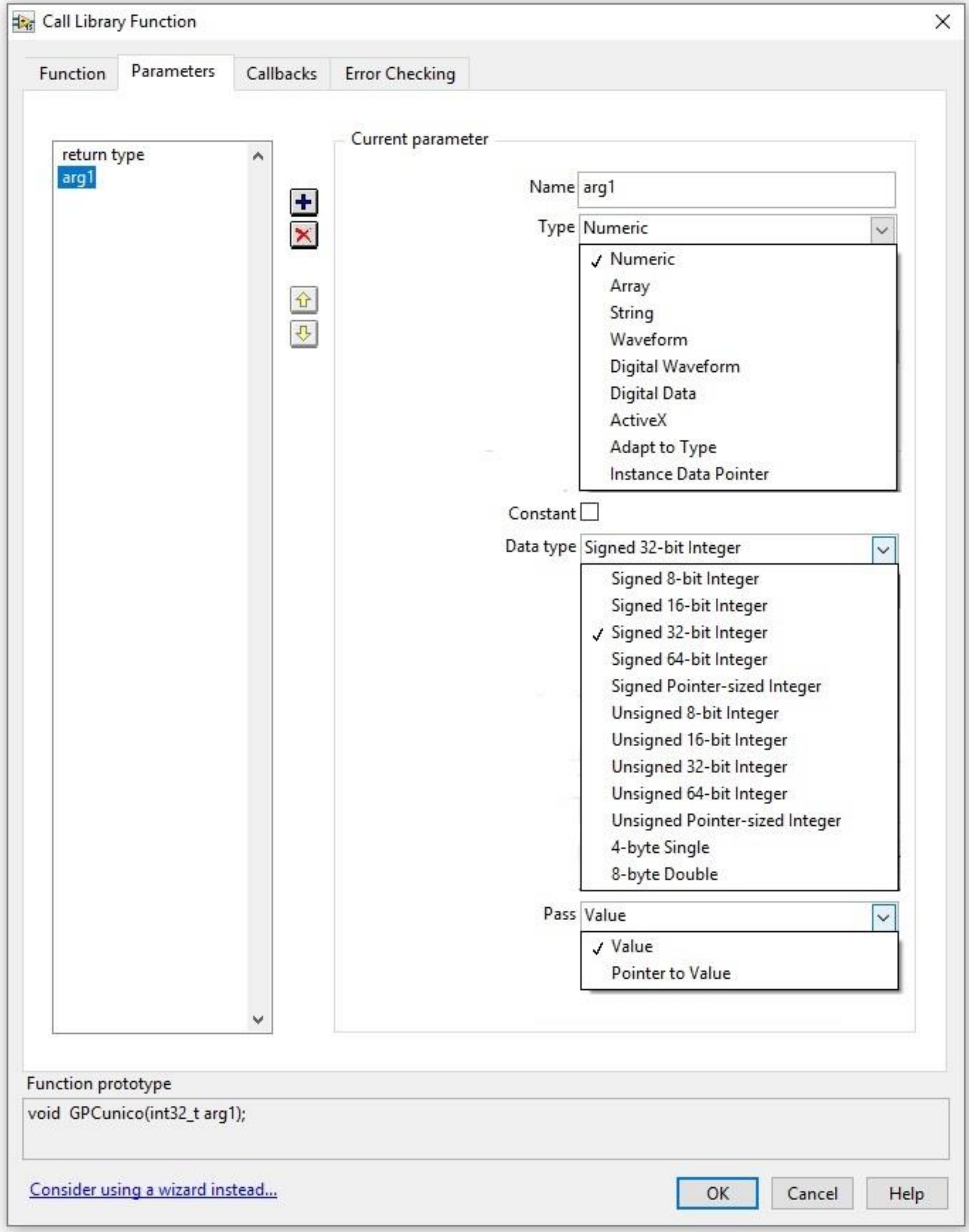


Figura 3-20. Definición de variables de una librería externa en LabVIEW.

Y a través de este último punto, es por donde se solucionará el problema de la imposibilidad de elegir parámetros de retorno para las funciones si son más complejos que los que ofrece LabVIEW. Ya que en caso que se quiera tomar uno o más valores de variables variable del código, lo que se hará es declarar dentro del prototipo de la función una variable que utilice este paso por referencia, y a la que a la entrada del bloque de llamada de función, no llegue ninguna variable dentro de LabVIEW. La variable que se quiera extraer debe ser declarada y asignada dentro del código. De esta forma se podrán obtener a través del bloque de función en LabVIEW los resultados que se quieran ver en línea.

Aunque a este respecto se le añade una excepción en el caso en que se necesite transmitir matrices de dos dimensiones, para ello, la matriz deberá ser linealizada por filas antes de ser enviada dentro del propio código (colocando todas las filas de la matriz en orden dentro de un nuevo vector) y además, a través de LabVIEW, se deberá asignar a la variable en cuestión, su longitud (Figura *3-21*), la cual, para aparecer en el desplegable disponible dentro de las opciones de la definida , tiene que ser una de las variables dentro del prototipo de la función, y por ello, ser un parámetro en el bloque de llamada de función de LabVIEW.

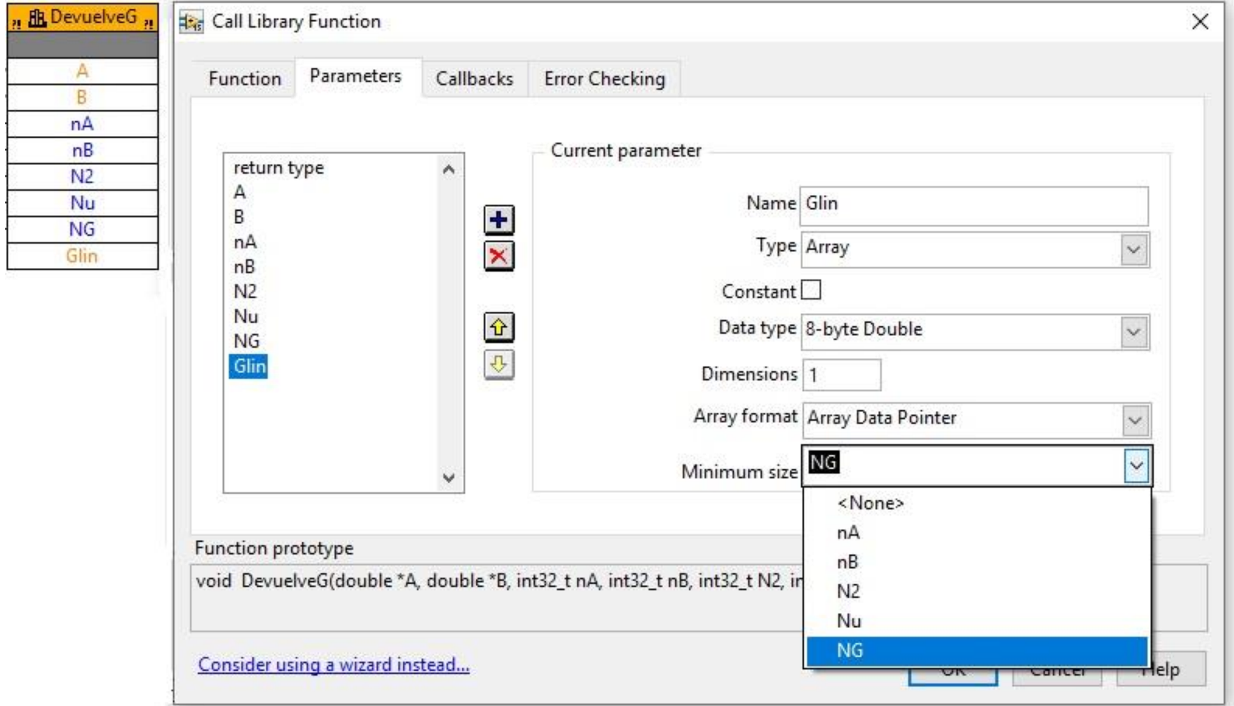


Figura 3-21. Adición de matrices de dos dimensiones en LabVIEW.

La linealización por filas de las matrices se realizará a través de una función (Figura *3-22*) que tome una matriz de entrada, sus dimensiones, y un vector resultado inicializado previamente, sabiendo que su dimensión será el producto del número de filas y columnas de la matriz de origen.

```
//FUNCION PARA LA LINEALIZACIÓN POR FILAS DE UNA MATRIZ

void WINAPI LinMat(double **M, int N1, int N2, double *Res) {

    int i, j;
    int k = 0;
    for (i = 0;i < N1;i++) {
        for (j = 0;j < N2;j++) {
            Res[k] = M[i][j];
            k += 1;
        }
    }

}
```

Figura 3-22. Función de linealización de una matriz por filas en C.

Si se debe utilizar este método de linealización para tener la posibilidad de transmitir matrices de dos dimensiones, si en algún caso esta matriz linealizada es la entrada dentro de otro bloque de llamada a una función de otra librería, esta matriz debe ser reconstruida si se quiere realizar cualquier tipo de cálculo con ella a posteriori, es por esto que se va a crear una función que realice el proceso inverso anterior, para la construcción de una matriz a través de un vector linealizado. De manera opuesta a la anterior función (Figura *3-23*), se tomará como parámetros de entrada el vector y las dimensiones de la matriz resultado, y la matriz construida será el resultado de esta función, debiendo haberse reservado el espacio para la misma previamente a la llamada de la función de reconstrucción.

```
void WINAPI ConstMat(double *V, int N1, int N2, double **Res) {

    int i, j;
    int k = 0;

    for (i = 0;i < N1;i++) {
        for (j = 0;j < N2;j++) {
            Res[i][j] = V[k];
            k += 1;
        }
    }

}
```

Figura 3-23. Función de construcción de una matriz linealizada por filas en C.

5.4 Ejemplo de implementación en LabVIEW de librerías para el cálculo de matrices de un GPC:

Expuestas todas las opciones relevantes que pone a disposición LabVIEW, a la hora de implementar códigos externos al entorno en forma de librerías, se va a plantear la realización de una aplicación para una librería en LabVIEW que, a través de los vectores que definen un modelo en función de transferencia y los horizontes de control y predicción, realice todos los cálculos preparatorios de un GPC de dos formas diferentes:

- A través de una única función que realice todos los cálculos.
- A través de dos funciones diferenciadas para cada matriz.

Para ambos casos se deberá mostrar por pantalla el resultado de las matrices linealizadas por filas, para que sean visualizables y exportables a través de un bloque de llamada de una librería.

5.4.1 Creación de las funciones de cálculo:

Partiendo del código utilizado en el ejemplo de resolución de la ecuación diofántica (Apartado *4.4*), en el que se realiza el cálculo preparatorio conjunto dentro de una única función en C (*Anexo D*), esta misma función, "*CalculosGyGf*", podrá ser exportada de forma directa al nuevo código DLL que se debe crear, del mismo modo que las funciones auxiliares necesarias para el cálculo:

- Función convolución.
- Función reserva de memoria dinámica para matrices y vectores (*"reservarMAT*" y "*reservarVEC"*).
- Función para linealización por filas de matrices ("*LinMAT*").

Pero al estar trabajando con un archivo DLL, se deberá añadir la convención de llamada de las funciones, con el fin de que no se encuentren incompatibilidades a la hora de compilar el código dentro del entorno LabVIEW, a todos los prototipos de las funciones el término "WINAPI" entre el tipo de función y el nombre de la misma (por ejemplo "*void WINAPI CalculosGyGf* ()").

Siguiendo este proceso en todas las funciones mencionadas, ya se deberá haber cumplido el primer objetivo del ejemplo. Para la segunda parte del mismo, se deberán crear dos funciones adicionales, una que realizará los cálculos necesarios para obtener cada una de las matrices por separado, tarea que podría ser resumida simplemente con la división de la función anterior, utilizando la parte del algoritmo que ambas funciones comparten (Cálculo de las matrices E y F a través de la resolución de la ecuación diofántica (Figura *3-24*)), seguido por las partes exclusivas del cálculo para las matrices correspondientes de cada función.

```
//CALCULO DE LA MATRIZ F

double **F;
F = reservarMAT(N2, nA - 1);

//Calculo del valor de la primera fila de F (obtenido de dividir AD entre 1)

for (i = 1;i < nA;i++) {
    F[0][i - 1] = -AD[i];
}


for (i = 0;i < N2 - 1;i++) {
    for (j = 0;j <= nA - 2;j++) {
        F[i + 1][j] = F[i][j + 1] - F[i][j] * AD[j + 1];

    }
    F[i + 1][1] = -F[i][0] * AD[nA - 1];
}


//CALCULO DE LA MATRIZ E

double **E;
E = reservarMAT(N2, Nu);

//Se inicializa el primer valor del vector que coincide con toda la primera columna
for (i = 0;i < N2;i++) {
    E[i][0] = 1;
}

k = 0;
for (i = 1;i < N2;i++) {
    for (j = 1 + k;j < Nu;j++) {
        E[j][i] = F[k][0];
    }
    k += 1;
}
```

Figura 3-24. Cálculo de las matrices de la ecuación diofáncita en C.

Completando el código de cada una de las nuevas funciones ("*CalcG*" y "*CalcGF*"), el cálculo para la resolución de la ecuación diofántica irá seguido por, el código para el cálculo de G (Figura *3-25*, izquierda) o el código para el cálculo de Gf (Figura *3-25*, derecha), representados a continuación.

```c
//CALCULO DE LA MATRIZ G
//Reserva espacio para la variable que contendrá la convolucion para calcular G

double *convG;
convG = reservarVEC(Nu + nB - 1);
//Reserva del vector completo con los valores de E
double *Eaux;
Eaux = reservarVEC(Nu);

//Extraccion de la componente con todos los valores que tendrá E en todo el horizonte
for (i = 0;i < Nu;i++) {
    Eaux[i] = EE[Nu - 1][i];
}


//Calculo de la convolucion de E y B
convolucion(Nu, nB, Eaux, BB, convG);

//Y por ultimo se colocan los términos de la convolución en las diagonales para obtener G

for (i = 0;i < N2;i++) {
    for (j = 0;j < Nu;j++) {
        if (i - j >= 0) {
            GG[i][j] = convG[i - j];
        }
    }
}
```

```c
//CALCULO COMPLETO DE LA MATRIZ Gf

double *Eaux;//En este vector se guardan las filas sucesivas de E
Eaux = reservarVEC(Nu);

double *aux;//En este vector se guarda el valor de la convolución sucesivamente
aux = reservarVEC(Nu + (nB - 1));

double **Gfaux;
Gfaux = reservarMAT(N2, (nB - 1));

for (i = 0;i < N2;i++) {
    for (j = 0;j < Nu;j++) {
        //Se extrae en cada bucle la componente de E correspondiente
        Eaux[j] = EE[i][j];
    }
    convolucion(i + nB, nB, Eaux, BB, aux);

    for (j = 0;j < (nB - 1);j++) {
        Gfaux[i][j] = aux[i + 1];
    }
}

//Se colocan en la matriz definitiva para Gf primero Gfaux y luego F

for (i = 0;i < N2;i++) {
    for (j = 0;j < (nB - 1);j++) {
        Gf[i][j] = Gfaux[i][j];
    }

}

for (i = 0;i < N2;i++) {
    for (j = nB - 1, k = 0;j < (nA + nB - 2), k < (nA - 1);j++, k++) {
        Gf[i][j] = FF[i][k];
    }
}
```

Figura 3-25. Comparación códigos para cálculo de matrices del GPC en C.

5.4.2 Creación de las funciones exportables de la DLL:

Con el fin de cumplir los dos requerimientos propuestos en el ejemplo, será necesaria la definición de tres funciones diferentes que se exporten a LabVIEW, siguiendo todas la misma estructura, definición y reserva de espacio de las matrices resultado del cálculo, llamada a la función de cálculo que sea necesaria, y por último, linealización de la matriz resultado (Figura *3-26*).

```c
void WINAPI GPCunico(double* A, double* B, int nA, int nB, int N2, int Nu, int NG, int NGf, double* Glin, double* Gflin) {

    double **G;
    G = reservarMAT(N2, Nu);

    double **Gf;
    Gf = reservarMAT(N2, (nA + nB - 1));

    CalculosGyGf(A, B, N2, Nu, nA, nB, G, Gf);

    //Linealización de G y Gf por filas para poder enviarse a traves de una función de labview
    LinMat(G, N2, Nu, Glin);
    LinMat(Gf, N2, (nA + nB - 1), Gflin);

}

void WINAPI DevuelveG(double* A, double *B, int nA, int nB, int N2, int Nu, int NG, double* Glin) {

    //Reserva de memoria y posterior calculo de la matriz G
    double **G;
    G = reservarMAT(N2, Nu);

    calcG(A, B, N2, Nu, nA, nB, G);

    //Linealización de G por filas para poder ser enviada a través de la dll en labview como Vector
    LinMat(G, N2, Nu, Glin);

}


void WINAPI DevuelveGf(double* A, double *B, int nA, int nB, int N2, int Nu, int NGf, double* Gflin) {

    double **Gf;
    Gf = reservarMAT(N2, (nA + nB - 1));

    calcGF(A, B, N2, Nu, nA, nB, Gf);

    //Linealización de Gf por filas
    LinMat(Gf, N2, (nA + nB - 1), Gflin);

}
```

Figura 3-26. Funciones exportables de una DLL a LabVIEW.

El último paso para que las funciones creadas sean exportables será su declaración (Figura *3-24*), pero, al contrario que las demás funciones, se declararán fuera del archivo principal, en el archivo de encabezados "*.h*" (Apartado *5.2.2*).

```
#pragma once

#ifdef __cplusplus

extern "C" {

#endif

    __declspec(dllexport) void WINAPI GPCunico(double* A, double* B, int nA, int nB, int N2, int Nu, int NG, int NGf, double* Glin, double* Gflin);

    __declspec(dllexport) void WINAPI DevuelveG(double* A, double *B, int nA, int nB, int N2, int Nu, int NG, double* Glin);

    __declspec(dllexport) void WINAPI DevuelveGf(double* A, double *B, int nA, int nB, int N2, int Nu, int NGf, double* Gflin);

#ifdef __cplusplus

}

#endif
```

Figura 3-27. Declaración de las funciones exportables en el archivo de encabezado.

El código de todos los archivos que comprenden la librería completa estarán disponibles, además de dentro de los archivos adjuntos de este proyecto (*GPCdll*), transcrito dentro del *Anexo D*.

5.4.3 Creación del instrumento virtual de LabVIEW:

Como último apartado dentro de la implementación y primera vez hasta este momento, se deberá crear un instrumento virtual nuevo en LabVIEW, al que en primer lugar se añadirán tres bloques de llamada a librerías externas con todos las variables que contenga cada una de las funciones que se quiera exportar (Figura *3-27*), donde de forma genérica para los parámetros vector se declararán como "array" (Figura *3-28*, izquierda) y para los parámetros que no sean vectores, se designarán como "numeric" (Figura *3-28*, derecha) acompañado del formato y tamaño que se necesite, por lo general para variables de tipo "int" se designará un "Signed 32-bit integer" y para las variables numéricas de tipo double, en cambio "8-bit double".

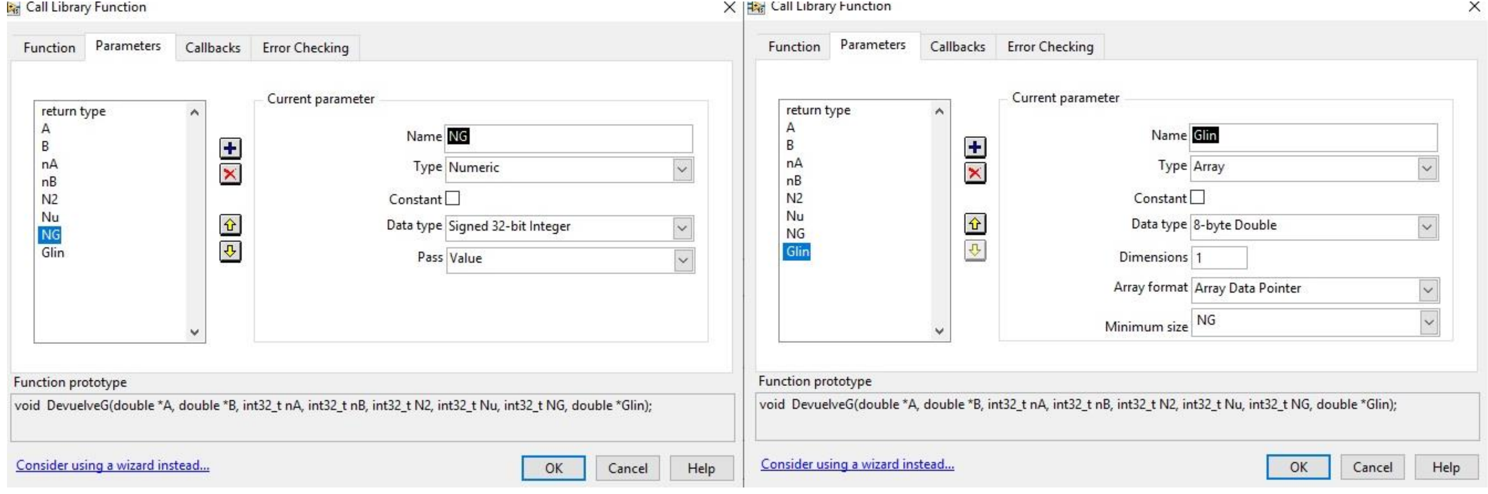


Figura 3-28. Definición variables principales de una librería en LabVIEW.

A través de LabVIEW se pueden realizar ciertas extracciones de datos y comprobaciones de valores de entrada para evitar errores de programación, que a través de C sería más difícil y tedioso, además de trasvasar una pequeña parte de carga de programación fuera de los códigos de las librerías. Lo primero que se va a añadir a la programación será un bloque que extraiga la longitud de los vectores que definen el modelo del proceso (Dentro de la paleta de funciones, "Array" > "Array Size"), ya que realizar esta tarea dentro del propio código en C puede generar incompatibilidades, si se intenta realizar a través de la longitud de los tipos de variables que forman los vectores. De esta forma se obtendrá el entero que representa la dimensión de los vectores.

En segundo lugar, se realizará una pequeña programación de un bucle "*if*" en LabVIEW, a través de la estructura "Case" (Dentro de la paleta de funciones, "Structures" > "Case Structure"), con la que se comprobará que el horizonte de control siempre sea menor o igual que el de predicción durante la ejecución, y en caso de que no se cumpla se utilizará el valor del horizonte de predicción, indicando a través de una luz de error en el panel frontal que el valor de Nu está siendo superado.

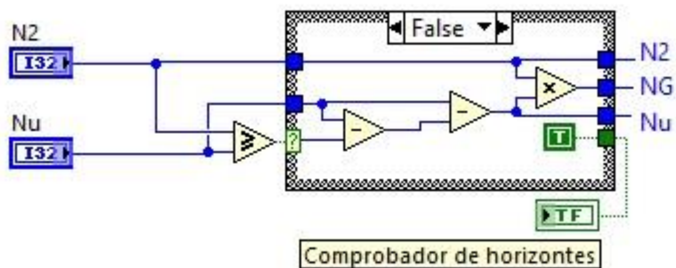


Figura 3-29. Comprobador de horizontes en LabVIEW.

Aplicando estos puntos a la programación, y aprovechando que las dimensiones de los diferentes vectores son muy fáciles de calcular a través de los bloques de operación de LabVIEW, la programación final del instrumento virtual para este ejemplo será similar a la que se puede ver a continuación (Figura *3-30*). En la que para tener un resultado más sencillo visualmente, se ha elegido dentro del desplegable de opciones del bloque de llamada de librerías, en la opción "Name Format" > "Names", lo que hará que se vean los nombre de las diferentes variables en vez de un indicador del tipo de variable.

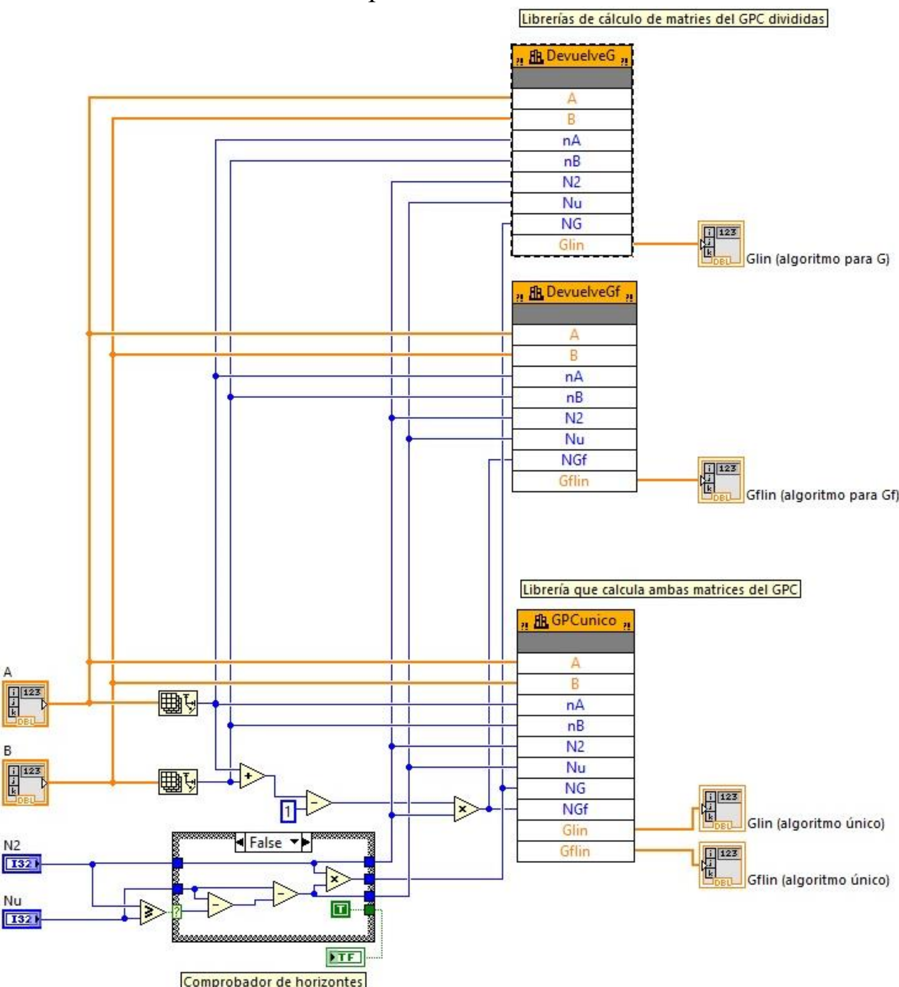


Figura 3-30. Diagrama de bloques de LabVIEW completo.

6 OPTIMIZACIÓN DE CÓDIGOS PARA SU USO EN EL ENTORNO LABVIEW

6.1 Planteamiento completo de la implementación en LabVIEW de un GPC:

En primer lugar, antes de la programación dentro del entorno en cuestión, es necesario plantear como se implementarán las diferentes partes que se comprenden dentro de un algoritmo GPC, ya que se puede abordar desde un sinfín de perspectivas según lo que se quiera conseguir. En este caso se tratará de premiar la modularidad del código de la forma eficiente, sin ser excesivos, aislando cada pequeña porción de código, pero separando las diferentes partes que cumplen funciones específicas, siguiendo un orden dentro de las declaraciones y programación en base a la relevancia y el orden de ejecución de las diferentes partes. Además se incluirá dentro de los códigos externos realizados en las librerías, la mayor parte de cálculos que sea posible, evitando las posibles incompatibilidades de código que puedan encontrarse. Con este punto puesto en perspectiva, se va a dividir la implementación en dos partes bien diferenciadas.

- Cálculo previo preparatorio del GPC.
- Algoritmo del bucle de control completo.

En lo referente al cálculo preparatorio del GPC, se tomará la decisión de separar el cálculo de las matrices principales del GPC (Matriz G y Vector de respuesta libre, Gf) en busca de la modularidad del código, para así facilitar cualquier tipo de modificación, ya que a la hora de realizar cambios, se hace bastante más sencilla la alteración de códigos que cumplen funciones lo más específicas posibles (hecho que no aplica a la hora de calificar la longitud de un código, ya que realizar una función simple puede llevar muchos cálculos y código detrás), más que códigos que abarcan la realización de varias tareas a la vez, además esta forma de programación, posteriormente si se diera el caso, ayudará a la hora de aplicar ingeniería inversa a un código para desgranar su funcionamiento.

Como apartados dentro del desarrollo del bucle de control habrá varias partes claramente diferenciadas, el cálculo de los diferentes valores de la función objetivo, el proceso de optimización a cargo de las librerías Gurobi o el proceso de actualización a final de cada ejecución del bucle. Podría ser interesante plantear la separación del proceso de optimización en una función externa dentro del apartado del bucle de control, al ser un proceso totalmente apartado y el único que requeriría la utilización de Gurobi. Pero esta decisión únicamente lo que haría, sería complicar en gran medida el proceso de comunicación entre los diferentes bloques de llamada de librerías en LabVIEW, el cual ya tiene bastantes limitaciones por cuestiones del propio entorno, por lo que finalmente se optará por realizar todo este bucle de control dentro de la misma función.

6.2 Planteamiento de la estructura de la función que realice el cálculo del GPC:

De cara al usuario que incluya un control predictivo a un sistema real monovariable, se buscará que el usuario no necesite acceder a los códigos completos del controlador, si no que este le dé la opción de sintonizar los parámetros del control, únicamente realizando la identificación del modelo (a través del método que el usuario elija según los objetivos que quiera cumplir).

Por ello lo único que se requerirá del usuario como parámetros de inicio, serán las variables que definan su modelo A, B (definidas a través del modelo en función de transferencia en discreto y términos autorregresivos), y el tiempo de muestreo del mismo y, por último, los horizontes de control y predicción que el usuario crea más convenientes para su proceso.

Posteriormente dentro del código, como se vio en el anterior ejemplo (Apartado *5.4*), se tomarán las dimensiones, previa entrada a las librerías, de las matrices representativas del modelo del proceso y se calcularán las respectivas dimensiones de las matrices de salida (necesario para que el retorno de un elemento "array" sea posible dentro de llamada de una DLL en LabVIEW), y se proseguirá con la realización de todos los cálculos preparatorios del GPC, en los bloques de las librerías vinculadas (Figura *4-1*), los cuales estarán divididos en dos bloques diferentes según su funcionalidad de cálculo.

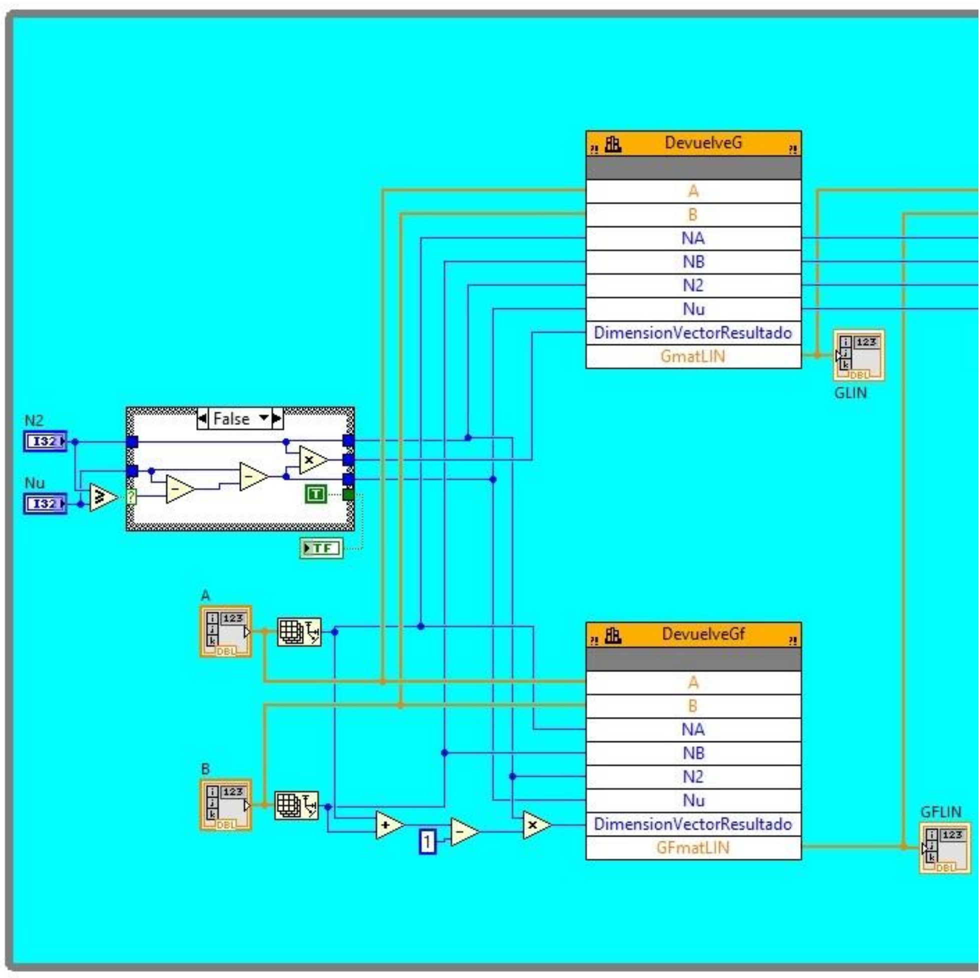


Figura 4-1. Diagrama LabVIEW para cálculo matricial del GPC

El resultado de esta parte del instrumento virtual de LabVIEW, será prácticamente igual a la programación desarrollada en el anterior ejemplo (Apartado *5.4.3*), con la salvedad de que se excluirá el bloque que realiza el cálculo conjunto de las matrices y además, como se puede observar en el extremo derecho de la figura (Figura *4-1*), las variables resultado de estos dos bloques de llamada de librerías y las variables que definen los horizontes y las dimensiones de los vectores del modelo, serán algunas de las entradas del bloque que llame a la función del bucle del GPC.

6.3 Planteamiento de la estructura de la función que realice el bucle de control:

A pesar de ya haberse abordado la forma en la que se ha implementado el bucle de control de un GPC (Apartado *4.4.4*), algunos apartados como, la actualización de variables del bucle de control y la optimización que corre a cargo de Gurobi, son segmentos del código han sido divididos previamente por su disparidad en cuanto a programación se refiere, pero son partes que van totalmente unidas entre sí, por ello a continuación se planteará la estructura completa del bucle de control.

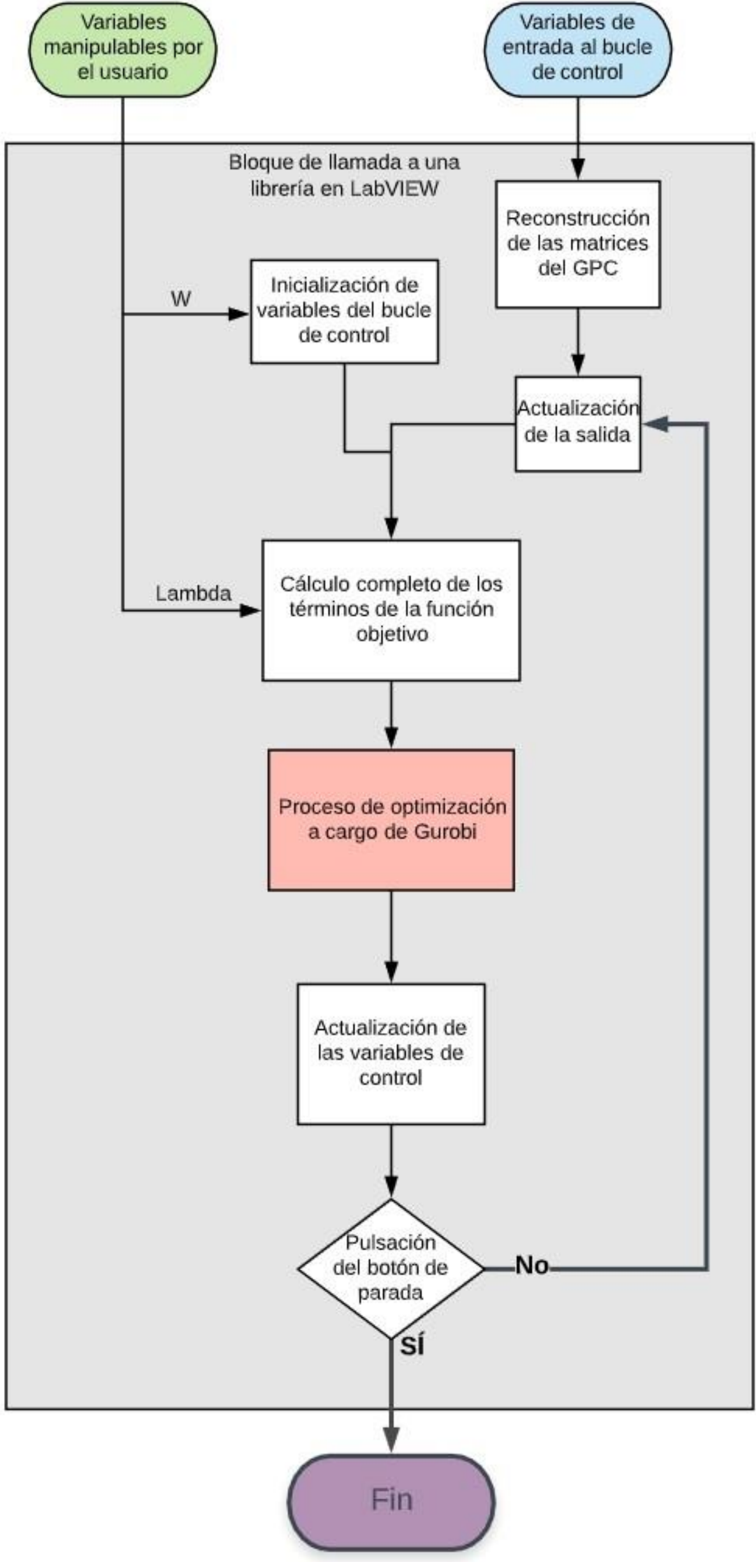


Figura 4-2. Diagrama de flujo del bucle de control del GPC

Partiendo de los detalles más relevantes expuestos anteriormente sobre el bucle de control (Apartados *4.4.3* y *4.4.4*), la estructura que se acaba de mostrar es muy similar, comenzando por una inicialización de las variables de control estáticas, construcción de las matrices de los cálculos preparatorios del GPC (ya que para enviarse dentro de LabVIEW hay que linelizarlas por filas), desarrollo completo de la función objetivo, cálculo de los parámetros cuadráticos, lineales e independiente, proceso de optimización a cargo de Gurobi y como último apartado la actualización de las diferentes variables para las señales de control, las cuales cerrarán el ciclo volviendo a ser enviadas al principio junto con la nueva salida del sistema, lo que permitirá al bucle continuar indefinidamente.

En este bucle de control aparecerán parámetros de entrada nuevos (Figura *4-3*), los cuales serán modificables por el usuario, y le servirán para manipular el controlador (modificando el parámetro de referencia W), sintonizar el comportamiento del mismo (modificando el parámetro λ), o incluso, aunque de mayor riesgo para la ejecución del código, modificar los parámetros de partida, los horizontes de control o el propio modelo que se tenga del proceso.

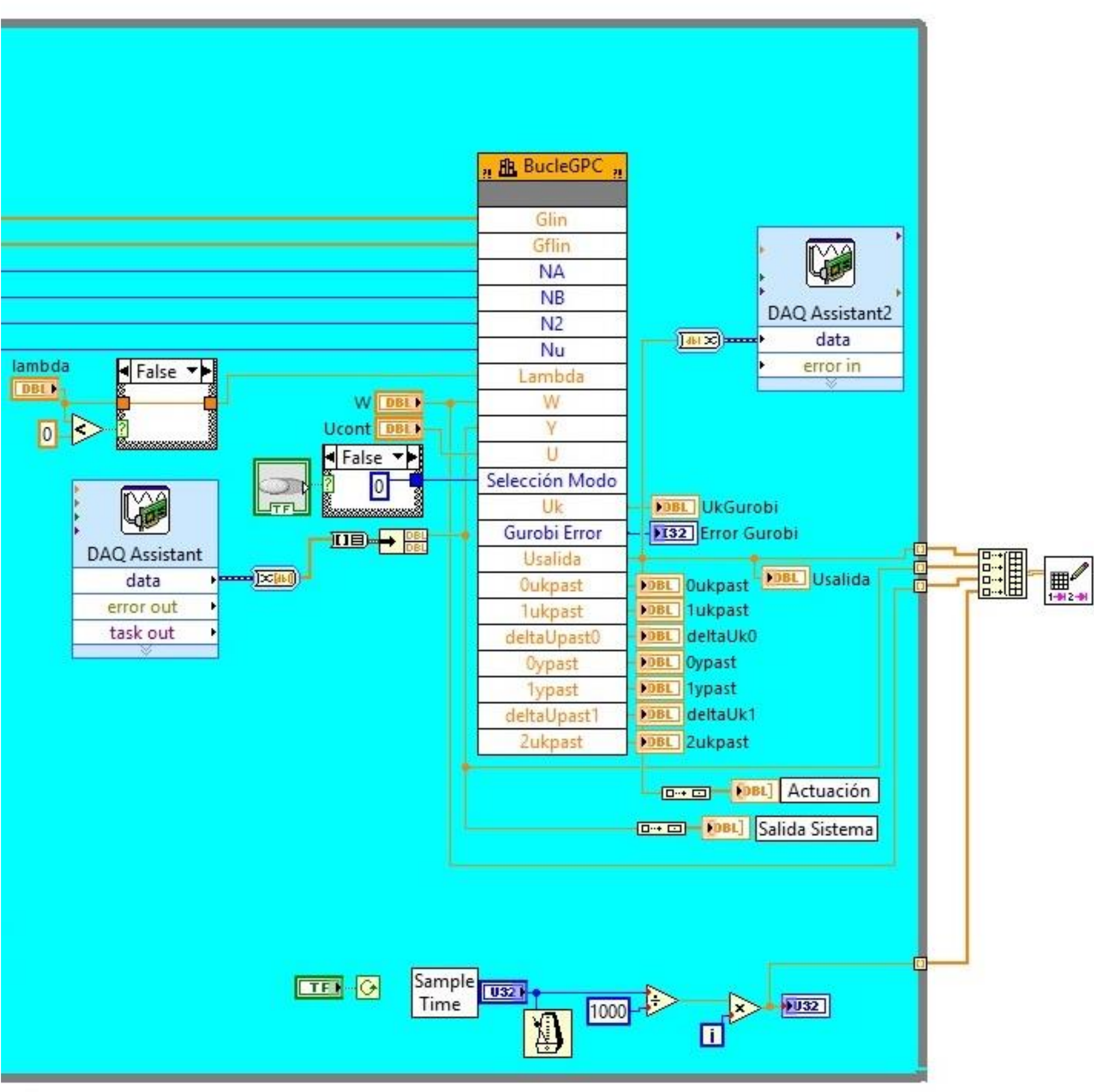


Figura 4-3. Diagrama LabVIEW para bucle de control del GPC

En la parte final del diagrama de bloques de Gurobi aparecen por primera vez los bloques de adquisición de datos (situados a izquierda y derecha del bloque de llamada de librerías, con el nombre “DAQ Assistant”), ya que este sistema al implementar la ejecución del bucle de control, será necesaria la conexión a un proceso del que se realice una toma de la salida y un envío del cálculo de la señal de control. En caso de que se precise el uso de un bucle de control predictivo compilable fuera de línea, se provee dentro del *Anexo E*, y dentro del archivo *GCPunciclo*, un *script* que ejecuta un único bucle de control para el caso en el que se precise hacer pruebas más específicas sobre apartados del propio bucle de control.

También serán destacables las últimas variables que están declaradas dentro del bloque de llamada a la librería, los cuales serán sencillamente valores extraídos para su visualización durante la ejecución y con el fin de que el usuario tenga más información aún, a la hora de evaluar el comportamiento del controlador.

6.4 Ampliación del uso de LabVIEW para mejorar las funcionalidades y limitaciones de la DLL:

En este apartado, previo al análisis de la programación completa dentro de los entornos Visual Studio y LabVIEW, se comentarán ciertos puntos de ampliación y mejora que se implementaron dentro de la aplicación posteriormente de realizadas unas pruebas mínimas. Se ha mostrado en los apartados previos algunas posibilidades a este respecto (Apartados *5.4.3* y *6.3*), como podría ser un limitador de ciertas variables manipulables por el usuario, que si tomasen ciertos valores podrían ocasionar roturas completas de los programas, o la simple toma y visualización de parámetros de la optimización en línea. Pero existen algunos puntos importantes de mejora o ampliación en la funcionalidad del código que pueden ser de interés.

- Limitación de la relación de los horizontes y del parámetro λ.

Previamente se hizo una mención al respecto de añadir un comprobador de horizontes, para así evitar que se introduzcan parámetros imposibles dentro del bucle de control que hagan que todo el programa se rompa, ya que las librerías de cálculo no serían compilables.

Por esto se deben de tomar más medidas de seguridad para los parámetros de entrada del control que son modificables por el usuario, a estos, además de la relación de los horizontes de control, se le añadirán el parámetro λ, que no podrá tomar nunca valores negativos durante la ejecución del programa.

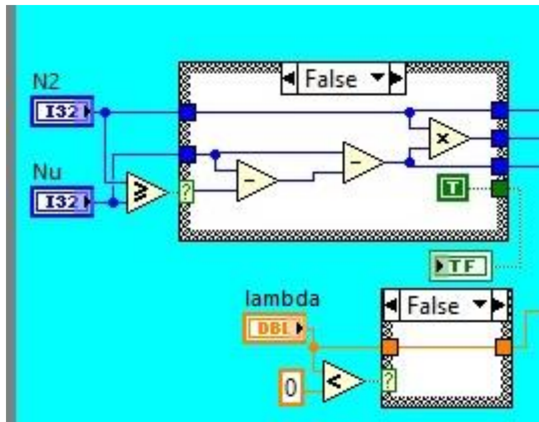


Figura 4-4. Comprobación de valores límite de variables manipulables por el usuario.

- Creación de una variable temporal para la ejecución del bucle.

El modelo creado a través del proceso, al realizarse a través de una función de transferencia en tiempo discreto, será necesaria la elección de un tiempo de muestreo adecuado al proceso en cuestión. Este tiempo de muestreo debe ser lo más reducido posible para poder representar las dinámicas del sistema, pero además, permitir la ejecución del bucle completo del GPC en cada iteración (si no se trabaja con procesos excesivamente rápidos esta elección no debería ser una tarea demasiado compleja).

A través de este tiempo de muestreo, en LabVIEW se podrá realizar una simple implementación de un bucle "*while*", con el objetivo de que la ejecución de cada uno de los bucles del control se haga dentro del tiempo de muestreo del proceso, y los datos sean tomados y actualizados a ese ritmo. Esto hará que todos los bloques sean ejecutados a una velocidad constante y no haya problemas durante la ejecución del bucle.

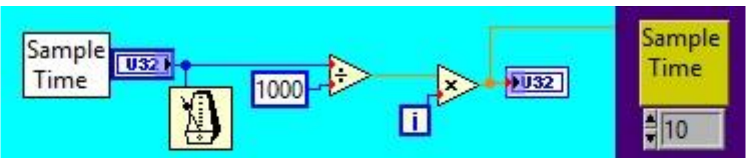


Figura 4-5. Tiempo de muestreo del modelo implementado en LabVIEW.

- Creación de gráficas para la visualización de ciertas variables relevantes.

A la hora de ejecutar cualquier algoritmo de control, ya sea predictivo o no, uno de los apartados imprescindibles para el análisis del comportamiento del controlador se realiza a través de diferentes gráficas, en estas se pueden analizar los valores más relevantes del control respecto a tiempo de ejecución, esto no puede ser de otra forma en el caso que se está tratando. Por ello, esta vez gracias al entorno LabVIEW, se pone a disposición la posibilidad de realizar una visualización gráficas de las variables que se necesite, en línea (Figura *4-6*).

Figura 4-6. Visualización gráfica de variables en LabVIEW.

La realización de extracción de información gráfica de manera tan relativamente sencilla, es un punto de gran mejora para el usuario o programador y a sus facilidades de rápida actuación o visualización de errores.

- Implementación de un modo automático y un modo de control predictivo.

Una sencilla ampliación al uso de LabVIEW, que influirá positivamente en el interfaz de usuario y sus posibilidades de actuación, podría ser un actuador que permita activar o no la entrada del control predictivo a la ejecución del proceso o mantenerlo en un modo automático (o cualquiera que se plantee).

Figura 4-7. Selector de modo de control implementado en LabVIEW.

Un selector de modo de control podría ser además ampliable si se precisan diferentes modos de control, en el caso de que se quisiera buscar una activación manual además de dentro de la programación (Figura *4-7*), método para el que LabVIEW no ofrece muchas dificultades a la hora de programarse, pudiendo añadirse multitud de casos distintos sin dificultad.

Este método tiene una ventaja muy clara en el apartado de seguridad del proceso, ya que, por ejemplo, si se tratase el caso de una primera ejecución del proceso, no sería extraño que se dé una señal de arranque que en caso que se tenga el controlador activo precise un esfuerzo de control muy grande que no está dentro de los cálculos previstos, si se realiza este proceso de encendido con el controlador desactivado habría menos posibilidades de que algo fuera de los planes ocurra.

- Visualización de parámetros de optimización en línea.

Este apartado ya ha sido tratado previamente, por la problemática que ofrece LabVIEW y su trato con las variables de retorno desde los bloques de llamada de librerías, pero en este caso, tratándolo a través de la existencia de un modo de ejecución automático, en el que no intervenga el controlador, podría ser muy interesante, a pesar de su inacción, que el controlador predictivo en todo momento esté siendo compilado dando valores por pantalla de las sucesivas ejecuciones pero sin afectar al proceso, permitiendo así al usuario decidir en qué momento quiere que el controlador entre en acción, o de forma contrario, hacer que el control deje de actuar en el proceso, ya sea porque se quiere cambiar de método o que ya no sea necesario.

Figura 4-8. Visualización de parámetros a través de una llamada a librería en LabVIEW.

6.5 Ejemplo completo de implementación de un GPC en LabVIEW:

Aplicando las mejoras y ampliaciones comentadas al código del que se partía en el ejemplo anterior, y sumando el desarrollo del algoritmo del bucle de control predictivo, el código completo estaría preparado para ser implementado al proceso real monovariable que se precise. Tomando unos valores para el modelo del proceso en formato de función de transferencia, un tiempo de muestreo y unos valores de horizontes adecuados, el código será ejecutable. Presentando al usuario un interfaz con la siguiente estructura:

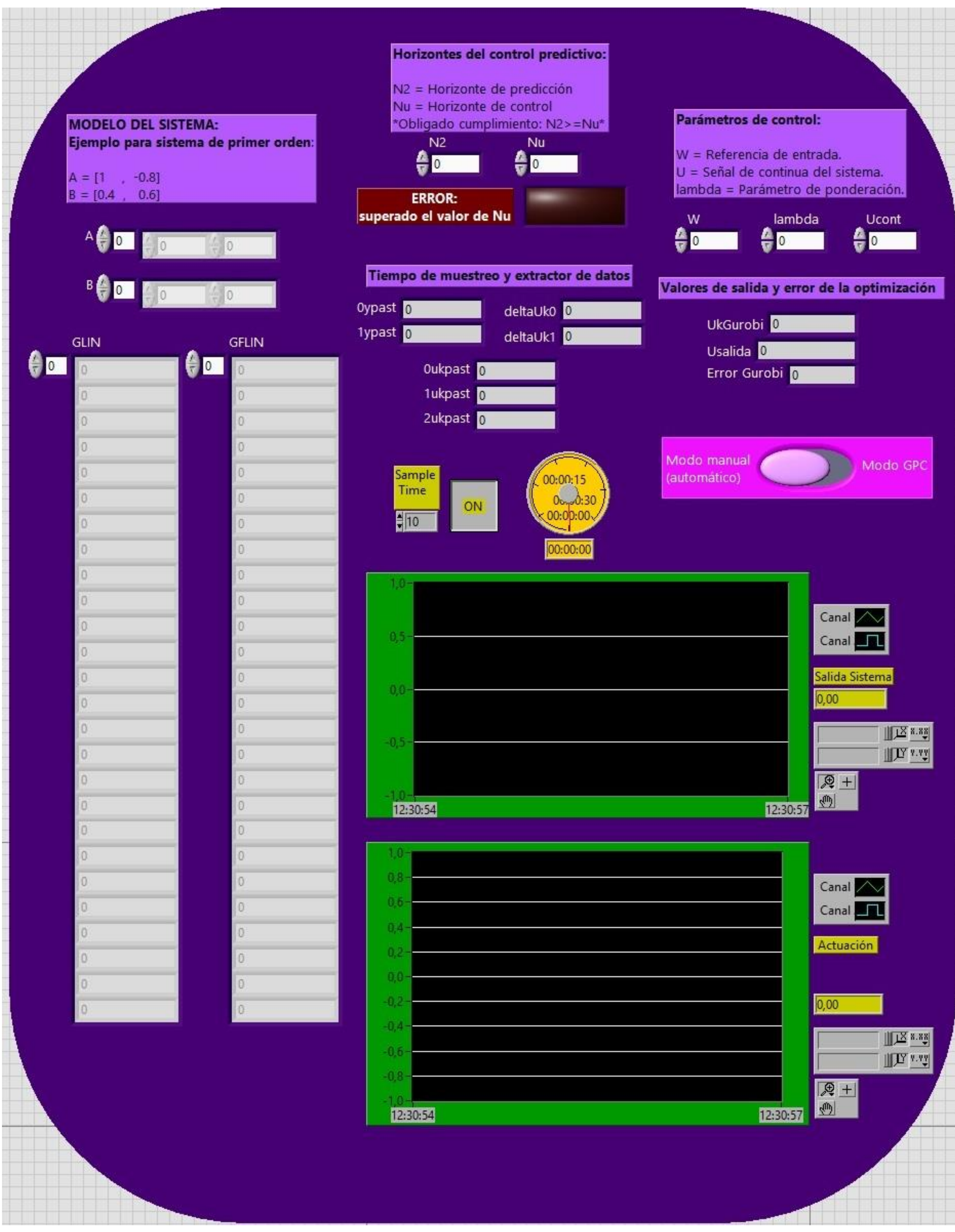


Figura 4-9. Interfaz de usuario en LabVIEW.

El diagrama de bloques que existe detrás del interfaz anterior, será la suma de las dos figuras de LabVIEW representadas en los apartados *6.2* y *6.3*, cuyo resultado será el siguiente:

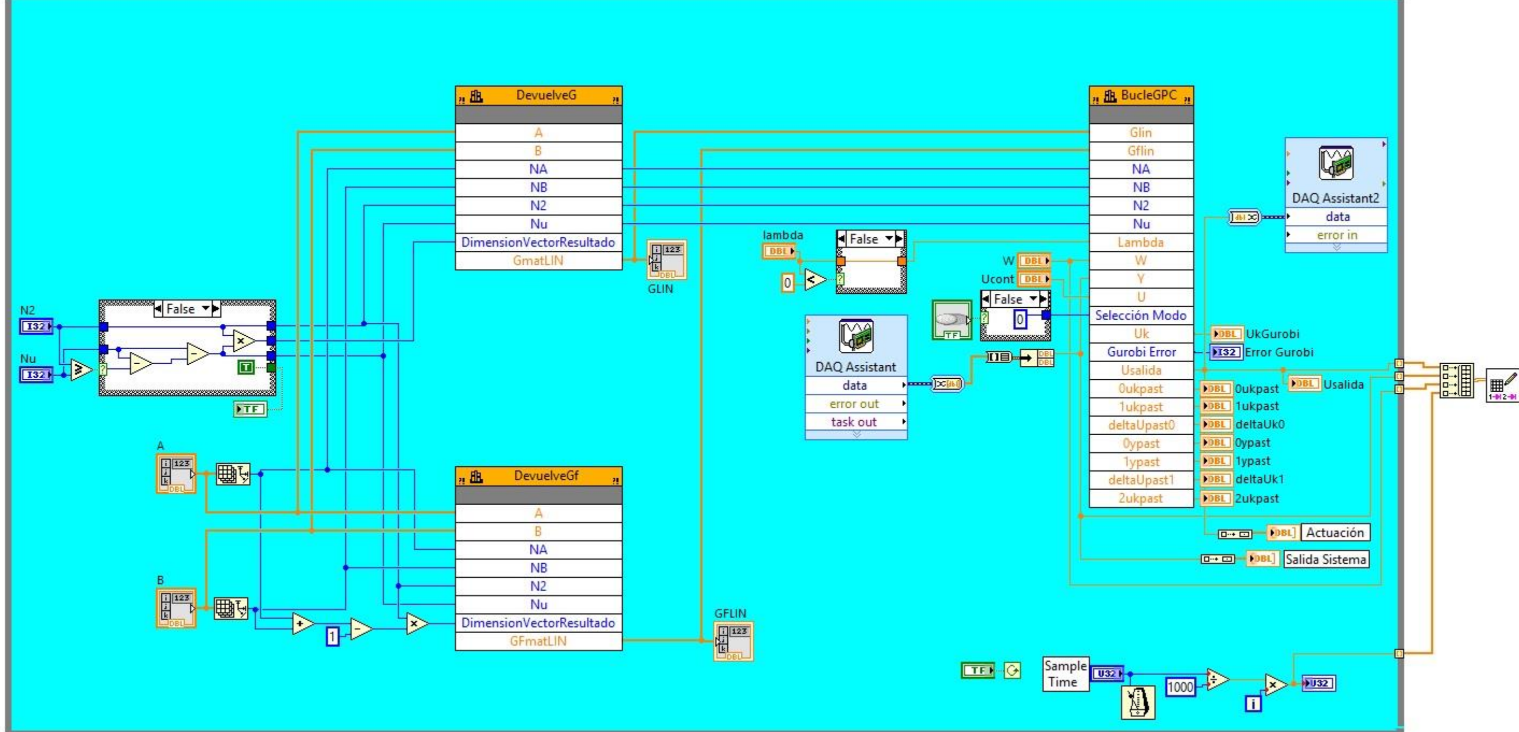


Figura 4-10. Diagrama de programación de LabVIEW completo.

A continuación no se analizarán todos los apartados del código de la librería, ya que al margen de ser relativamente extensa, ya han sido analizadas previamente casi en su totalidad por ello se destacarán únicamente los apartados de nueva inclusión, haciendo eso sí, referencia a las partes del código ya analizadas.

Partiendo de la programación preparatoria del GPC analizada en el ejemplo anterior (Apartado *5.4*), en las que se tienen dos funciones exportables a LabVIEW, las cuales van a proveer a la última función de cálculo del bucle de control (*BucleGPC*), las matrices en cuestión, linealizadas por filas, dimensiones del modelo, horizontes y parámetros de control y diseño (parámetro λ, referencia futura y modo de control).

Por ello, como habitualmente, el primer paso dentro del bucle será la inicialización y reconstrucción de las matrices (Figura *4-11*), seguida de la inicialización de las variables de control (Apartado *4.4.4*, Figura *2-69*) y de la posterior actualización de valores de entrada del proceso (Figura *2-70*).

```
/*
    INIALIZACIÓN Y RECONSTRUCCIÓN DE LAS MATRICES PARA EL CÁLCULO DEL GCP
*/

//Reserva de memoria para las matrices
double **G;
G = reservarMAT(N2, Nu);
double **Gf;
Gf = reservarMAT(N2, (nA + nB));

//Reconstrucción de las matrices linealizadas por filas

ConstMat(Glin, N2, Nu, G);
ConstMat(Gflin, N2, (nA + nB), Gf);
```

Figura 4-11. Inicialización y reconstrucción de matrices del GPC.

Terminado el proceso de inicialización de todas las variables necesarias, se debe proseguir con el desarrollo completo de los términos de la función objetivo (Apartado *4.4.3*), los cuales permitirán que el proceso de optimización se lleve a cabo a cargo de Gurobi. La optimización en Gurobi siempre debe empezar con la declaración de un entorno y al menos un modelo (Apartados *3.5.2* y *3.5.3*), el cual siempre estará por defecto vacío, ya que las variables se añadirán a continuación (Figura *4-12*).

```
/*
    DECLARACIÓN DE VARIABLES NECESARIAS PARA LA OPTIMIZACION EN GUROBI
*/

GRBenv *env = NULL;
GRBmodel *model = NULL;


double *deltaU;
deltaU = reservarVEC(Nu);

/*
    INICIALIZACIÓN DEL ENTORNO Y EL MODELO
*/

//CREACION DEL ENTORNO\\

*error = GRBloadenv(&env, "NuevaUK.log");
if (*error != 0) { Erroracion(env); }

//CREACION DEL MODELO VACIO\\

*error = GRBnewmodel(env, &model, "NuevaUK", 0, NULL, NULL, NULL, NULL, NULL);
if (*error != 0) { Erroracion(env); }
```

Figura 4-12. Inicialización entorno y modelo de Gurobi.

La adición de variables al modelo vacío (Apartado *3.5.4*), es el siguiente paso lógico de la declaración del proceso en Gurobi, se declaran todas las variables existentes, ya sean lineales o cuadráticas, aunque se deberán utilizar funciones diferentes para cada tipo de variable (Figura *4-13*).

```
/*
    ADICIÓN DE LOS TODAS LAS VARIABLES AL MODELO VACÍO
*/

//ADICIÓN DE VARIABLES LINEALES\\
//Tendremos tantas variables lineales iguales a Nu

*error = GRBaddvars(model, Nu, 0, NULL, NULL, NULL, linval, NULL, NULL, NULL, NULL);
if (*error != 0) { Erroracion(env); }

//ADICIÓN DE VARIABLES CUADRÁTRICAS\\

*error = GRBaddqpterms(model, Nqvar, qfil, qcol, qval);
if (*error != 0) { Erroracion(env); }

//ADICIÓN DEL TÉRMINO INDEPENDIENTE\\

*error = GRBsetdblattr(model, GRB_DBL_ATTR_OBJCON, f0);
if (*error != 0) { Erroracion(env); }
```

Figura 4-13. Adición de todas las variables al modelo de Gurobi.

En caso de que al modelo se le desee añadir restricciones, se le deberá añadir la declaración de las mismas en esta parte del código, al igual que si se desea que los valores límite del proceso puedan ser modificables, se podrán añadir como parámetro de entrada al bucle, para poder ser modificado por el usuario.

La declaración de restricciones se llevará a cabo a través de la función correspondiente de Gurobi (Apartado *3.5.6*), pero con la salvedad de que se tendrá que realizar un bucle completo para añadir la restricción superior e inferior de los valores a todas las variables inicializadas en Gurobi (Figura *4-14*) (las cuales se conoce que son la misma señal de control en los instantes de tiempo futuros).

```
//ADICIÓN DE RESTRICCIONES DE VARIABLES DE CONTROL\\

int indcU[] = { 0 };
double linvalU[] = { 1 };

for (i = 0;i < Nu;i++) {

    *error = GRBaddconstr(model, 1, indcU, linvalU, GRB_LESS_EQUAL, Ulim, NULL);
    if (*error != 0) { Erroracion(env); }

    *error = GRBaddconstr(model, 1, indcU, linvalU, GRB_GREATER_EQUAL, -Ulim, NULL);
    if (*error != 0) { Erroracion(env); }

    indcU[0] += 1;
}


//ADICIÓN DE RESTRICCIONES A LA SEÑAL DE SALIDA\\
//En primer lugar debemos realizar los cálculos despejando al vector U, que es en torno al que se hace la restricción

//Inicializamos y reservamos espacio para las variables que contiene los indices de las variables lineales
int *indcY;
indcY = (int*)calloc(Nu, sizeof(int));
if (indcY == NULL) {
    printf("No se ha podido reservar memoria.");
    exit(1);
}

for (i = 0;i < Nu;i++) {
    indcY[i] = i;
}

//Y además los valores de G ordenados por filas
double* linvalY;
linvalY = reservarVEC(Nu);


for (j = 0;j < N2;j++) {

    for (i = 0;i < Nu;i++) {
        linvalY[i] = G[j][i];
    }

    *error = GRBaddconstr(model, Nu, indcY, linvalY, GRB_LESS_EQUAL, Ylim, NULL);
    if (*error != 0) { Erroracion(env); }

}
```

Figura 4-14. Adición de restricciones al modelo de Gurobi.

Como se tienen que realizar ciertas modificaciones al código del bucle de control, al igual que al código en LabVIEW, en el caso que se quieran añadir restricciones al proceso, se adjuntarán a este proyecto un modelo y archivo de librerías completo en el que se excluya la adición de restricciones (*GPCcompleto*), además se puede acceder a la transcripción del código en el *Anexo F*. Y como es obvio, otro que sí contenga este último añadido al código (*GPCrestrict*), cuya transcripción del código está disponible en el *Anexo G*.

```
/*
    COMPROBACIÓN DEL SENTIDO DE LA OPTIMIZACIÓN
*/

//Para control predictivo siempre se elegirá minimización
*error = GRBsetintattr(model, GRB_INT_ATTR_MODELSENSE, GRB_MINIMIZE);
if (*error != 0) { Erroracion(env); }


/*
    OPTIMIZACION DEL MODELO
*/

*error = GRBoptimize(model);
if (*error != 0) { Erroracion(env); }

/*
    CAPTURA DE SOLUCION
*/

*error = GRBgetdblattrarray(model, GRB_DBL_ATTR_X, 0, Nu, deltaU);
if (*error != 0) { Erroracion(env); }


/*
    LIBERACIÓN DEL MODELO Y DEL ENTORNO (en este orden específico)
*/

GRBfreemodel(model);
GRBfreeenv(env);
```

Figura 4-15. Optimización, captura de solución y liberación de Gurobi.

Realizada todas las llamadas previas necesarias a las funciones de Gurobi, para que se tengan todos los datos del proceso que se va a optimizar, se debe tomar el último paso en lo referente al uso de las librerías (Figura *4-15*), comprobación del sentido de la optimización (Apartado *3.5.5*), optimización del modelo (Apartado *3.5.7*), captura de solución (Apartado *3.5.8*) y liberación del sistema (Apartado *3.5.9*) para finalizar el uso de Gurobi.

```
/*///////////////////////////////
                               //
        SELECCION DE MODO      //
                               //
*///////////////////////////////


//Actualizamos el valor de la variable de salida según el modo elegido
if (Modo == 0) {

    *Ugurobi = deltaU[0];
    *Usal = U;

}
else {

    *Ugurobi = deltaU[0];
    *Usal = U + deltaU[0];

}
```

Figura 4-16. Selección de modalidad de control.

Terminado el proceso de optimización, el bucle de control ha cumplido casi todas las funciones para las que estaba dispuesto, en este punto dependiendo de si se introdujo un selector de modos de ejecución, se realizará la comprobación correspondiente (Figura *4-16*) y el envío de una señal u otra a la variable de salida. Y a continuación se realizará la actualización de valores del bucle de control (*Figura 4-17*), preparando así el algoritmo para la siguiente ejecución y enviando además los parámetros de visualización que se precise de la ejecución del bucle de control actual.

```
/*
    ACTUALIZACION DE VALORES DEL BUCLE DE CONTROL
*/

//Se deben actualizar los dos vectores referentes a las señales de control pasadas


//En primer lugar se actualiza el vector variaciones de control pasadas
for (i = nB - 1; i > 0; i--) {
    deltaUpast[i] = deltaUpast[i - 1];
}
deltaUpast[0] = deltaU[0];

//A continuación se actualiza el vector de salidas de control pasadas
for (i = nB; i > 0;i--) {
    ukpast[i] = ukpast[i - 1];
}
ukpast[0] = *Usal;

*ukpast0 = ukpast[0];
*ukpast1 = ukpast[1];
*ukpast2 = ukpast[2];
*deltaupast0 = deltaUpast[0];
*deltaupast1 = deltaUpast[1];
*ypast1 = ypast[1];
*ypast0 = ypast[0];
```

Figura 4-17. Actualización del bucle de control y toma de variables en línea.

6.6 Problemas de integración de una DLL que utiliza las librerías Gurobi dentro de LabVIEW:

Es importante recalcar, que de igual manera que la inclusión y utilización de las librerías de optimización Gurobi dentro de Visual Studio, puede no llegar a ser un proceso tan directo como puede parecer teóricamente, la compilación de códigos externos, que además de realizar ciertos cálculos que puedan ser más o menos complejos, contienen un proceso de optimización con Gurobi, es aún menos directo y puede llevar a errores constantes, roturas completas del programa o incluso errores los cuales no se pueda saber muy sencillamente de donde proceden.

Esto sucede, gracias a la mezcla de una variedad de entornos de programación diferentes, que no tienen por qué ser compatibles en todas sus características o tener diferentes apartados en los que exista un conflicto y no permita su ejecución. Por esto se analizarán a continuación ciertos problemas de integración de entornos bastante comunes:

- Utilización de algunas funciones en C pueden inhabilitar cálculos completos sin producir errores.

Como es obvio, hay multitud de posibilidades a la hora de afrontar cualquier tipo de cálculo o programación, es por esto que puede ocurrir que alguno de los métodos elegidos para afrontar alguna tarea, no sea el ideal para la integración en entornos diversos, esto en el caso de programación en lenguaje C, puede entrañar problemas que no son fácilmente detectables, por ejemplo, para el cálculo de las dimensiones de un vector.

```
double A[] = { 1,  -0.8 };
double B[] = { 0.4, 0.6 };

int nA = (sizeof(A) / sizeof(double));
int nB = (sizeof(B) / sizeof(double));
```

Figura 4-18. Cálculo de dimensiones de vectores erróneo.

Abordando el simple cálculo de la forma representada anteriormente (Figura *4-19*), en casos de implementaciones sencillas en las que no se busque externalizar el código, no debería haber problemas, pero en el caso en el que se quiera diseñar una librería para ser exportada a otro entorno, programaciones de este tipo pueden dar problemas devolviendo valores nulos o un valor

Además otro punto, no de incompatibilidad, pero de utilización errónea de ciertas funciones dentro de LabVIEW, podría ser la función de C, "*exit ()*", la cual si en algún momento es llamada durante la ejecución de un código a través de una librería, ésta en vez de realizar el comportamiento esperable en C, terminando la ejecución del código únicamente, lo que ocurrirá será un cierre completo de la herramienta LabVIEW con la necesaria recuperación del archivo con el que se estaba trabajando. Por situaciones como estas, es importante tener cuidado con las funciones que se intentan integrar a través de varios entornos de programación a la vez.

- No es sencilla una monitorización del algoritmo en toda su ejecución, menos aún dentro de Gurobi.

Durante la ejecución de códigos más o menos complejos, la depuración y corrección de errores acaba siendo el apartado más trabajoso, pero en el caso de trabajar con librerías en las que se utilicen las librerías Gurobi, resultará más difícil que la depuración de un script de la misma complejidad.

En primer lugar, la forma más útil que ofrece Gurobi a la hora de detectar errores es una única función (Apartados *3.4.1.1* y *3.5.1*) que en caso de que exista un error, dará de vuelta un comando que se referirá a una lista de errores predefinida de la cual se deben extraer las conclusiones del error que está ocurriendo. Esta forma de detección no está mal pero resulta a veces incompleto, puede ocurrir que se encuentre algún error no definido previamente, además si se trabaja a través de LabVIEW, si no se destina una variable explícitamente para la extracción de la variable de error, no se podrá obtener ninguna información a través del código sobre lo que ocurre dentro de la librería que resulta en un paro de la ejecución o en una rotura del programa.

- La escritura de archivos y toma de información en línea puede resultar en roturas del programa.

Dentro de las posibilidades que Gurobi ofrece dentro de sus funcionalidades al margen de la optimización, es la escritura de archivos con los modelos utilizados o los resultados obtenidos, y generalmente no debería de haber ningún problema para la realización de scripts o códigos sencillos, pero como se viene analizando previamente, la compatibilidad entre plataformas es algo que no viene asegurado, al margen de que el trabajo de escritura puede ser un proceso más largo de lo que se puede permitir dentro de un proceso, haciendo que el programa en la misma ejecución no pueda ir lo suficientemente rápido.

En especial, con el uso de las librerías Gurobi y sus funciones de escritura, a través del entorno LabVIEW, ofrecerán multitud de errores de escritura, al no encontrar la ruta del archivo en el que se va a realizar la escritura o no poder acceder a un archivo ya existente para continuar la escritura de la última ejecución, cuyos comandos de error están remarcados dentro de la información que el grupo de desarrollo de Gurobi pone a disposición, pero existe la posibilidad incluso, de que durante la ejecución no se cree ningún archivo de escritura y el propio Gurobi no alerte al usuario de que exista un error siquiera.

Por estos resultados y multitud más, se desestimó la utilización de las funciones de escritura de Gurobi dentro de los códigos finales, y se optó por la toma de datos en línea, lo cual da muchas más opciones que la escritura de resultados a posteriori de la ejecución.

7 APLICACIÓN DE LABORATORIO

7.1 Presentación del sistema real:

Partiendo del objetivo de la programación de un controlador predictivo para un proceso genérico monovariable, finalizado en el Apartado *4*, se debe poner a prueba la programación realizada integrando cada una de las partes desarrolladas individualmente:

- Proceso de optimización de Gurobi (Apartado *3*).
- Programación de archivos de librerías dinámicas para el cálculo del controlador en Visual Studio (Apartado *5*).
- Implementación del control predictivo en el entorno LabVIEW (Apartados *5.3, 5.4* y *6*).

Para todo el proceso que se va a desarrollar en los siguientes apartados se debe presentar el sistema con el que se va a trabajar, el cual estará formado por dos elementos fundamentales,

- El propio proceso real.

Para la elección del sistema físico, como se anticipaba al principio de la programación del control predictivo (Apartado *4.4*), se tomará uno de los sistemas disponibles en el laboratorio de control de la Escuela, en este caso se utilizará un secador (Figura *5-1*), el cual como requisito indispensable, ofrece la posibilidad de extraer las funciones del controlador al exterior del proceso, lo que será totalmente necesario a la hora de conectar el proceso con toda la programación realizada anteriormente para el controlador predictivo (o para el tipo de controlador que se haya diseñado en su caso).

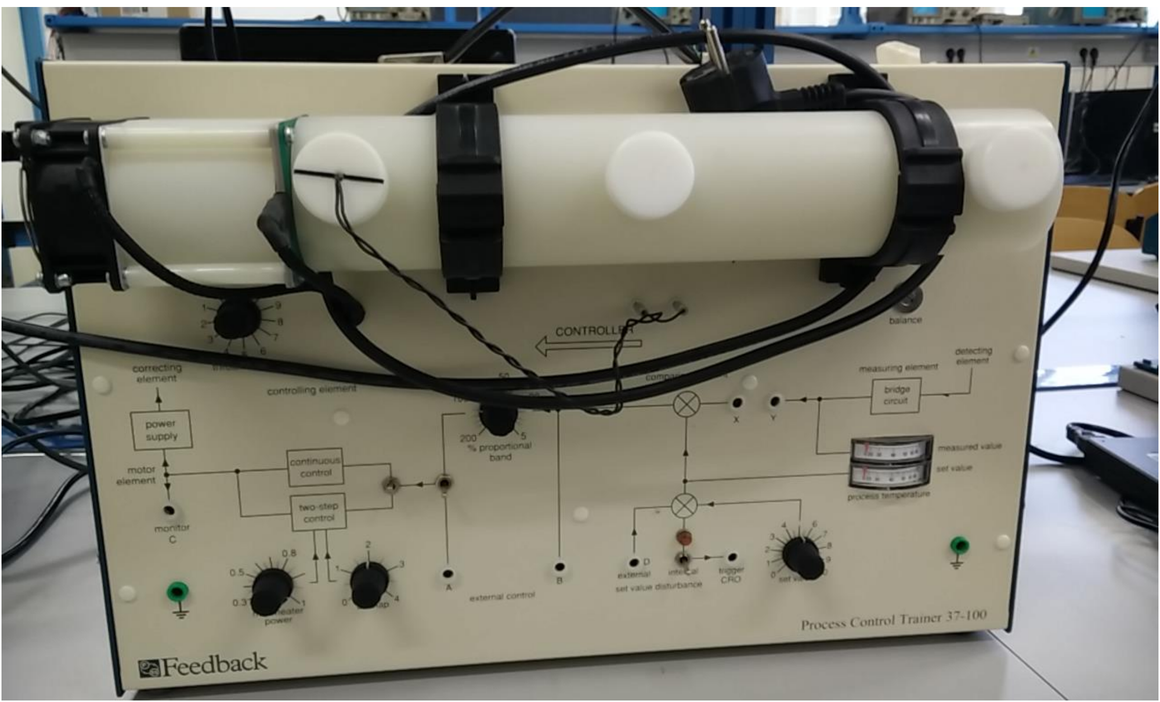


Figura 5-1. Sistema físico elegido.

- El sistema de comunicación entre el proceso y las diferentes partes de la programación.

Con el fin de realizar la comunicación entre el proceso y la programación del controlador dentro del ordenador, se utilizará una tarjeta de adquisición de datos (Figura *5-2*), esta no sólo servirá para la toma de los datos de la salida del sistema necesarios para el control, sino que también permitirá que se envíen al proceso las señales de control calculadas por el controlador predictivo.

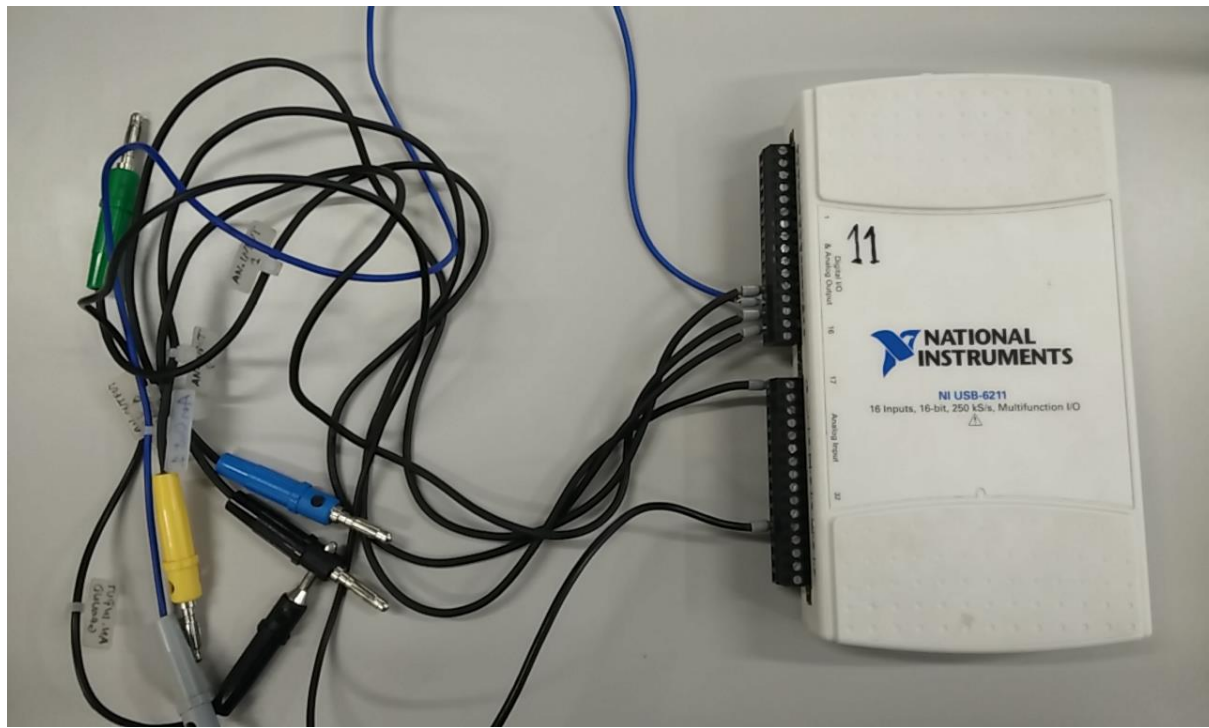


Figura 5-2. Tarjeta de adquisición de datos.

7.1.1 Conexión del sistema real y la tarjeta de adquisición:

Para este caso concreto al tomarse un proceso relativamente sencillo, únicamente se dispondrá del equipo correspondiente al sistema real y de una tarjeta de adquisición. Esta tarjeta será el método de conexión existente entre el proceso y toda la programación realizada previamente (que será contenida dentro de un único instrumento virtual en el entorno LabVIEW). Para realizar la conexión de estos tres elementos la mayor complicación reside a la hora de conectar el sistema y la tarjeta de adquisición de datos, ya que desde la tarjeta hasta el ordenador se utilizará una simple conexión a través de un puerto *USB*.

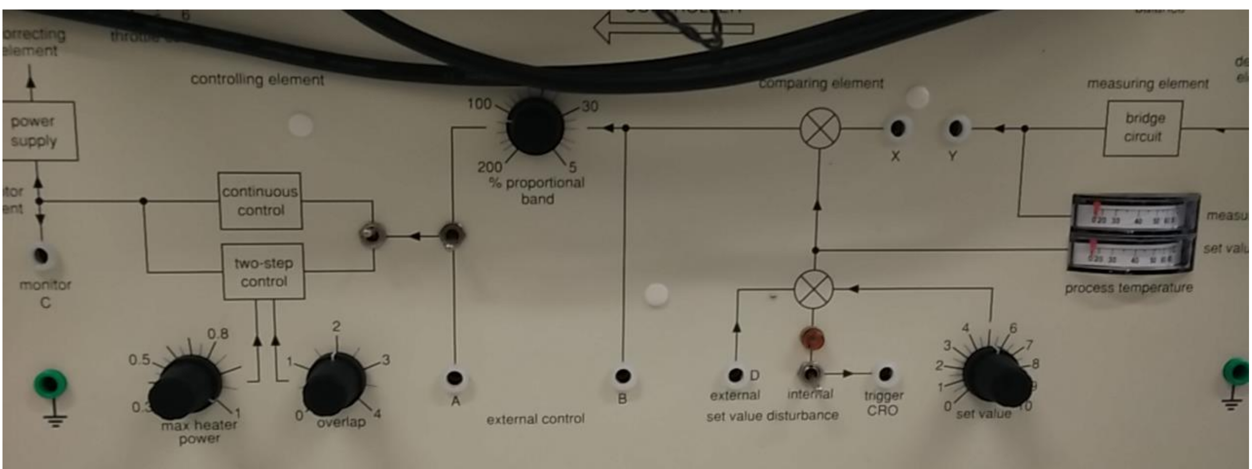


Figura 5-3. Frontal de conexiones del sistema físico.

Conociendo el panel frontal de las salidas del sistema (Figura *5-3*), como principal consideración a seguir para realizar una conexión correcta entre la tarjeta de adquisición y el sistema real, es la inversión de las entradas y las salidas entre los dos elementos, esto quiere decir que, lo que se reconoce como entrada y salida del sistema se conectarán de manera opuesta en la tarjeta de adquisición, correspondiendo la salida del sistema a una de las entradas de la tarjeta, Analog Input (Entrada analógica), y al contrario para la entrada del sistema se dispondrá una salida de la tarjeta, Analog Output (Salida Analógica). Este cambio y se puede ver representado en las siguientes figuras (Figura *5-4*).

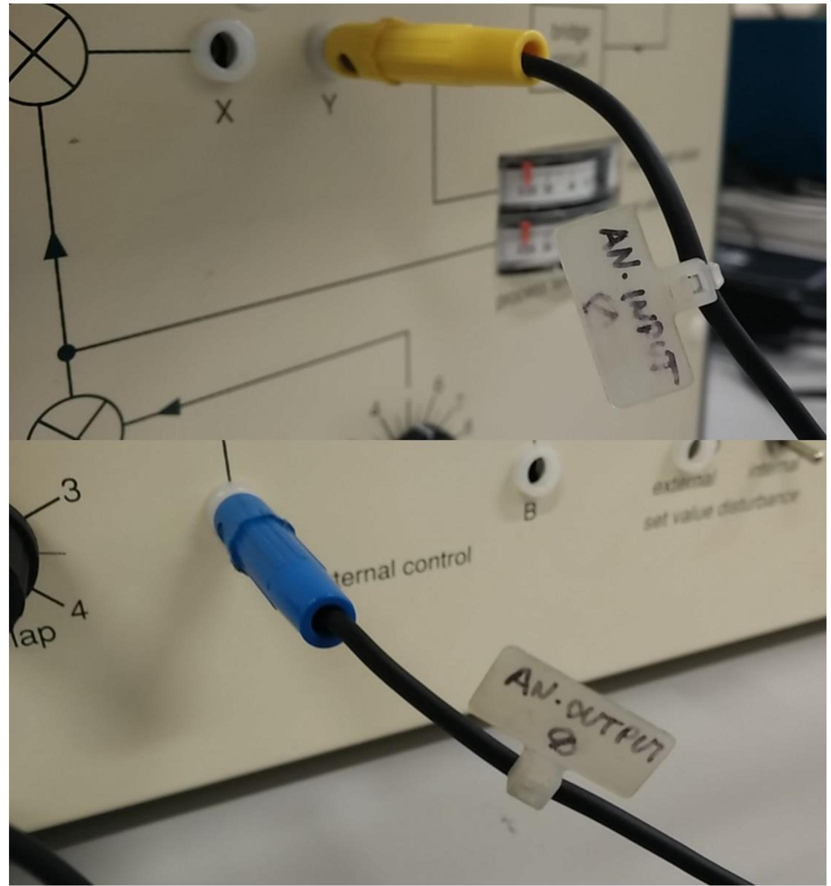


Figura 5-4. Conexión de la salida (arriba) y la entrada (abajo).

Por último la última conexión que se deberá realizar serán las tomas de tierra de la tarjeta de adquisición, que irán a una de las tomas de tierra del sistema como se puede observar a continuación (Figura *5-5*).

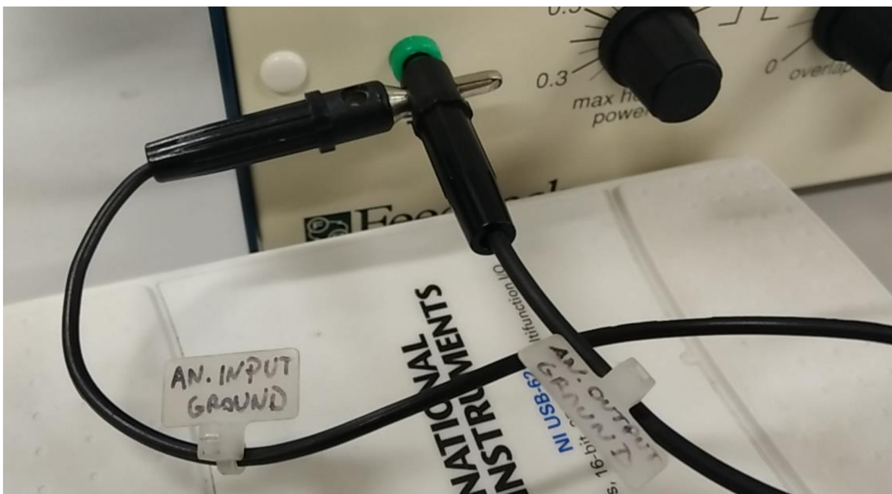


Figura 5-5. Conexión de las tierras.

Pero la labor de conexión entre los elementos no termina aquí, ya que a través de LabVIEW habrá que identificar los puertos utilizados señalados dentro de la propia tarjeta de adquisición. Accediendo a cada uno de los bloques de adquisición de datos, después de un tiempo de inicialización se abrirá una ventana en la que habrá que hacer click derecho en el desplegable de canales que aparecen en la parte inferior, y acceder a la opción "Change Physical Channel…" (Figura *5-6*), apareciendo a continuación una ventana donde estarán todos los canales disponibles para hacer la selección del correcto. Esto deberá repetirse tantas veces como elementos para adquisición de datos existan en el instrumento visual con el que se trabaje.

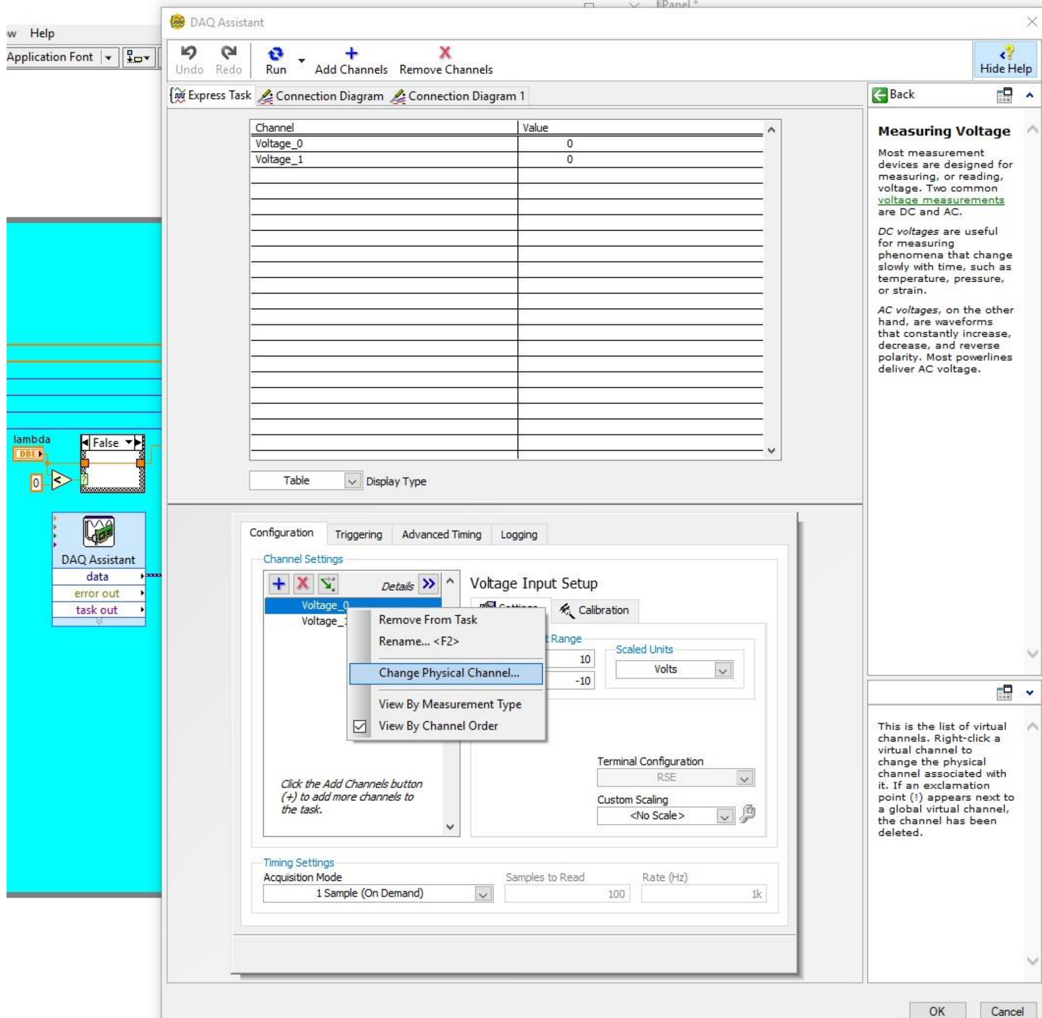


Figura 5-6. Menú del bloque de adquisición de datos de LabVIEW.

7.1.2 Identificación del sistema:

En los anteriores apartados, relacionados con la programación genérica de las diferentes partes de un controlador predictivo, fue obviado uno de los apartados más relevantes, y en torno a los que se basan los MPCs, el modelo del proceso a controlar. Por la naturaleza del desarrollo de este trabajo, no era necesaria la existencia de un proceso real con su correspondiente modelado, ya que uno de los requisitos es la generalidad ante el proceso a controlar, pero en este punto del desarrollo como ya se dispone de un sistema real en concreto para la experimentación, será necesaria la creación de un modelo del sistema.

El primer paso a llevar a cabo es la identificación del mismo, por lo que partiendo de lo expuesto en el Apartado *4.1.2*, se debe tener presente que la complejidad o precisión del modelo dependerá del diseñador, premiando la sencillez para su cálculo (como se elegirá para el cálculo de este apartado) o incluso realizando un modelado de manera recursiva. La toma de una decisión u otra repercutirá en el comportamiento del controlador, por lo que cuanto mejor se repliquen las dinámicas del proceso en el modelo, mejor será el resultado final de controlador.

En el caso de este trabajo se hará un modelo basado en una función de transferencia sencilla de primer orden, a través de una identificación de respuesta ante una señal en escalón. Esto se puede realizar sencillamente a través del instrumento virtual de LabVIEW completo creado anteriormente, ejecutando el modo automático y únicamente modificando el valor "Ucont" del panel de control. Y a través de este proceso se enviarán al sistema diferentes señales en escalón que combinen flancos de subida y flancos de bajada en torno a los mismos valores de la señal de control (Figura *5-7*).

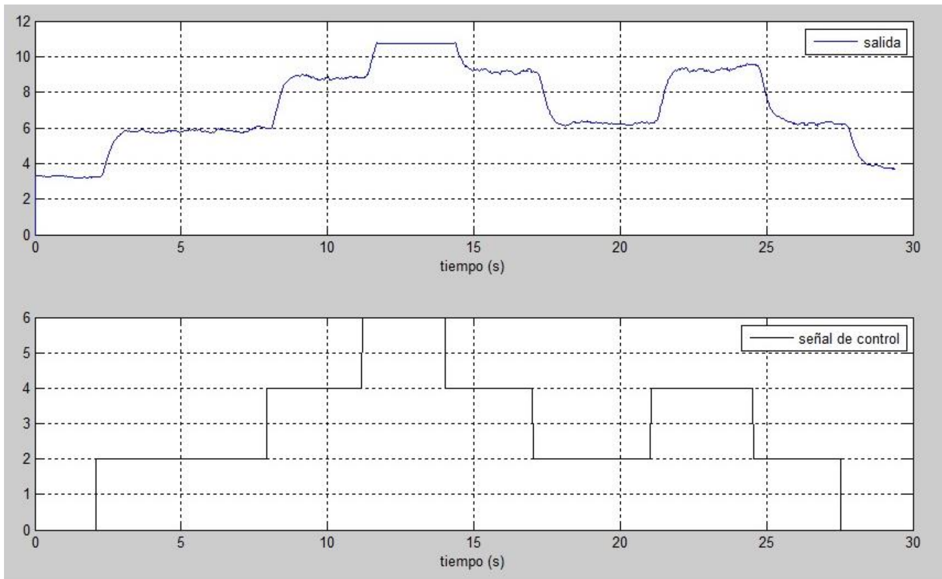


Figura 5-7. Señal para realizar la identificación del sistema.

Tomando dos flancos, uno de subida y otro de bajada, de la señal de control donde toma los valores de 2-4, se realizará una sencilla identificación calculando la ganancia en cada flanco como el cociente entre la variación de la salida y la variación de la señal de control.

$$K1 = \frac{(8{,}8 - 5{,}85)}{(4 - 2)} = 1{,}475$$

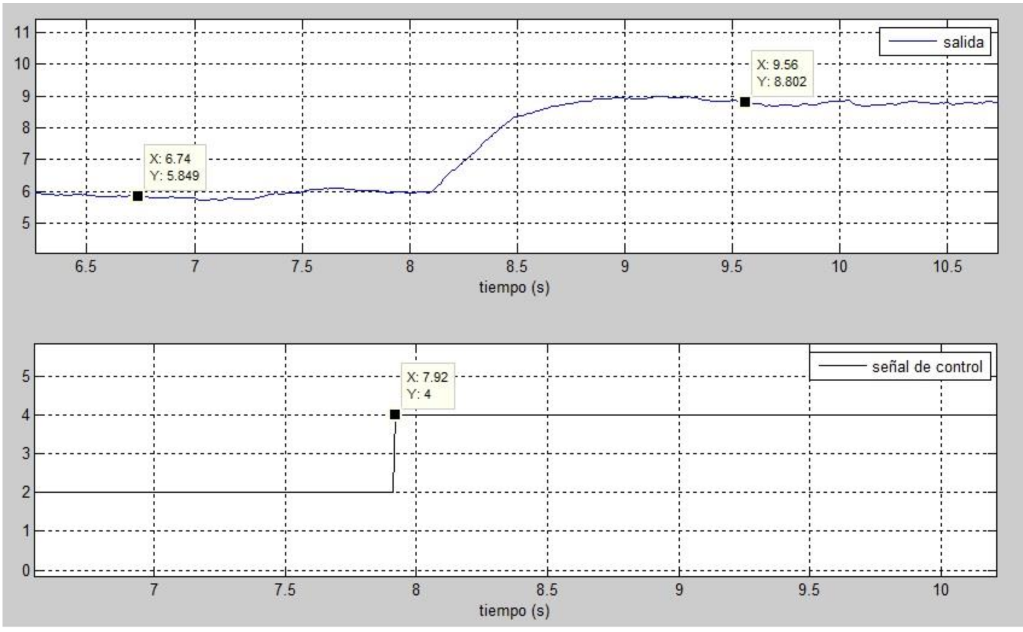


Figura 5-8. Cálculo de la ganancia para el flanco de subida.

$$K2 = \frac{(9,15 - 6,25)}{(4 - 2)} = 1,45$$

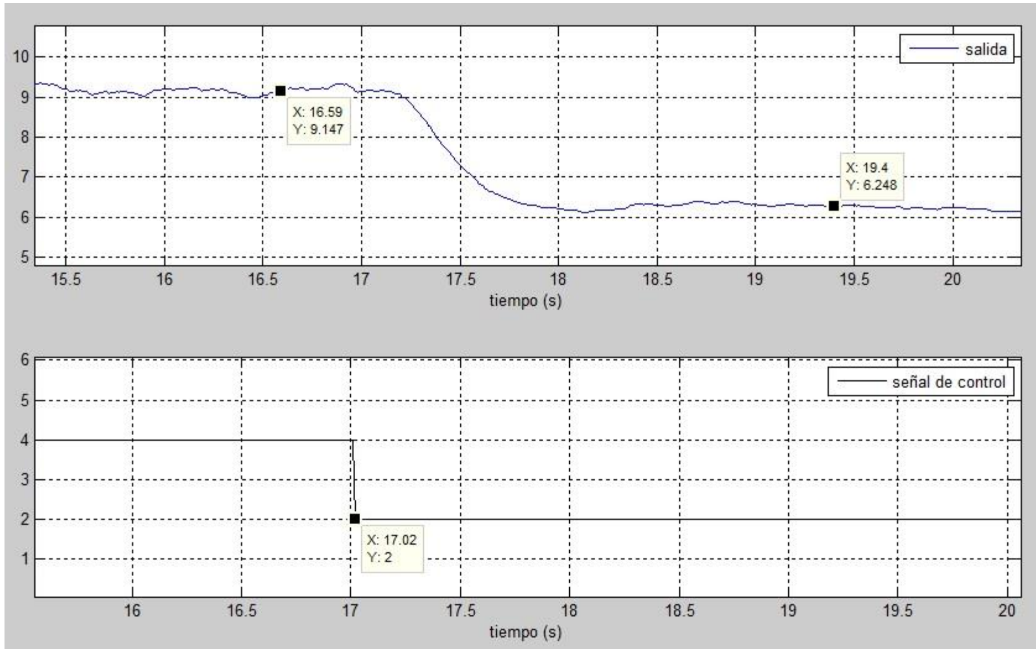


Figura 5-9. Cálculo de la ganancia para el flanco de bajada.

A continuación se calculará las constantes de tiempo, igualmente para ambos flancos, tomando la convención del valor correspondiente al tiempo de subida en el que se ha alcanzado el 63,3% del valor de la respuesta en su periodo estable, por ello en primer lugar se debe buscar los valores correspondientes a:

$$\tau 1_{63\%} = (8,8 - 5,85) * 0,633 + 5,85 = 7,7174$$

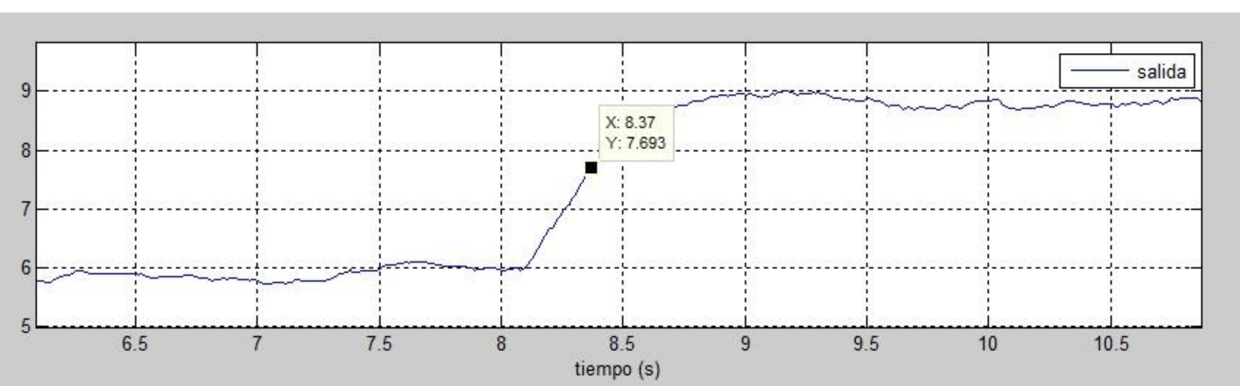


Figura 5-10. Cálculo de la constante de tiempo para el flanco de subida.

$$\tau 2_{63\%} = 9{,}15 - (9{,}15 - 6{,}25) * 0{,}633 = 7{,}3143$$

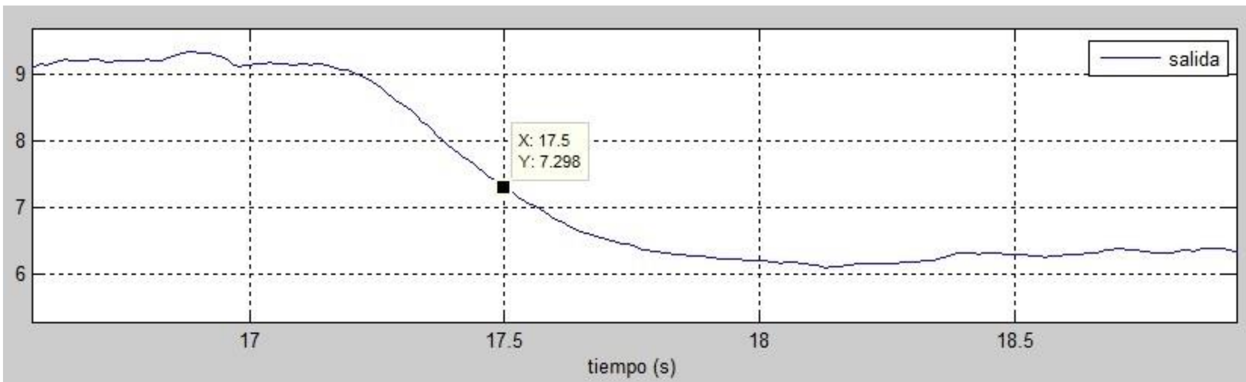


Figura 5-11. Cálculo de la constante de tiempo para el flanco de bajada.

Tomando los valores del tiempo (correspondientes a la variable "X" de las Figura *5-10* y *5-11*), y los valores de ganancia, realizando la media aritmética para cada par de valores, se tendrá un modelo en función de transferencia con el que se podrá experimentar con el sistema, no esperando el mejor resultado posible, pero obteniendo algo razonable que sirva para dictaminar la eficiencia y la utilidad del uso de unas librerías de cálculo externas, además de la facilidad para realizar manipulaciones en línea.

7.2 Planificación de los experimentos:

Con el fin de realizar de la forma más eficiente los experimentos, y conociendo la dificultad que entraña combinar varios entornos en una misma aplicación, se realizarán una variedad de experimentos pero, como lo que se está probando es el funcionamiento de un controlador predictivo, no sólo se debe realizar la conexión del mismo y comprobar que cumple sus funciones sin problemas, sino que también se realizarán pruebas de sintonización, modificando los parámetros a disposición del diseñador, como el parámetro λ en diferentes rangos de valores o la inclusión o no de restricciones al controlador, incluso capacidades ya externas al control, como la funcionalidad de la compilación del sistema en modo automático. Es por eso que en base a su complejidad, los experimentos se realizarán con el siguiente orden:

- Prueba de la señal de continua.

La prueba más sencilla y directa, que se podría realizar para la propia identificación del sistema real, será en la que éste sea conectado directamente sin pasar por el controlador externo. Este experimento no contribuye a la hora de comprobar el funcionamiento del control, pero como se ha comentado previamente (Apartado *6.4*), al disponerse una funcionalidad adicional al panel de control, es necesario probar su funcionamiento.

- Prueba del control con valores por defecto.

A continuación la prueba en orden lógico en complejidad simplemente sería activando la función del panel en la que se aplica el controlador externo, específicamente en este caso el control predictivo con los valores por defecto (sin modificar ninguno de los parámetros de diseño excepto los necesarios para la ejecución del control, modelo del proceso, horizontes de control y predicción y tiempo de muestreo).

- Pruebas de valores de λ no nulos.

Como se ha anticipado la siguiente prueba por orden lógico será modificando parámetros de diseño, como será en este caso el parámetro λ, pero no sólo se modificará una única vez para observar el resultado, sino que además se tomarán valores para este parámetro en el que se consigan respuestas del sistema con una diferencia de rendimiento relativamente notable.

7.3 Toma, exposición y análisis de resultados:

Presentado el orden que se va a tomar para los experimentos y explicado el razonamiento detrás, a continuación se va a ir mostrando cada una de las tomas de datos de cada uno de las experiencias haciendo un análisis crítico sobre el porqué o no los resultados son los esperadas y comparando, en el caso que proceda, el comportamiento entre diferentes de los experimentos realizados.

- Análisis del funcionamiento en modo automático:

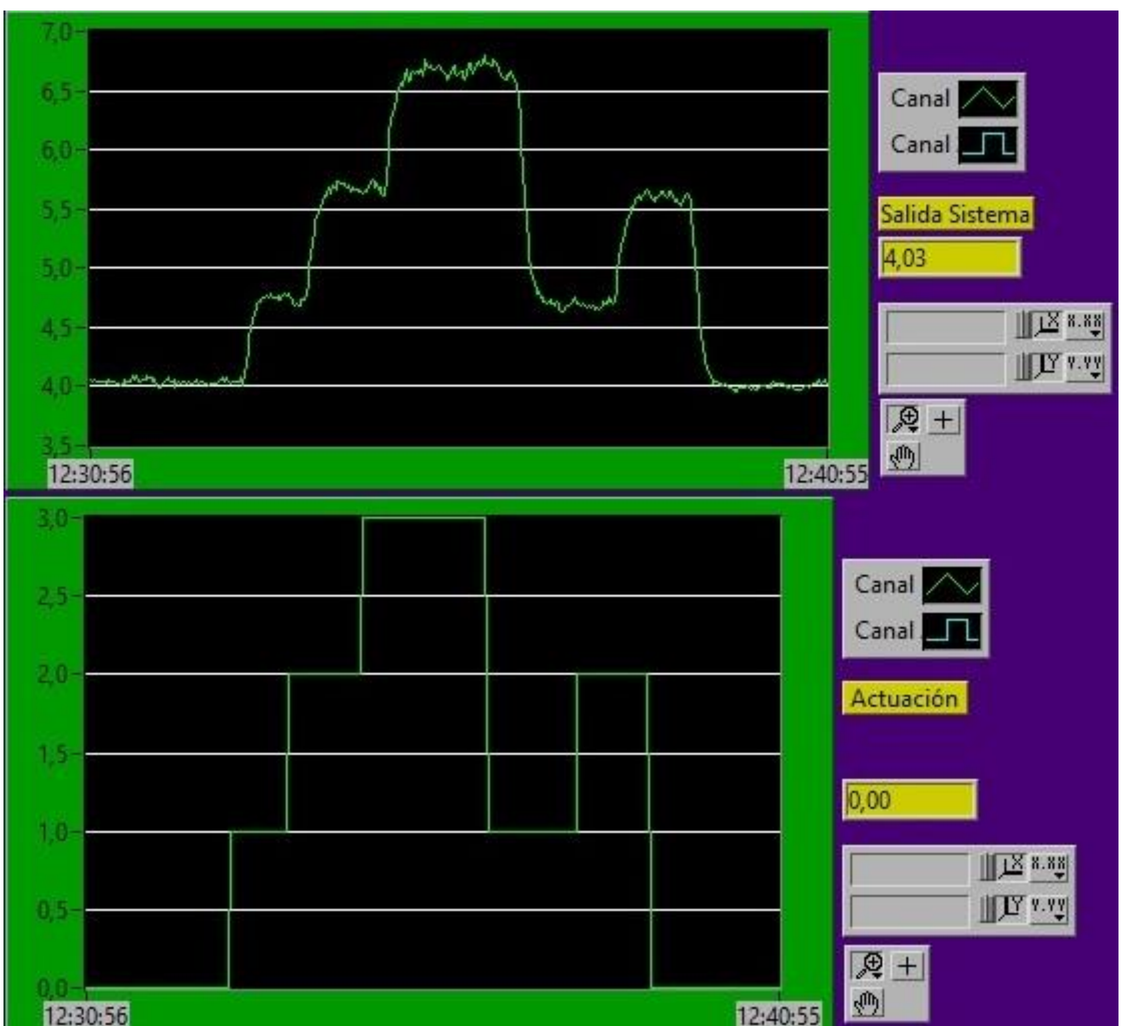


Figura 5-12. Prueba del sistema en modo automático.

De forma similar a lo que se realizó en el apartado de identificación (Apartado *7.1.2*), se puede observar un tipo de gráficas para la señal de control y salida del sistema prácticamente exactas a la representada en la Figura *5-7*. Por ello se puede comprobar que la identificación puede ser realizada a través de este modo y además, proporciona la ventaja de que un controlador externo se pueda estar ejecutando a la vez, permitiendo así realizar pruebas sin que se interfiera en la ejecución del sistema hasta que se desee.

- Análisis del funcionamiento del controlador predictivo con valores por defecto:

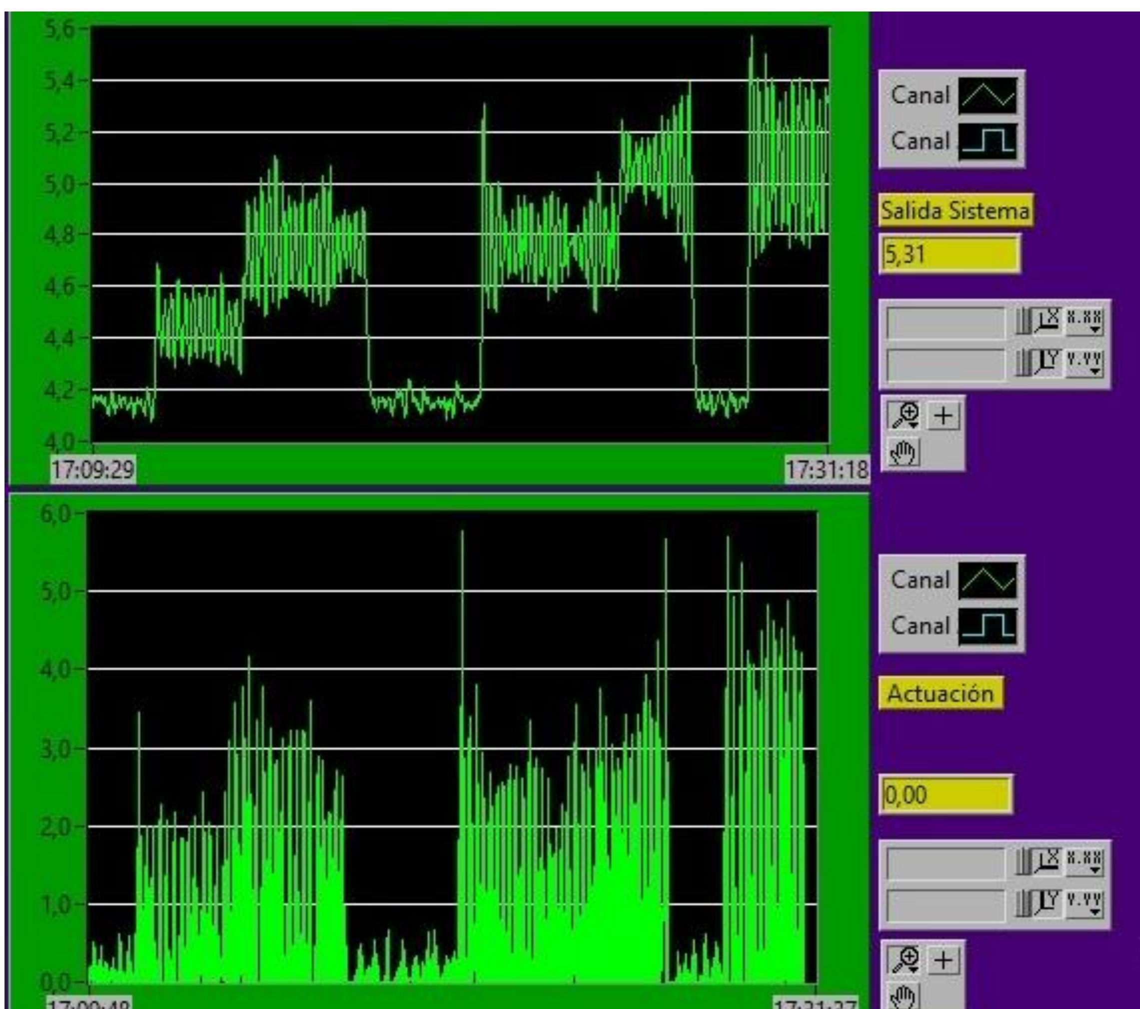


Figura 5-13. Prueba del sistema pasando por el controlador externo.

Probando el modo en el que se activa el controlador externo, en este caso el controlador predictivo, se observa como las variaciones en la señal de control son bastante inestables, teniendo mucha variación a la vez que ruido en la señal, por esto es que ningún controlador podrá funcionar con los valores por defecto, necesita ser sintonizado según el diseñador precise y según el sistema que se controle.

- Análisis modificando parámetros de sintonización del controlador:

Al margen de todos los parámetros de diseño que se tienen a disposición y que pueden modificar el comportamiento del sistema, como la precisión del modelo, aunque el parámetro principal en torno al que se van a realizar pruebas será el parámetro λ. A través de la experimentación aumentando los valores de este parámetro se puede llegar a obtener una respuesta como la que se tiene en la imagen anterior (Figura *5-14*). En las que se observa claramente la diferencia en la forma que tiene la señal de control, mucho menos ruidosa, con menos oscilación y más cerca de lo que se espera de un controlador. Es cierto que según se aumenta el valor de la referencia se pueden encontrar sobreoscilaciones mayores de esta señal de control, pero sin llegar a compararse con el experimento anterior.

Figura 5-14. Prueba del sistema modificando parámetros de sintonización.

Por último, se va a acompañar esta experimentación con la comparativa de dos valores diferentes para el parámetro λ, no muy alejados entre sí pero en los que se vea claramente una diferencia en la señal de control y se pueda hacer una comparativa de la diferencia en los resultados que se puede obtener.

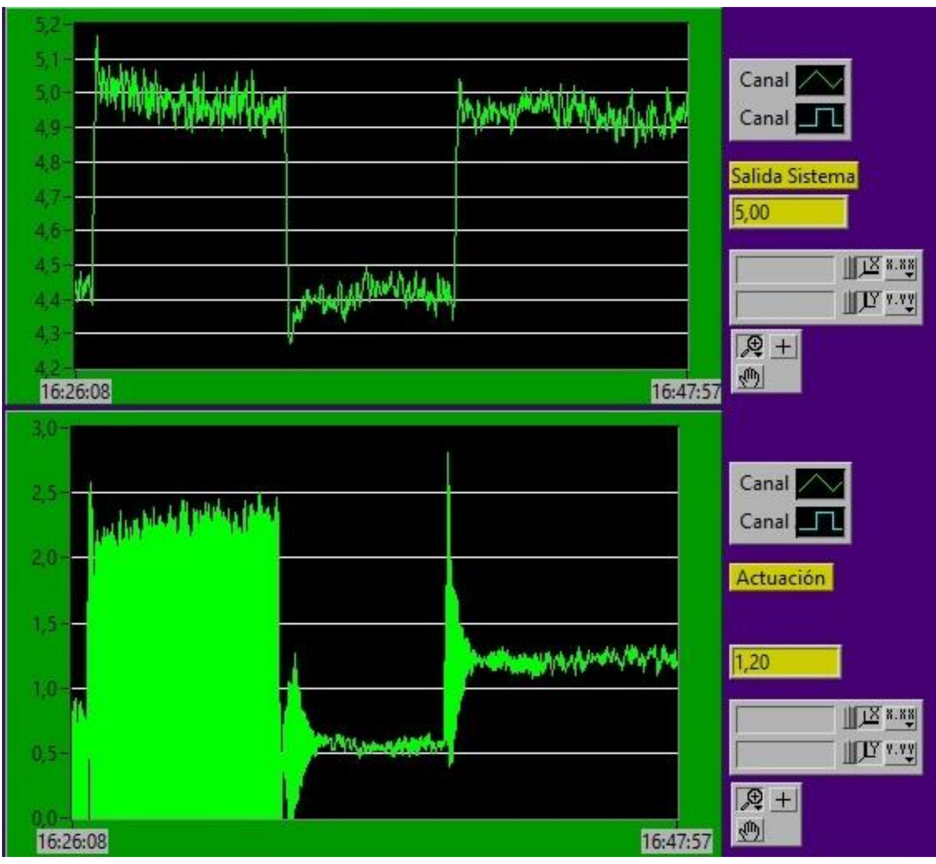


Figura 5-15. Prueba del sistema comparando diferentes valores.

Con una diferencia de tres décimas en el parámetro λ se pueden llegar a obtener señales tan dispares en la señal de control, que llevadas a la realidad de los actuadores dentro del proceso, podrían resultar en fallos en la ejecución o problemas de rendimiento, es por esto que las labores de sintonización de un controlador es el punto más escabroso y con mayor relevancia a la hora de obtener unos buenos resultados en un diseño.

7.4 Perspectivas de mejora y ampliación:

Terminada toda la exposición, desarrollo y aplicación de este trabajo, se van a abordar apartados que podrían ser mejorados o ampliados pero que por motivos de tiempo, complejidad o que eran áreas de desarrollo que distaban de los objetivos principales que se trazaron en el Apartado *1*, no se han podido desarrollar y merecen una mención especial.

- Mejoras en el modelado del proceso:

En la cuestión del tipo de modelo y de la precisión que puede añadir al funcionamiento del controlador predictivo, ha sido un tema ya abordado en varios apartados de este trabajo, y es por ello, y la importancia que tiene el modelo en este tipo de controladores, que sería el primer punto de mejora de este proyecto.

Punto que no se ha abordado en toda la profundidad que ofrece, por el hecho de que este proyecto no está centrado en el modelado si no en el proceso de optimización y el uso de librerías externas. Pero en caso de una posible mejora para el cálculo del modelo, sería muy interesante utilizar un algoritmo de mínimos cuadrados recursivos, dado la naturaleza del trabajo que se ha desarrollado previamente, para el cálculo del modelo del proceso en línea.

- Modificación de los códigos para la adición del tiempo muerto al proceso:

Por el hecho de ha arrastrado bastante complejidad el proceso de integración de varios elementos provenientes de entornos muy diferentes en un mismo proceso, que el hecho de ir. Aunque teóricamente y en cuanto a programación, el planteamiento de la adición del tiempo muerto al modelo del proceso y al controlador puede ser relativamente sencillo, como un simple desplazamiento de las matrices en el caso del último, posteriormente al disponer todo el sistema en bucle cerrado, el resultado puede estar muy lejos del esperado y entrañar mayores dificultades de las planteadas en primera instancia para solventarlo.

- Realización de un mayor número de pruebas experimentales:

Aunque para el desarrollo de la aplicación y multitud de las partes anteriores se hayan tenido que hacer infinidad de pruebas de los diferentes códigos, no ha sido suficiente para pulir aspectos de sintonización y trata de restricciones de la manera óptima. Por ello otro punto de ampliación sería la realización de una mayor realización de pruebas en estos aspectos.

Además volviendo a lo relativo del punto anterior, en caso de que se proceda a realizar estas mejoras y ampliaciones, se deberían hacer todas las pruebas necesarias añadiendo las restricciones del proceso, y por último modificando el código añadiendo el tiempo muerto junto con las restricciones y proceder a la sintonización de todo el conjunto, para así tener una perspectiva completa de todo lo que se puede desarrollar sin aumentar en exceso la complejidad del modelo (ni llegando al caso de un sistema multivariable), y tocando la mayoría de posibilidades que ofrece este tipo de control predictivo.

ANEXOS

ANEXO A: RESOLUCIÓN DE UN QP CON LAS LIBRERÍAS DE OPTIMIZACIÓN GUROBI (QPEJEMPLO)

```
#include <stdio.h>
#include <stdlib.h>
#include <gurobi_c.h>
#include <math.h>

int main() {

	int error = 0;
	int estadoptimo;

	int inc[2];
	double val[2];

	int qfil[3];
	int qcol[3];
	double qval[3];

	GRBenv *env = NULL;
	GRBmodel *model = NULL;


	//CREACION DEL ENTORNO\\

	error = GRBloadenv(&env, "QPmatlab.log");
	if (error != 0) { goto QUIT; }

	//CREACION DE MODELO (siempre vacío)\\

	error = GRBnewmodel(env, &model, "QPmatlab", 0, NULL, NULL, NULL, NULL, NULL);
	if (error != 0) { goto QUIT; }

	/*
	SIENDO x,y:

	Minimizar:         0.5x^2   +    y^2   -   x*y   -  2x   -   6y

	Sujeto a:             x     +    y         <=   2
	                                     -x     +   2y        <=   2
	                                     2x     +   y         <=   3
	                                      x                   >=   0
	                                      y                   >=   0
	*/
```

```
	//ADICION DE VARIABLES\\
	//Aqui añadimos la parte lineal de la funcion en la CREACION DE VARIABLES
	double obj[] = { -2,-6 };

	error = GRBaddvars(model, 2, 0, NULL, NULL, NULL, obj, NULL, NULL, NULL, NULL);
	if (error != 0) { goto QUIT; }

	//Ahora añadimos los términos cuadráticos a parte
	qcol[0] = 0; qfil[0] = 0; qval[0] = 0.5;
	qcol[1] = 1; qfil[1] = 1; qval[1] = 1;
	qcol[2] = 0; qfil[2] = 1; qval[2] = -1;

	error = GRBaddqpterms(model, 3, qfil, qcol, qval);
	if (error != 0) { goto QUIT; }

	//ELEGIMOS EL SENTIDO DE LA OPTIMIZACION (depues de la declaración de la funcion
objetivo)
	error = GRBsetintattr(model, GRB_INT_ATTR_MODELSENSE, GRB_MINIMIZE);
	if (error != 0) { goto QUIT; }

	//ADICION DE RESTRICCIONES\\
	//Solo tenemos restricciones lineales

	inc[0] = 0;inc[1] = 1;//Todas las restriciones hasta las dos ultimas utilizan las
dos variables

	val[0] = 1;val[1] = 1;
	error = GRBaddconstr(model, 2, inc, val, GRB_LESS_EQUAL, 2, "lc0");
	if (error != 0) { goto QUIT; }

	val[0] = -1;val[1] = 2;
	error = GRBaddconstr(model, 2, inc, val, GRB_LESS_EQUAL, 2, "lc1");
	if (error != 0) { goto QUIT; }

	val[0] = 2;val[1] = 1;
	error = GRBaddconstr(model, 2, inc, val, GRB_LESS_EQUAL, 3, "lc2");
	if (error != 0) { goto QUIT; }



	inc[0] = 0; val[0] = 1;
	error = GRBaddconstr(model, 1, inc, val, GRB_GREATER_EQUAL, 0, "lc3");
	if (error != 0) { goto QUIT; }

	inc[0] = 1; val[0] = 1;
	error = GRBaddconstr(model, 1, inc, val, GRB_GREATER_EQUAL, 0, "lc4");
	if (error != 0) { goto QUIT; }

	printf("ESCRITURA DEL MODELO COMPLETO REALIZADA\n");
	system("pause");


	//OPTIMIZACION DEL MODELO\\

	error = GRBoptimize(model);
	if (error != 0) { goto QUIT; }

	printf("\n\nOPTIMIZACION COMPLETADA\n\n");
```

```
	//ESCRITURA DEL MODELO COMPLETO\\

	error = GRBwrite(model, "QPmatlab.lp");
	if (error != 0) { goto QUIT; }



	//CAPTURA DE SOLUCION\\
	//Primero INT DEL ESTADO OPTIMO; Segundo DBL VALOR OBJETIVO; Tercero ARRAY X
(solucion)
	error = GRBgetintattr(model, GRB_INT_ATTR_STATUS, &estadoptimo);
	if (error != 0) { goto QUIT; }


	double sol[2];
	error = GRBgetdblattrarray(model, GRB_DBL_ATTR_X, 0, 2, sol);
	if (error != 0) { goto QUIT; }



	//COMPROBACION DE LA SOLUCION OPTIMA\\
	//Parte no realmente necesaria pero recomendable

	if (estadoptimo == GRB_OPTIMAL) {
		printf("   x = %.4f,   y = %.4f\n\n", sol[0], sol[1]);
	}
	else if (estadoptimo == GRB_INF_OR_UNBD) {
		printf("MODELO IRREALIZABLE O NO LIMITADO\n");
	}
	else {
		printf("LA OPTIMIZACION FINALIZÓ ANTES DE TERMINAR\n");
	}


	//LIBERACION DEL SISTEMA\\

	GRBfreemodel(model);

	GRBfreeenv(env);



	if (error != 0) {

	QUIT://COMPROBACION DE ERRORES\\

		printf("ERROR = %s\n", GRBgeterrormsg(env));
		printf("Se detecto un error y se procede a cerrar la aplicacion\n");
		system("pause");
		exit(1);

	}

	system("pause");
	return 0;
}
```

ANEXO B: RESOLUCIÓN DE LA ECUACIÓN DIOFÁNTICA EN MATLAB (DIOFANEQMATLAB)

```
%% Calculo GPC
close all;clear all;

%% En caso de poseer una función de transferencia en continuo:
%Descomentar el apartado y rellenar los siguientes dos valores


%Inicializar valores para la función de transferencia
%K =;
%tau =;

% G = tf(K,[tau 1]);
%

% Tm = 0.1;
%
% Z = c2d(G,Tm);
% znum = Z.num{1,1};
% zden = Z.den{1,1};
%

% Zp = filt(znum,zden,Tm);
% AA = Zp.den{1,1};
% BB = Zp.num{1,1};


%% Cálculo realizado conociendo los polinomios de G(z-1)

A = [1 -0.8];
B = [0.4 0.6];

%Ventana de prediccion
Nu = 3;      %Horizonte de control
N1 = 1;      %Horizonte Inicial
N2= 3;       %Horizonte Final

%Coeficientes de ponderación
lambda=0.5;

%% Cálculo de la ecuación diofántica 1 = E*AD+(z^-j)*F
delta = [1 -1];
AD = conv(A,delta);

%Los polinomios E y F pueden ser obtenidos dividiendo 1 y AD

nA = size(AD);nA = nA(2);
nB = size(B);nB = nB(2);

%% Matriz F
F = zeros(N2,nA-1);
F(1,:) = -AD(2:nA);
```

```
for i = 1:N2-1
    for j = 1:nA-2
        F(i+1,j) = F(i,j+1) - F(i,1)*AD(j+1);
    end
    F(i+1,2) = -F(i,1)*AD(nA);
end

%% Matriz E
E = zeros(N2,N2);E(:,1) = 1;
for i = 2: N2
    E(i:N2,i)=F(i-1,1);
end

%% Matriz G
Gaux=zeros(N2,N2);
for i=0:N2-1
 Gaux(i+1,1:nB+i) = conv(E(i+1,1:nB+i-1),B);
end

%Extracción de los valores no contenidos en G para la matriz Gf
Gf=zeros(N2,1);
for i=1:N2+1
    for j=1:N2+1
        if (j>i) && (j-i==1)
            Gf(i,1)=Gaux(i,j);
            Gaux(i,j)=0;
        end
    end
end
Gaux = Gaux(:,1:N2);


%Se reorganizan los términos de G
G=zeros(N2,N2);
for i=1:N2
 k=1;
 for j=i:-1:1
 G(i,k)=Gaux(i,j);
 k=k+1;
 end
end


%% Matriz f completa (respuesta libre)

f = [Gf F];

%% Matrices caso sin restricciones

H = (G'*G+lambda*eye(N2));

invH = -(inv(H));

K = invH*G';
```

ANEXO C: RESOLUCIÓN DE LA ECUACIÓN DIOFÁNTICA EN C (GPCSCRIPT)

```
#include <stdio.h>
#include <stdlib.h>
#include <gurobi_c.h>
#include <math.h>

void CalculosGyGf(double AA[], double BB[], int N2, int Nu, int na, int nB, double** G,
double** Gf);


//Funciones de convolución de vectores:
void convolucion(int NA, int NB, double A[], double B[], double *Res);


//Funciones de reserva de espacio en memoria para matrices y vectores
double** reservarMAT(int filas, int columnas);
double* reservarVEC(int filas);


void printMAT(double **mat, int filas, int columnas);
void printVEC(double *vec, int filas);

int main() {

	//Horizonte final y de control
	int N2 = 3; int Nu = 3;

	//Valores del modelo del proceso en forma de función de transferencia
	double A[] = { 1, -0.8 }; double B[] = { 0.4, 0.6 };
	int nA = 2; int nB = 2;

	//Inicialización de las matrices resultado
	double **G;
	G = reservarMAT(N2, Nu);

	double **Gf;
	Gf = reservarMAT(N2, (nA + nB - 1));

	//Llamada a la función de cálculo
	CalculosGyGf(A, B, N2, Nu, nA, nB, G, Gf);

	printMAT(G, N2, Nu);

	printMAT(Gf, N2, (nA + nB - 1));


	system("pause");
	return 0;
}


void CalculosGyGf(double AA[], double BB[], int N2, int Nu, int na, int nB, double** G,
double** Gf) {

	int i, j, k;
	int nA = na + 1;
```

```
	double delta[] = { 1, -1 };

	double *AD;
	AD = reservarVEC(nA);

	convolucion(na, 2, AA, delta, AD);


	//CALCULO DE LA MATRIZ F

	double **F;
	F = reservarMAT(N2, nA - 1);

	//Calculo del valor de la primera fila de F (obtenido de dividir AD entre 1)

	for (i = 1;i < nA;i++) {
		F[0][i - 1] = -AD[i];
	}


	for (i = 0;i < N2 - 1;i++) {
		for (j = 0;j <= nA - 2;j++) {
			F[i + 1][j] = F[i][j + 1] - F[i][j] * AD[j + 1];

		}
		F[i + 1][1] = -F[i][0] * AD[nA - 1];
	}


	//CALCULO DE LA MATRIZ E

	double **E;
	E = reservarMAT(N2, Nu);

	//Se inicializa el primer valor del vector que coincide con toda la primera
columna
	for (i = 0;i < N2;i++) {
		E[i][0] = 1;
	}

	k = 0;
	for (i = 1;i < N2;i++) {
		for (j = 1 + k;j < Nu;j++) {
			E[j][i] = F[k][0];
		}
		k += 1;
	}


	//CALCULO DE LA MATRIZ G

	//Se reserva espacio
	double *convG;//Vector que contendrá la convolucion entre E y B
	convG = reservarVEC(Nu + nB - 1);

	double *Eaux;//En este vector se guardarán la última componente de E
	Eaux = reservarVEC(Nu);

	//Extraccion de la componente con todos los valores que tendrá E en todo el
horizonte
	for (i = 0;i < Nu;i++) {
		Eaux[i] = E[Nu - 1][i];
	}
```

```
        //Calculo de la convolucion de E y B
        convolucion(Nu, nB, Eaux, BB, convG);

        //Se recolocan los términos de la convolución en las diagonales para obtener G
        for (i = 0;i < N2;i++) {
                for (j = 0;j < Nu;j++) {
                        if (i - j >= 0) {
                                G[i][j] = convG[i - j];
                        }
                }
        }


        //CALCULO COMPLETO DE Gf

        double *aux;//En este vector guardaremos el valor de la convolución sucesivamente
        aux = reservarVEC(Nu + (nB - 1));

        double **Gfaux;
        Gfaux = reservarMAT(N2, (nB - 1));

        for (i = 0;i < N2;i++) {
                for (j = 0;j < Nu;j++) {
                        //Se extrae en cada bucle la componente de E correspondiente
                        Eaux[j] = E[i][j];
                }
                convolucion(i + nB, nB, Eaux, BB, aux);

                for (j = 0;j < (nB - 1);j++) {
                        Gfaux[i][j] = aux[i + 1];
                }
        }

        //Se colocan en la matriz definitiva para Gf primero Gfaux y luego F

        for (i = 0;i < N2;i++) {
                for (j = 0;j < (nB - 1);j++) {
                        Gf[i][j] = Gfaux[i][j];
                }

        }

        //Se completa el vector con la respuesta libre entera
        for (i = 0;i < N2;i++) {
                for (j = nB - 1, k = 0;j < (nA + nB - 2), k < (nA - 1);j++, k++) {
                        Gf[i][j] = F[i][k];
                }
        }

}
```

```
//FUNCIÓN DE CONVOLUCIÓN DE VECTORES

void convolucion(int NA, int NB, double A[], double B[], double *Res) {

	int Naux = NA + NB - 1;
	int i, j, k;

	double **Maux;
	Maux = reservarMAT(NB + 1, Naux);


	k = 0;
	for (i = 0;i < NB;i++) {
		for (j = 0;j < NA;j++) {
			Maux[i][j + k] = B[i] * A[j];
		}
		k += 1;
	}


	for (i = 0;i < NB;i++) {
		for (j = 0;j < Naux;j++) {
			Maux[i + 1][j] = Maux[i + 1][j] + Maux[i][j];
		}
	}


	for (i = 0;i < Naux;i++) {
		Res[i] = Maux[NB][i];
	}

}


//FUNCIONES DE RESERVA DE ESPACIO EN MEMORIA PARA MATRICES Y VECTORES RESPECTIVAMENTE

double** reservarMAT(int filas, int columnas) {

	double **mat;
	int i;

	mat = (double**)calloc(filas, sizeof(double*));
	if (mat == NULL) {
		printf("No se ha podido reservar memoria.");
		exit(1);
	}
	for (i = 0; i < filas; i++) {
		mat[i] = (double*)calloc(columnas, sizeof(double));
		if (mat[i] == NULL) {
			printf("No se ha podido reservar memoria.");
			exit(1);
		}
	}

	return mat;
}
```

```c
double* reservarVEC(int filas) {

	double *vec;

	vec = (double*)calloc(filas, sizeof(double));
	if (vec == NULL) {
		printf("No se ha podido reservar memoria.");
		exit(1);
	}


	return vec;
}



//Plot de matrices y vectores

void printMAT(double **mat, int filas, int columnas) {

	int i, j;

	for (i = 0;i < filas;i++) {
		for (j = 0, printf("\n");j < columnas;j++) {
			printf("%f\t", mat[i][j]);
		}
	}printf("\n\n");

}

void printVEC(double *vec, int filas) {

	int i;

	for (i = 0;i < filas;i++) {
		printf("%f\n", vec[i]);
	}printf("\n\n");

}
```

ANEXO D: IMPLEMENTACIÓN EN LABVIEW DE LIBRERÍAS PARA EL CÁLCULO MATRICIAL DE UN GPC (GPCDLL)

```
// GPCdll.cpp : Define las funciones exportadas de la aplicación DLL.
//

#include "stdafx.h"

#include "stdafx.h"
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include "GPCdll.h"


//Funciones para el cálculo preparatorio de las matrices del control predictivo
void WINAPI calcG(double *AA, double *BB, int N2, int Nu, int na, int nB, double** G);
void WINAPI calcGF(double *AA, double *BB, int N2, int Nu, int na, int nB, double** Gf);
void WINAPI CalculosGyGf(double *AA, double *BB, int N2, int Nu, int na, int nB, double**
GG, double** Gf);

//Función de cálculo de convolución entre vectores
void WINAPI convolucion(int NA, int NB, double A[], double B[], double *Res);

//Funciones de reserva de espacio en memoria para matrices y vectores
double** WINAPI reservarMAT(int filas, int columnas);
double* WINAPI reservarVEC(int filas);

//Función de linealización por filas de una matriz:
void WINAPI LinMat(double **M, int N1, int N2, double *Res);


void WINAPI GPCunico(double* A, double* B, int nA, int nB, int N2, int Nu, int NG, int
NGf, double* Glin, double* Gflin) {

	double **G;
	G = reservarMAT(N2, Nu);

	double **Gf;
	Gf = reservarMAT(N2, (nA + nB - 1));

	CalculosGyGf(A, B, N2, Nu, nA, nB, G, Gf);

	//Linealización de G y Gf por filas para poder enviarse a traves de una función de
labview
	LinMat(G, N2, Nu, Glin);
	LinMat(Gf, N2, (nA + nB - 1), Gflin);

}

void WINAPI DevuelveG(double* A, double *B, int nA, int nB, int N2, int Nu, int NG,
double* Glin) {
```

```
	//Reserva de memoria y posterior calculo de la matriz G
	double **G;
	G = reservarMAT(N2, Nu);

	calcG(A, B, N2, Nu, nA, nB, G);

	//Linealización de G por filas para poder ser enviada a través de la dll en
labview como Vector
	LinMat(G, N2, Nu, Glin);

}


void WINAPI DevuelveGf(double* A, double *B, int nA, int nB, int N2, int Nu, int NGf,
double* Gflin) {

	double **Gf;
	Gf = reservarMAT(N2, (nA + nB - 1));

	calcGF(A, B, N2, Nu, nA, nB, Gf);

	//Linealización de Gf por filas
	LinMat(Gf, N2, (nA + nB - 1), Gflin);

}



//FUNCIONES PARA EL CALCULO DE LAS MATRICES NECESARIAS PARA EL GPC

//Fúnción única del cálculo del GPC
void WINAPI CalculosGyGf(double *AA, double *BB, int N2, int Nu, int na, int nB, double**
G, double** Gf) {

	int i, j, k;
	int nA = na + 1;

	double delta[] = { 1, -1 };

	double *AD;
	AD = reservarVEC(nA);

	convolucion(na, 2, AA, delta, AD);


	//CALCULO DE LA MATRIZ F

	double **F;
	F = reservarMAT(N2, nA - 1);

	//Calculo del valor de la primera fila de F (obtenido de dividir AD entre 1)

	for (i = 1;i < nA;i++) {
		F[0][i - 1] = -AD[i];
	}
```

```
	for (i = 0;i < N2 - 1;i++) {
		for (j = 0;j <= nA - 2;j++) {
			F[i + 1][j] = F[i][j + 1] - F[i][j] * AD[j + 1];

		}
		F[i + 1][1] = -F[i][0] * AD[nA - 1];
	}



	//CALCULO DE LA MATRIZ E

	double **E;
	E = reservarMAT(N2, Nu);

	//Se inicializa el primer valor del vector que coincide con toda la primera
columna
	for (i = 0;i < N2;i++) {
		E[i][0] = 1;
	}

	k = 0;
	for (i = 1;i < N2;i++) {
		for (j = 1 + k;j < Nu;j++) {
			E[j][i] = F[k][0];
		}
		k += 1;
	}


	//CALCULO DE LA MATRIZ G

	//Se reserva espacio
	double *convG;//Vector que contendrá la convolucion entre E y B
	convG = reservarVEC(Nu + nB - 1);

	double *Eaux;//En este vector se guardarán la última componente de E
	Eaux = reservarVEC(Nu);

	//Extraccion de la componente con todos los valores que tendrá E en todo el
horizonte
	for (i = 0;i < Nu;i++) {
		Eaux[i] = E[Nu - 1][i];
	}


	//Calculo de la convolucion de E y B
	convolucion(Nu, nB, Eaux, BB, convG);

	//Se recolocan los términos de la convolución en las diagonales para obtener G
	for (i = 0;i < N2;i++) {
		for (j = 0;j < Nu;j++) {
			if (i - j >= 0) {
				G[i][j] = convG[i - j];
			}
		}
	}
```

```
	//CALCULO COMPLETO DE Gf

	double *aux;//En este vector guardaremos el valor de la convolución sucesivamente
	aux = reservarVEC(Nu + (nB - 1));

	double **Gfaux;
	Gfaux = reservarMAT(N2, (nB - 1));

	for (i = 0;i < N2;i++) {
		for (j = 0;j < Nu;j++) {
			//Se extrae en cada bucle la componente de E correspondiente
			Eaux[j] = E[i][j];
		}
		convolucion(i + nB, nB, Eaux, BB, aux);

		for (j = 0;j < (nB - 1);j++) {
			Gfaux[i][j] = aux[i + 1];
		}
	}

	//Se colocan en la matriz definitiva para Gf primero Gfaux y luego F

	for (i = 0;i < N2;i++) {
		for (j = 0;j < (nB - 1);j++) {
			Gf[i][j] = Gfaux[i][j];
		}

	}

	//Se completa el vector con la respuesta libre entera
	for (i = 0;i < N2;i++) {
		for (j = nB - 1, k = 0;j < (nA + nB - 2), k < (nA - 1);j++, k++) {
			Gf[i][j] = F[i][k];
		}
	}

}


//Función desdoblada del cálculo del GPC
void WINAPI calcG(double *AA, double *BB, int N2, int Nu, int na, int nB, double** GG) {

	int i, j, k;

	int nA = na + 1;

	double delta[] = { 1, -1 };

	double *AD;
	AD = reservarVEC(nA);

	convolucion(na, 2, AA, delta, AD);


	//CALCULO DE LA MATRIZ F

	double **FF;
	FF = reservarMAT(N2, nA - 1);

	//Calculo del valor de la primera fila de F (obtenido de dividir AD entre 1)

	for (i = 1;i < nA;i++) {
		FF[0][i - 1] = -AD[i];
	}
```

```
	for (i = 0;i < N2 - 1;i++) {
		for (j = 0;j <= nA - 2;j++) {
			FF[i + 1][j] = FF[i][j + 1] - FF[i][j] * AD[j + 1];

		}
		FF[i + 1][1] = -FF[i][0] * AD[nA - 1];
	}



	//CALCULO DE LA MATRIZ E

	double **EE;
	EE = reservarMAT(N2, Nu);

	//Inicialización del primer valor del vector que coincide con toda la primera
columna
	for (i = 0;i < N2;i++) {
		EE[i][0] = 1;
	}

	k = 0;
	for (i = 1;i < N2;i++) {
		for (j = 1 + k;j < Nu;j++) {
			EE[j][i] = FF[k][0];
		}
		k += 1;
	}


	//CALCULO DE LA MATRIZ G
	//Reserva espacio para la variable que contendrá la convolucion para calcular G

	double *convG;
	convG = reservarVEC(Nu + nB - 1);
	//Reserva del vector completo con los valores de E
	double *Eaux;
	Eaux = reservarVEC(Nu);

	//Extraccion de la componente con todos los valores que tendrá E en todo el
horizonte
	for (i = 0;i < Nu;i++) {
		Eaux[i] = EE[Nu - 1][i];
	}


	//Calculo de la convolucion de E y B
	convolucion(Nu, nB, Eaux, BB, convG);

	//Y por ultimo se colocan los términos de la convolución en las diagonales para
obtener G

	for (i = 0;i < N2;i++) {
		for (j = 0;j < Nu;j++) {
			if (i - j >= 0) {
				GG[i][j] = convG[i - j];
			}
		}
	}
}
```

```
void WINAPI calcGF(double *AA, double *BB, int N2, int Nu, int na, int nB, double** Gf) {

	int i, j, k;

	int nA = na + 1;

	double delta[] = { 1, -1 };

	double *AD;
	AD = reservarVEC(nA);

	convolucion(na, 2, AA, delta, AD);


	//CALCULO DE LA MATRIZ F

	double **FF;
	FF = reservarMAT(N2, nA - 1);

	//Calculo del valor de la primera fila de F (obtenido de dividir AD entre 1)

	for (i = 1;i < nA;i++) {
		FF[0][i - 1] = -AD[i];
	}


	for (i = 0;i < N2 - 1;i++) {
		for (j = 0;j <= nA - 2;j++) {
			FF[i + 1][j] = FF[i][j + 1] - FF[i][j] * AD[j + 1];

		}
		FF[i + 1][1] = -FF[i][0] * AD[nA - 1];
	}



	//CALCULO DE LA MATRIZ E

	double **EE;
	EE = reservarMAT(N2, Nu);

	//Se inicializa el primer valor del vector que coincide con toda la primera
columna
	for (i = 0;i < N2;i++) {
		EE[i][0] = 1;
	}

	k = 0;
	for (i = 1;i < N2;i++) {
		for (j = 1 + k;j < Nu;j++) {
			EE[j][i] = FF[k][0];
		}
		k += 1;
	}

	//CALCULO COMPLETO DE LA MATRIZ Gf

	double *Eaux;//En este vector se guardan las filas sucesivas de E
	Eaux = reservarVEC(Nu);
	double *aux;//En este vector se guarda el valor de la convolución sucesivamente
	aux = reservarVEC(Nu + (nB - 1));

	double **Gfaux;
	Gfaux = reservarMAT(N2, (nB - 1));
```

```
	for (i = 0;i < N2;i++) {
		for (j = 0;j < Nu;j++) {
			//Se extrae en cada bucle la componente de E correspondiente
			Eaux[j] = EE[i][j];
		}
		convolucion(i + nB, nB, Eaux, BB, aux);

		for (j = 0;j < (nB - 1);j++) {
			Gfaux[i][j] = aux[i + 1];
		}
	}

	//Se colocan en la matriz definitiva para Gf primero Gfaux y luego F

	for (i = 0;i < N2;i++) {
		for (j = 0;j < (nB - 1);j++) {
			Gf[i][j] = Gfaux[i][j];
		}

	}

	for (i = 0;i < N2;i++) {
		for (j = nB - 1, k = 0;j < (nA + nB - 2), k < (nA - 1);j++, k++) {
			Gf[i][j] = FF[i][k];
		}
	}
}



//Función para la convolución de vectores
void WINAPI convolucion(int NA, int NB, double A[], double B[], double *Res) {

	int Naux = NA + NB - 1;
	int i, j, k;

	double **Maux;
	Maux = reservarMAT(NB + 1, Naux);


	k = 0;
	for (i = 0;i < NB;i++) {
		for (j = 0;j < NA;j++) {
			Maux[i][j + k] = B[i] * A[j];
		}
		k += 1;
	}


	for (i = 0;i < NB;i++) {
		for (j = 0;j < Naux;j++) {
			Maux[i + 1][j] = Maux[i + 1][j] + Maux[i][j];
		}
	}


	for (i = 0;i < Naux;i++) {
		Res[i] = Maux[NB][i];
	}

}
```

```c
//FUNCIONES DE RESERVA DE ESPACIO EN MEMORIA PARA MATRICES Y VECTORES RESPECTIVAMENTE

double** WINAPI reservarMAT(int filas, int columnas) {

	double **mat;
	int i;

	mat = (double**)calloc(filas, sizeof(double*));
	if (mat == NULL) {
		printf("No se ha podido reservar memoria.");
		exit(1);
	}
	for (i = 0; i < filas; i++) {
		mat[i] = (double*)calloc(columnas, sizeof(double));
		if (mat[i] == NULL) {
			printf("No se ha podido reservar memoria.");
			exit(1);
		}
	}

	return mat;
}

double* WINAPI reservarVEC(int filas) {

	double *vec;

	vec = (double*)calloc(filas, sizeof(double));
	if (vec == NULL) {
		printf("No se ha podido reservar memoria.");
		exit(1);
	}


	return vec;
}



//FUNCION PARA LA LINEALIZACIÓN POR FILAS DE UNA MATRIZ

void WINAPI LinMat(double **M, int N1, int N2, double *Res) {

	int i, j;
	int k = 0;
	for (i = 0;i < N1;i++) {
		for (j = 0;j < N2;j++) {
			Res[k] = M[i][j];
			k += 1;
		}
	}

}
```

ANEXO E: SCRIPT DE RESOLUCIÓN DE UN ÚNICO BLUCLE DE CONTROL (GPCUNCICLO)

```
#include <stdio.h>
#include <stdlib.h>
#include <gurobi_c.h>
#include <math.h>


//Funciones para el cálculo de las matrices necesarias invariantes en el control
predictivo
void calcG(double *AA, double *BB, int N2, int Nu, int na, int nB, double** GG);
void calcGF(double *AA, double *BB, int N2, int Nu, int na, int nB, double** Gf);


//Funciones para las operaciones necesarias con vectores y matrices:
void convolucion(int NA, int NB, double A[], double B[], double *Res);
void traspMAT(double **M, int N1, int N2, double **Res);
void multiMAT(double **M1, double **M2, int N1, int N2, int N3, double **Res);
void multiVEC(double *V1, double *V2, int N, double Res);
void MATxVEC(double **M, int N1, int N2, double *V, double *Res);
void VECxMAT(double **M, int N1, int N2, double *V, double *Res);



//Funciones de reserva de espacio en memoria para matrices y vectores
double** reservarMAT(int filas, int columnas);
double* reservarVEC(int filas);


void printMAT(double **mat, int filas, int columnas);
void printVEC(double *vec, int filas);

void Erroracion(GRBenv *env);


int main() {

	int i, j, k;

	//PARAMETROS MODIFICABLES DEL BUCLE

	int N2 = 3; int Nu = 3;

	double Y = 0;
	double lambda = 0.5;

	//Vector de referencias futuras

	double *w;
	w = reservarVEC(N2);//Vamos a aplicar la convención de que las referencias futuras
serán constantes en cada bucle

	for (i = 0;i < N2;i++) {
		w[i] = 2;
	}
```

```
	//Parámetros constantes en todos los bucles

	//double A[] = {1, -0.8074}; double B[] = {0, 0.2816};
	double A[] = { 1, -0.8 }; double B[] = { 0.4, 0.6 };


	int nA = 2; int nB = 2;

	//Reserva de memoria y posterior calculo de la matriz G
	double **G;
	G = reservarMAT(N2, Nu);

	calcG(A, B, N2, Nu, nA, nB, G);
	printMAT(G, N2, Nu);


	//Debemos saber las dimensiones de esta matriz de antemano, y serán: (N2) filas y
(nB-1)+(nA-1)
	double **Gf;
	Gf = reservarMAT(N2, (nA + nB - 1));

	calcGF(A, B, N2, Nu, nA, nB, Gf);
	printMAT(Gf, N2, (nA + nB - 1));



	/*
	INIALIZACIÓN DE VARIABLES DE CONTROL NECESARIAS
	*/

	static double uk;//Incremento de la señal de control
	double Usal;//Señal de salida

	static double *ukpast;//Vector de las salidas pasadas necesarios (se inicializa en
el primer bucle en cero)
	ukpast = reservarVEC(nB - 1);

	static double *ypast;//Vector con las entradas pasadas necesarias
	ypast = reservarVEC(nA);//si estamos en el primer bucle, el segundo valor será
cero,hay que inicializarlo igualandolo al primero
	ypast[0] = Y;
	for (i = 1;i < nA;i++) {
		if (ypast[i] == 0) {
			ypast[i] = Y;
		}
	}



	//En primer lugar vamos a calcular el valor del término "(f-w)" que está presente
en los términos LINEAL e INDEPENDIENTE
	//La matriz Gf se divide por columnas en primer lugar, (nB - 1) columnas
multiplicadas por un término del vector ukpast
	for (i = 0;i < (nB - 1);i++) {
		for (j = 0;j < N2;j++) {
			Gf[j][i] = Gf[j][i] * ukpast[i];
		}
	}
```

```
	//Las demás columnas (nA) deben ser multiplicadas por las entradas pasadas, en primera instancia éstas serán siempre el valor Y

	for (i = (nB - 1);i < (nA + nB - 1);i++) {
		for (j = 0;j < N2;j++) {
			Gf[j][i] = Gf[j][i] * ypast[i];
		}
	}

	//Por último para el cálculo del término que dijimos, debemos sumar entre sí todos los términos de las filas para restar W
	double *Gfaux;
	Gfaux = reservarVEC(N2);

	double aux;
	for (i = 0;i < N2;i++) {
		aux = 0;
		for (j = 0;j < (nA + nB - 1);j++) {
			aux += Gf[i][j];
		}
		Gfaux[i] = aux - w[i];
	}


	/*
	CALCULO DEL TÉRMINO INDEPENDIENTE
	*/

	double f0 = 0;
	multiVEC(Gfaux, Gfaux, N2, f0);


	/*
	CALCULO DE LOS TÉRMINOS LINEALES
	*/

	//Primero multiplicamos Gfaux por 2 y luego lo multiplicaremos por G, el vector resultado contendrá los valores que acompañan los términos lineales en orden creciente

	for (i = 0;i < N2;i++) {
		Gfaux[i] = Gfaux[i] * 2;
	}

	double *linval;
	linval = reservarVEC(Nu);

	VECxMAT(G, N2, Nu, Gfaux, linval);

	/*
	CALCULO DE LOS TÉRMINOS CUADRÁTICOS
	*/

	//En primer lugar realizaremos el cálculo matricial de H
	//Creamos y construimos G traspuesta
	double **GT;
	GT = reservarMAT(Nu, N2);

	traspMAT(G, N2, Nu, GT);

	//Creamos la matriz H y en ella guardamos el resultado de la multiplicación de G trapuesta por G
	double **H;
	H = reservarMAT(Nu, Nu);
	multiMAT(GT, G, Nu, N2, Nu, H);printMAT(H, Nu, Nu);
```

```
	//Como la matriz identidad multiplicada por lambda va a ser una matriz con ÚNICAMENTE LAMBDAS EN LA DIAGONAL, los sumamos directamente

	for (i = 0, j = 0;i < Nu, j < Nu;i++, j++) {
		H[i][j] += lambda;
	}
	//Con esto ya tendríamos calculada H
	printMAT(H, Nu, Nu);
	system("pause");

	for (i = 0;i < Nu;i++) {
		for (j = 0;j < Nu;j++) {
			if (i < j) {
				H[i][j] += H[j][i];
			}
		}
	}
	printMAT(H, Nu, Nu);


	//Calculo del número de variables cuadráticas
	int Nqvar;
	Nqvar = ((Nu*Nu - Nu) / 2) + Nu;

	//Calculo genérico de la combinación de los índices de las variables cuadráticas en vectores
	//Deben estar en formato "int" para que pueda ser utilizable a posteriori por la función de Gurobi

	int *qfil; int *qcol;

	qfil = (int*)calloc(Nqvar, sizeof(int));
	if (qfil == NULL) {
		printf("No se ha podido reservar memoria.");
		exit(1);
	}

	qcol = (int*)calloc(Nqvar, sizeof(int));
	if (qcol == NULL) {
		printf("No se ha podido reservar memoria.");
		exit(1);
	}

	k = 0;
	for (i = 0;i < Nu;i++) {
		for (j = i;j < Nu;j++) {
			qfil[k] = i;
			qcol[k] = j;
			k++;
		}
	}
```

```
	double *qval;//Inicializamos el vector que va a contener los valores de todas las 
variables cuadráticas en orden
	qval = reservarVEC(Nqvar);

	k = 0;
	for (i = 0;i < Nu;i++) {
		for (j = 0;j < Nu;j++) {
			if (i <= j) {
				qval[k] = H[i][j];
				k++;
			}
		}
	}
	printVEC(qval, Nqvar);
	system("pause");

	/*
	PROCESO DE OPTIMIZACIÓN DE LA FUNCIÓN OBJETIVO
	Y UTILIZACIÓN DE LAS LIBRERÍAS GUROBI
	*/


	GRBenv *env = NULL;
	GRBmodel *model = NULL;

	int error = 0;

	double *Solucion;
	Solucion = reservarVEC(Nu);

	//CREACION DEL ENTORNO\\

	error = GRBloadenv(&env, "QP1ciclo.log");
	if (error != 0) { Erroracion(env); }

	//CREACION DEL MODELO VACIO\\

	error = GRBnewmodel(env, &model, "QP1ciclo", 0, NULL, NULL, NULL, NULL/*Las 
variables siempre serán continuas por defecto*/, NULL);
	if (error != 0) { Erroracion(env); }

	//ADICIÓN DE VARIABLES LINEALES\\
	//Tendremos tantas variables lineales iguales a Nu

	error = GRBaddvars(model, Nu, 0, NULL, NULL, NULL, linval, NULL, NULL, NULL, 
NULL);
	if (error != 0) { Erroracion(env); }

	//ADICIÓN DE VARIABLES CUADRÁTRICAS\\

	error = GRBaddqpterms(model, Nqvar, qfil, qcol, qval);
	if (error != 0) { Erroracion(env); }

	//ADICIÓN DEL TÉRMINO INDEPENDIENTE\\

	error = GRBsetdblattr(model, GRB_DBL_ATTR_OBJCON, f0);
	if (error != 0) { Erroracion(env); }


	//ELECCIÓN DEL SENTIDO DE OPTIMIZACIÓN (para control predictivo siempre elegiremos 
minimización)\\

	error = GRBsetintattr(model, GRB_INT_ATTR_MODELSENSE, GRB_MINIMIZE);
	if (error != 0) { Erroracion(env); }
```

```
	//ESCRITURA Y OPTIMIZACION DEL MODELO\\

	error = GRBwrite(model, "GCPX1.lp");
	if (error != 0) { Erroracion(env); }

	error = GRBoptimize(model);
	if (error != 0) { Erroracion(env); }


	//CAPTURA DE SOLUCION\\

	error = GRBgetdblattrarray(model, GRB_DBL_ATTR_X, 0, Nu, Solucion);
	if (error != 0) { Erroracion(env); }

	//Liberación del modelo y el entorno en el que éste está (en ese orden específico)
	GRBfreemodel(model);
	GRBfreeenv(env);

	/*
		ACTUALIZACION DE VALORES DEL BUCLE DE CONTROL
	*/
	printf("Vector U:  ");
	for (i = 0;i < Nu;i++) {
		printf("%f\t", Solucion[i]);
	}
	printf("\n\n");
	uk = Solucion[0];

	//Actualizamos el valor del vector de señales de control pasadas
	if ((nB - 1) > 1) {
		for (i = (nB - 1);i > 0; i--) {
			ukpast[i] = ukpast[i - 1];
		}
		ukpast[0] = uk;
	}
	else {
		ukpast[0] = uk;
	}

	//Actualizamos el valor de la variable de salida
	Usal = uk;

	printf("Usalida : %f\n\n", Usal);


	system("pause");
	return 0;
}
//FUNCIONES PARA EL CALCULO DE LAS MATRICES NECESARIAS PARA EL GPC

void calcG(double *AA, double *BB, int N2, int Nu, int na, int nB, double** GG) {

	int i, j, k;

	int nA = na + 1;

	double delta[] = { 1, -1 };

	double *AD;
	AD = reservarVEC(nA);

	convolucion(na, 2, AA, delta, AD);
```

```
        //CALCULO DE LA MATRIZ F

        double **FF;
        FF = reservarMAT(N2, nA - 1);

        //Calculo del valor de la primera fila de F (obtenido de dividir AD entre 1)

        for (i = 1;i < nA;i++) {
                FF[0][i - 1] = -AD[i];
        }


        for (i = 0;i < N2 - 1;i++) {
                for (j = 0;j <= nA - 2;j++) {
                        FF[i + 1][j] = FF[i][j + 1] - FF[i][j] * AD[j + 1];

                }
                FF[i + 1][1] = -FF[i][0] * AD[nA - 1];
        }



        //CALCULO DE LA MATRIZ E

        double **EE;
        EE = reservarMAT(N2, Nu);

        //Inicializamos el primer valor del vector que coincide con toda la primera
columna
        for (i = 0;i < N2;i++) {
                EE[i][0] = 1;
        }

        k = 0;
        for (i = 1;i < N2;i++) {
                for (j = 1 + k;j < Nu;j++) {
                        EE[j][i] = FF[k][0];
                }
                k += 1;
        }


        //CALCULO DE LA MATRIZ G
        //Reservamos espacio para la variable que contendrá la convolucion para calcular G
y el vector máximo de E
        double *convG;
        convG = reservarVEC(Nu + nB - 1);

        double *Eaux;
        Eaux = reservarVEC(Nu);

        //Extraccion de la componente con todos los valores que tendrá E en todo el
horizonte
        for (i = 0;i < Nu;i++) {
                Eaux[i] = EE[Nu - 1][i];
        }


        //Calculo de la convolucion de E y B
        convolucion(Nu, nB, Eaux, BB, convG);
```

```
	//Y por ultimo colocamos los términos de la convolución en las diagonales para
obtener G

	for (i = 0;i < N2;i++) {
		for (j = 0;j < Nu;j++) {
			if (i - j >= 0) {
				GG[i][j] = convG[i - j];
			}
		}
	}
}

void calcGF(double *AA, double *BB, int N2, int Nu, int na, int nB, double** Gf) {

	int i, j, k;

	int nA = na + 1;

	double delta[] = { 1, -1 };

	double *AD;
	AD = reservarVEC(nA);

	convolucion(na, 2, AA, delta, AD);


	//CALCULO DE LA MATRIZ F

	double **FF;
	FF = reservarMAT(N2, nA - 1);

	//Calculo del valor de la primera fila de F (obtenido de dividir AD entre 1)

	for (i = 1;i < nA;i++) {
		FF[0][i - 1] = -AD[i];
	}


	for (i = 0;i < N2 - 1;i++) {
		for (j = 0;j <= nA - 2;j++) {
			FF[i + 1][j] = FF[i][j + 1] - FF[i][j] * AD[j + 1];

		}
		FF[i + 1][1] = -FF[i][0] * AD[nA - 1];
	}

	//CALCULO DE LA MATRIZ E

	double **EE;
	EE = reservarMAT(N2, Nu);

	//Inicializamos el primer valor del vector que coincide con toda la primera
columna
	for (i = 0;i < N2;i++) {
		EE[i][0] = 1;
	}

	k = 0;
	for (i = 1;i < N2;i++) {
		for (j = 1 + k;j < Nu;j++) {
			EE[j][i] = FF[k][0];
		}
		k += 1;
	}
```

```
        double *Eaux;//En este vector guardaremos las filas sucesivas de E
        Eaux = reservarVEC(Nu);

        double *aux;//En este vector guardaremos el valor de la convolución sucesivamente
        aux = reservarVEC(Nu + (nB - 1));

        double **Gfaux;
        Gfaux = reservarMAT(N2, (nB - 1));

        for (i = 0;i < N2;i++) {
                for (j = 0;j < Nu;j++) {
                        //Extraemos en cada bucle la componente de E correspondiente
                        Eaux[j] = EE[i][j];
                }
                convolucion(i + nB, nB, Eaux, BB, aux);

                for (j = 0;j < (nB - 1);j++) {
                        Gfaux[i][j] = aux[i + 1];
                }
        }

        //Colcamos en la matriz definitiva para Gf primero Gfaux y luego F

        for (i = 0;i < N2;i++) {
                for (j = 0;j < (nB - 1);j++) {
                        Gf[i][j] = Gfaux[i][j];
                }

        }

        for (i = 0;i < N2;i++) {
                for (j = nB - 1, k = 0;j < (nA + nB - 2), k < (nA - 1);j++, k++) {
                        Gf[i][j] = FF[i][k];
                }
        }
}



//FUNCIÓN DE CONVOLUCIÓN DE VECTORES

void convolucion(int NA, int NB, double A[], double B[], double *Res) {

        int Naux = NA + NB - 1;
        int i, j, k;

        double **Maux;
        Maux = reservarMAT(NB + 1, Naux);


        k = 0;
        for (i = 0;i < NB;i++) {
                for (j = 0;j < NA;j++) {
                        Maux[i][j + k] = B[i] * A[j];
                }
                k += 1;
        }
```

```
	for (i = 0;i < NB;i++) {
		for (j = 0;j < Naux;j++) {
			Maux[i + 1][j] = Maux[i + 1][j] + Maux[i][j];
		}
	}


	for (i = 0;i < Naux;i++) {
		Res[i] = Maux[NB][i];
	}

}


//FUNCION TRASPUESTA DE UNA MATRIZ

void traspMAT(double **M, int N1, int N2, double **Res) {

	int i, j;

	for (i = 0; i < N1;i++) {
		for (j = 0;j < N2;j++) {
			Res[j][i] = M[i][j];
		}
	}
}


//FUNCIONES DE PRODUCTO ENTRE MATRICES Y ENTRE VECTORES (RESPECTIVAMENTE)

void multiMAT(double **M1, double **M2, int N1, int N2, int N3, double **Res) {
	//N1 = FILAS M1-----------N2 = COLUMNAS M1-------------N3 = COLUMNAS M2

	//N2 debe ser el número compartido por las matrices para ser operable (COLUMNAS de 
M1, FILAS de M2)

	int i, j, k;
	double aux;

	for (i = 0; i < N1;i++) {//i para las filas de la matriz Resultado
		for (j = 0;j < N3;j++) {//j para las columnas de la matriz Resultado
			aux = 0;

			for (k = 0;k < N2;k++) {//k para apoyarnos a la hora de hacer la 
multiplicación
				aux = aux + (M1[i][k] * M2[k][j]);
			}
			Res[i][j] = aux;
		}
	}
}

void multiVEC(double *V1, double *V2, int N, double Res) {

	//En esto caso únicamente necesitamos una dimensión para ser multiplicables y el 
resultado es un escalar

	int i;

	for (i = 0;i < N;i++) {
		Res = Res + (V1[i] * V2[i]);
	}
}
```

```
//FUNCIONES DE PRODUCTO ENTRE MATRICES-VECTORES y VECTORES-MATRICES (RESPECTIVAMENTE)

void MATxVEC(double **M, int N1, int N2, double *V, double *Res) {

	//En este caso sólo necesitaremos las dimensiones de la matriz ya que las
dimensiones del vector serán N1

	int i, j;
	double aux = 0;

	for (i = 0; i < N1;i++) {
		for (j = 0;j < N2;j++) {
			aux = aux + (M[i][j] * V[j]);
		}
		Res[i] = aux;
		aux = 0;
	}
}

void VECxMAT(double **M, int N1, int N2, double *V, double *Res) {

	//En este caso sólo necesitaremos las dimensiones de la matriz ya que las
dimensiones del vector serán N2

	int i, j;
	double aux = 0;

	for (j = 0;j < N2;j++) {
		for (i = 0;i < N1;i++) {
			aux = aux + (V[i] * M[i][j]);
		}
		Res[j] = aux;
		aux = 0;
	}
}


//FUNCIONES DE RESERVA DE ESPACIO EN MEMORIA PARA MATRICES Y VECTORES RESPECTIVAMENTE

double** reservarMAT(int filas, int columnas) {

	double **mat;
	int i;

	mat = (double**)calloc(filas, sizeof(double*));
	if (mat == NULL) {
		printf("No se ha podido reservar memoria.");
		exit(1);
	}
	for (i = 0; i < filas; i++) {
		mat[i] = (double*)calloc(columnas, sizeof(double));
		if (mat[i] == NULL) {
			printf("No se ha podido reservar memoria.");
			exit(1);
		}
	}

	return mat;
}
```

```c
double* reservarVEC(int filas) {

	double *vec;

	vec = (double*)calloc(filas, sizeof(double));
	if (vec == NULL) {
		printf("No se ha podido reservar memoria.");
		exit(1);
	}


	return vec;
}



//Plot de matrices y vectores

void printMAT(double **mat, int filas, int columnas) {

	int i, j;

	for (i = 0;i < filas;i++) {
		for (j = 0, printf("\n");j < columnas;j++) {
			printf("%f\t", mat[i][j]);
		}
	}printf("\n\n");

}

void printVEC(double *vec, int filas) {

	int i;

	for (i = 0;i < filas;i++) {
		printf("%f\n", vec[i]);
	}printf("\n\n");

}


//FUNCION DE COMPROBACIÓN DE ERRORES GENERAL DE GUROBI

void Erroracion(GRBenv *env) {

	printf("ERROR = %s\n", GRBgeterrormsg(env));
	printf("PUES HUBO ERRORAZION\n");
	system("pause");
	exit(1);

}
```

ANEXO F: CÓDIGO COMPLETO DE LA IMPLEMENTACIÓN DE UNA LIBRERÍA PARA EL CÁLCULO DE UN GPC EN LABVIEW (GPCCOMPLETO)

```
//Se incluyen todas las librerías que necesitará el código
#include "stdafx.h"
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include "GPCcompleto.h"

#include <gurobi_c++.h>


// FUNCIONES DE LA LIBRERÍA, SIEMPRE EJECUTADAS CON WINAPI \\

//Funciones para el cálculo de las matrices necesarias invariantes en el control
predictivo
void WINAPI calcG(double *AA, double *BB, int N2, int Nu, int na, int nB, double** GG);
void WINAPI calcGF(double *AA, double *BB, int N2, int Nu, int na, int nB, double** Gf);


//Funciones para las operaciones necesarias con vectores y matrices:
void WINAPI convolucion(int NA, int NB, double A[], double B[], double *Res);
void WINAPI traspMAT(double **M, int N1, int N2, double **Res);
void WINAPI multiMAT(double **M1, double **M2, int N1, int N2, int N3, double **Res);
double WINAPI multiVEC(double *V1, double *V2, int N);
void WINAPI MATxVEC(double **M, int N1, int N2, double *V, double *Res);
void WINAPI VECxMAT(double **M, int N1, int N2, double *V, double *Res);


//Funciones de linealización y reconstruccion por filas de una matriz:
void WINAPI LinMat(double **M, int N1, int N2, double *Res);
void WINAPI ConstMat(double *V, int N1, int N2, double **Res);


//Funciones de reserva de espacio en memoria para matrices y vectores
double** WINAPI reservarMAT(int filas, int columnas);
double* WINAPI reservarVEC(int filas);


//Funciones propias relativas a la resolución de Gurobi
void WINAPI Erroracion(GRBenv *env);


void WINAPI DevuelveG(double* A, double *B, int nA, int nB, int N2, int Nu, int Ntotal,
double* Glin) {
```

```
/*///////////////////////////////////////////////////////////
FUNCIÓN QUE CALCULA Y DEVUELVE LA MATRIZ G LINEALIZADA //
/*//////////////////////////////////////////////////////////

	//Reserva de memoria y posterior calculo de la matriz G
	double **G;
	G = reservarMAT(N2, Nu);

	calcG(A, B, N2, Nu, nA, nB, G);

	//Linealización de G por filas para poder ser enviada a través de la dll en
labview como Vector
	LinMat(G, N2, Nu, Glin);

}


void WINAPI DevuelveGf(double* A, double *B, int nA, int nB, int N2, int Nu, int Ntotal,
double* Gflin) {

/*///////////////////////////////////////////////////////////
FUNCIÓN QUE CALCULA Y DEVUELVE LA MATRIZ Gf LINEALIZADA//
/*//////////////////////////////////////////////////////////

	//Debemos saber las dimensiones de esta matriz de antemano, y serán: (N2) filas y
(nB-1)+(nA-1)
	double **Gf;
	Gf = reservarMAT(N2, (nA + nB - 1));

	calcGF(A, B, N2, Nu, nA, nB, Gf);

	//Linealización de Gf por filas
	LinMat(Gf, N2, (nA + nB - 1), Gflin);

}


void WINAPI BucleGPC(double *Glin, double *Gflin, int nA, int nB, int N2, int Nu, double
lambda, double W, double Y, double U, int Modo, double *Ugurobi, int *error, double
*Usal,/*Displays auxiliares*/     double *ukpast0, double *ukpast1, double *deltaupast0,
double *ypast0, double *ypast1, double *deltaupast1, double *ukpast2) {
```

```
	/*/////////////////////////////////////////////////////////////////////////
		FUNCIÓN QUE ENGLOBA EL CÁLCULO COMPLETO DE LA FUNCIÓN OBJETIVO
		JUNTO CON LA RESOLUCIÓN DEL PROBLEMA DE OPTIMIZACIÓN CON GUROBI
	/*/////////////////////////////////////////////////////////////////////////

	int i, j, k;

	/*
	INIALIZACIÓN Y RECONSTRUCCIÓN DE LAS MATRICES PARA EL CÁLCULO DEL GCP
	*/

	//Reserva de memoria para las matrices
	double **G;
	G = reservarMAT(N2, Nu);
	double **Gf;
	Gf = reservarMAT(N2, (nA + nB));

	//Reconstrucción de las matrices linealizadas por filas

	ConstMat(Glin, N2, Nu, G);
	ConstMat(Gflin, N2, (nA + nB), Gf);



	/*
	INIALIZACIÓN DE VARIABLES DE CONTROL NECESARIAS
	*/

	static int startup = 0;//Variable para la iniciación de los valores para el primer
bucle

	static double *deltaUpast;//Incremento de la señal de control pasadas
	static double *ukpast;//Vector de las salidas pasadas necesarios (se inicializa en
el primer bucle en cero)
	static double *ypast;//Vector con las entradas pasadas necesarias

	//Reserva de memoria de las variables estáticas en la primera ejecución del bucle
	if (startup == 0) {

		deltaUpast = reservarVEC(nB);
		ukpast = reservarVEC(nB + 1);//Nesario para calcular la actualizacion del
vector anterior
		ypast = reservarVEC(nA);//si estamos en el primer bucle, el segundo valor
será cero,hay que inicializarlo igualandolo al primero

		startup = 1;
	}

	//Actualización de las salidas pasadas del sistema
	for (i = nA - 1;i > 0;i--) {

		ypast[i] = ypast[i - 1];

	}ypast[0] = Y;//Actualización del vector salida


	//Inicialización del vector de referencias futuras
	double *w;
	w = reservarVEC(N2);
	for (i = 0;i < N2;i++) {
		w[i] = W;//Se aplica la convención de referencias constantes en el
horizonte
	}
```

```
/*////////////////////////////////////////////////////////////////
       FORMULACION COMPLETA DE LA FUNCIÓN OBJETIVO
////////////////////////////////////////////////////////////////

       SIENDO EL VECTOR U, VECTOR CON LAS VARIACIONES DE CONTROL DEL HORIZONTE DE
PREDICCION NU

       Minimizar:      J    =      1/2 * UT * 2(GT * G + lambda*I(NU,NU)) * U    +   2(f -
W)T * G * U    +   (f - W)T*  (f - W)

                                  J      =      1/2 * UT *            H                 *
U    +    bT     * G * U      +          f0

                                                  (------------TERMINO CUADRÁTICO------------)
(--TERMINO LINEAL--)    (--TERMINO INDEPENDIENTE--)

       */


       //En primer lugar se va a calcular el valor del término "(f-w)" que está presente
en los términos LINEAL e INDEPENDIENTE
       //La matriz Gf se divide por columnas en primer lugar, (nB - 1) columnas
multiplicadas por un término del vector ukpast
       for (i = 0;i < (nB - 1);i++) {
              for (j = 0;j < N2;j++) {
                     Gf[j][i] = Gf[j][i] * ukpast[i];
              }
       }

       //Las demás columnas (nA) deben ser multiplicadas por las entradas pasadas, ykpast
       //en primera instancia éstas serán siempre el valor Y

       for (i = (nB - 1);i < (nA + nB - 1);i++) {
              for (j = 0;j < N2;j++) {
                     Gf[j][i] = Gf[j][i] * ypast[i - 1];
              }
       }

       //Para finalizar el cálculo se debe sumar entre sí todos los términos de las filas
       //Y cada uno restar el valor correspondiente de la referencia, W
       double *Gfaux;
       Gfaux = reservarVEC(N2);

       double f_reslib;
       for (i = 0;i < N2;i++) {
              f_reslib = 0;
              for (j = 0;j < (nA + nB - 1);j++) {
                     f_reslib += Gf[i][j];
              }
              Gfaux[i] = f_reslib - w[i];
       }
```

```
	/*///////////////////////////////////////////////////////////////
	CALCULO DE CADA UNO DE LOS TÉRMINOS
	DE FORMA INDEPENDIENTE PARA LA FUNCiÓN OBJETIVO
	*////////////////////////////////////////////////////////////////


	/*
	CALCULO DEL TÉRMINO INDEPENDIENTE
	*/

	double f0 = multiVEC(Gfaux, Gfaux, N2);


	/*
	CALCULO DE LOS TÉRMINOS LINEALES
	*/

	//Primero se multiplica Gfaux por 2 y luego lo multiplicaremos por G


	for (i = 0;i < N2;i++) {
		Gfaux[i] = Gfaux[i] * 2;//Sobreescribir los valores en el propio vector
	}

	double *linval;//el vector resultado contendrá los valores que acompañan los
términos lineales en orden creciente
	linval = reservarVEC(Nu);

	VECxMAT(G, N2, Nu, Gfaux, linval);


	/*
	CALCULO DE LOS TÉRMINOS CUADRÁTICOS
	*/

	//En primer lugar se realizará el cálculo matricial de H

	//Se crea y guarda G traspuesta
	double **GT;
	GT = reservarMAT(Nu, N2);

	traspMAT(G, N2, Nu, GT);


	//Creamos la matriz H y en ella guardamos el resultado de la multiplicación de G
trapuesta por G
	double **H;
	H = reservarMAT(Nu, Nu);

	multiMAT(GT, G, Nu, N2, Nu, H);


	//Como la matriz identidad multiplicada por lambda va a ser una matriz con
ÚNICAMENTE LAMBDAS EN LA DIAGONAL
	//Se suman directamente el valor de lambda a la diagonal de la matriz H.

	for (i = 0, j = 0;i < Nu, j < Nu;i++, j++) {
		H[i][j] += lambda;
	}
```

```
	//Se suman los términos que ocupan el lugar por encima de la diagonal
	//en los términos correspondientes de la matriz triangular inferior
	//Al representar a las mismas variables cuadráticas

	for (i = 0;i < Nu;i++) {
		for (j = 0;j < Nu;j++) {
			if (i < j) {
				H[i][j] += H[j][i];
			}
		}
	}



	/*//////////////////////////////////////////////////////////////////
	PROCESO DE OPTIMIZACIÓN DE LA FUNCIÓN OBJETIVO
	Y UTILIZACIÓN DE LAS LIBRERÍAS GUROBI
	*//////////////////////////////////////////////////////////////////

	/*
	CÁLCULO PREVIO DE VARIABLES CUADRÁTICAS NECESARIO PARA LA OPTIMIZACIÓN EN GUROBI
	*/

	//Calculo del número de variables cuadráticas
	int Nqvar;
	Nqvar = ((Nu*Nu - Nu) / 2) + Nu;

	//Calculo genérico de la combinación de los índices de las variables cuadráticas
en vectores
	//Deben estar en formato "int" para que pueda ser utilizable a posteriori por la
función de Gurobi


	//Por lo que reservamos exceptionalmente espacio en memoria dentro del código
	int *qfil; int *qcol;

	qfil = (int*)calloc(Nqvar, sizeof(int));
	if (qfil == NULL) {
		printf("No se ha podido reservar memoria.");
		exit(1);
	}

	qcol = (int*)calloc(Nqvar, sizeof(int));
	if (qcol == NULL) {
		printf("No se ha podido reservar memoria.");
		exit(1);
	}


	//Bucles de asignación de los índices para las variables cuadráticas en dos
vectores
	k = 0;
	for (i = 0;i < Nu;i++) {
		for (j = i;j < Nu;j++) {
			qfil[k] = i;
			qcol[k] = j;
			k++;
		}
	}
```

```
	//Se inicializa el vector que contendrá los valores de todas las variables
cuadráticas en orden
	double *qval;
	qval = reservarVEC(Nqvar);

	k = 0;
	for (i = 0;i < Nu;i++) {
		for (j = 0;j < Nu;j++) {
			if (i <= j) {
				qval[k] = H[i][j];
				k++;
			}
		}
	}


	/*
	DECLARACIÓN DE VARIABLES NECESARIAS PARA LA OPTIMIZACION EN GUROBI
	*/

	GRBenv *env = NULL;
	GRBmodel *model = NULL;


	double *deltaU;
	deltaU = reservarVEC(Nu);

	/*
	INICIALIZACIÓN DEL ENTORNO Y EL MODELO
	*/

	//CREACION DEL ENTORNO\\

	*error = GRBloadenv(&env, "NuevaUK.log");
	if (*error != 0) { Erroracion(env); }

	//CREACION DEL MODELO VACIO\\

	*error = GRBnewmodel(env, &model, "NuevaUK", 0, NULL, NULL, NULL, NULL, NULL);
	if (*error != 0) { Erroracion(env); }


	/*
	ADICIÓN DE LOS TODAS LAS VARIABLES AL MODELO VACÍO
	*/

	//ADICIÓN DE VARIABLES LINEALES\\
	//Tendremos tantas variables lineales iguales a Nu

	*error = GRBaddvars(model, Nu, 0, NULL, NULL, NULL, linval, NULL, NULL, NULL,
NULL);
	if (*error != 0) { Erroracion(env); }

	//ADICIÓN DE VARIABLES CUADRÁTRICAS\\

	*error = GRBaddqpterms(model, Nqvar, qfil, qcol, qval);
	if (*error != 0) { Erroracion(env); }

	//ADICIÓN DEL TÉRMINO INDEPENDIENTE\\

	*error = GRBsetdblattr(model, GRB_DBL_ATTR_OBJCON, f0);
	if (*error != 0) { Erroracion(env); }
```

```
/*
COMPROBACIÓN DEL SENTIDO DE LA OPTIMIZACIÓN
*/

//Para control predictivo siempre se elegirá minimización
*error = GRBsetintattr(model, GRB_INT_ATTR_MODELSENSE, GRB_MINIMIZE);
if (*error != 0) { Erroracion(env); }


/*
OPTIMIZACION DEL MODELO
*/

*error = GRBoptimize(model);
if (*error != 0) { Erroracion(env); }

/*
CAPTURA DE SOLUCION
*/

*error = GRBgetdblattrarray(model, GRB_DBL_ATTR_X, 0, Nu, deltaU);
if (*error != 0) { Erroracion(env); }


/*
LIBERACIÓN DEL MODELO Y DEL ENTORNO (en este orden específico)
*/

GRBfreemodel(model);
GRBfreeenv(env);



/*///////////////////////////
SELECCION DE MODO          //
*//////////////////////////


//Actualizamos el valor de la variable de salida según el modo elegido
if (Modo == 0) {

       *Ugurobi = deltaU[0];
       *Usal = U;

}
else {

       *Ugurobi = deltaU[0];
       *Usal = U + deltaU[0];

}


/*
ACTUALIZACION DE VALORES DEL BUCLE DE CONTROL
*/

//Se deben actualizar los dos vectores referentes a las señales de control pasadas
```

```
        //En primer lugar se actualiza el vector variaciones de control pasadas
        for (i = nB - 1; i > 0; i--) {
               deltaUpast[i] = deltaUpast[i - 1];
        }
        deltaUpast[0] = deltaU[0];

        //A continuación se actualiza el vector de salidas de control pasadas
        for (i = nB; i > 0;i--) {
               ukpast[i] = ukpast[i - 1];
        }
        ukpast[0] = *Usal;

        *ukpast0 = ukpast[0];
        *ukpast1 = ukpast[1];
        *ukpast2 = ukpast[2];
        *deltaupast0 = deltaUpast[0];
        *deltaupast1 = deltaUpast[1];
        *ypast1 = ypast[1];
        *ypast0 = ypast[0];

}


//FUNCIONES PARA EL CALCULO DE LAS MATRICES NECESARIAS PARA EL GPC

void WINAPI calcG(double *AA, double *BB, int N2, int Nu, int na, int nB, double** GG) {

        int i, j, k;

        int nA = na + 1;

        double delta[] = { 1, -1 };

        double *AD;
        AD = reservarVEC(nA);

        convolucion(na, 2, AA, delta, AD);


        //CALCULO DE LA MATRIZ F

        double **FF;
        FF = reservarMAT(N2, nA - 1);

        //Calculo del valor de la primera fila de F (obtenido de dividir AD entre 1)

        for (i = 1;i < nA;i++) {
               FF[0][i - 1] = -AD[i];
        }


        for (i = 0;i < N2 - 1;i++) {
               for (j = 0;j <= nA - 2;j++) {
                      FF[i + 1][j] = FF[i][j + 1] - FF[i][j] * AD[j + 1];

               }
               FF[i + 1][1] = -FF[i][0] * AD[nA - 1];
        }
```

```
	//CALCULO DE LA MATRIZ E

	double **EE;
	EE = reservarMAT(N2, Nu);

	//Inicializamos el primer valor del vector que coincide con toda la primera
columna
	for (i = 0;i < N2;i++) {
		EE[i][0] = 1;
	}

	k = 0;
	for (i = 1;i < N2;i++) {
		for (j = 1 + k;j < Nu;j++) {
			EE[j][i] = FF[k][0];
		}
		k += 1;
	}


	//CALCULO DE LA MATRIZ G
	//se reserva espacio para la variable que contendrá la convolucion para calcular G
y el vector máximo de E
	double *convG;
	convG = reservarVEC(Nu + nB - 1);

	double *Eaux;
	Eaux = reservarVEC(Nu);

	//Extraccion de la componente con todos los valores que tendrá E en todo el
horizonte
	for (i = 0;i < Nu;i++) {
		Eaux[i] = EE[Nu - 1][i];
	}


	//Calculo de la convolucion de E y B
	convolucion(Nu, nB, Eaux, BB, convG);

	//Y por ultimo colocamos los términos de la convolución en las diagonales para
obtener G

	for (i = 0;i < N2;i++) {
		for (j = 0;j < Nu;j++) {
			if (i - j >= 0) {
				GG[i][j] = convG[i - j];
			}
		}
	}
}

void WINAPI calcGF(double *AA, double *BB, int N2, int Nu, int na, int nB, double** Gf) {

	int i, j, k;

	int nA = na + 1;

	double delta[] = { 1, -1 };

	double *AD;
	AD = reservarVEC(nA);

	convolucion(na, 2, AA, delta, AD);
```

```
	//CALCULO DE LA MATRIZ F

	double **FF;
	FF = reservarMAT(N2, nA - 1);

	//Calculo del valor de la primera fila de F (obtenido de dividir AD entre 1)

	for (i = 1;i < nA;i++) {
		FF[0][i - 1] = -AD[i];
	}


	for (i = 0;i < N2 - 1;i++) {
		for (j = 0;j <= nA - 2;j++) {
			FF[i + 1][j] = FF[i][j + 1] - FF[i][j] * AD[j + 1];

		}
		FF[i + 1][1] = -FF[i][0] * AD[nA - 1];
	}



	//CALCULO DE LA MATRIZ E

	double **EE;
	EE = reservarMAT(N2, Nu);

	//Inicializamos el primer valor del vector que coincide con toda la primera
columna
	for (i = 0;i < N2;i++) {
		EE[i][0] = 1;
	}

	k = 0;
	for (i = 1;i < N2;i++) {
		for (j = 1 + k;j < Nu;j++) {
			EE[j][i] = FF[k][0];
		}
		k += 1;
	}

	double *Eaux;//En este vector se guardará las filas sucesivas de E
	Eaux = reservarVEC(Nu);

	//CÁLCULO DE Gf ORIGINAL BASADO EN EL LIBRO, MODEL PREDICTIVE CONTROL

	double *aux;//En este vector guardaremos el valor de la convolución sucesivamente
	aux = reservarVEC(Nu + (nB - 1));

	double **Gfaux;
	Gfaux = reservarMAT(N2, (nB - 1));

	for (i = 0;i < N2;i++) {
		for (j = 0;j < Nu;j++) {
			//Extraemos en cada bucle la componente de E correspondiente
			Eaux[j] = EE[i][j];
		}
		convolucion(i + nB, nB, Eaux, BB, aux);

		for (j = 0;j < (nB - 1);j++) {
			Gfaux[i][j] = aux[i + 1];
		}
	}
```

```
	//Colcamos en la matriz definitiva para Gf primero Gfaux y luego F

	for (i = 0;i < N2;i++) {
		for (j = 0;j < (nB - 1);j++) {
			Gf[i][j] = Gfaux[i][j];
		}

	}

	for (i = 0;i < N2;i++) {
		for (j = nB - 1, k = 0;j < (nA + nB - 2), k < (nA - 1);j++, k++) {
			Gf[i][j] = FF[i][k];
		}
	}

}



//FUNCIÓN DE CONVOLUCIÓN DE VECTORES

void WINAPI convolucion(int NA, int NB, double A[], double B[], double *Res) {

	int Naux = NA + NB - 1;
	int i, j, k;

	double **Maux;
	Maux = reservarMAT(NB + 1, Naux);


	k = 0;
	for (i = 0;i < NB;i++) {
		for (j = 0;j < NA;j++) {
			Maux[i][j + k] = B[i] * A[j];
		}
		k += 1;
	}


	for (i = 0;i < NB;i++) {
		for (j = 0;j < Naux;j++) {
			Maux[i + 1][j] = Maux[i + 1][j] + Maux[i][j];
		}
	}


	for (i = 0;i < Naux;i++) {
		Res[i] = Maux[NB][i];
	}

}


//FUNCION TRASPUESTA DE UNA MATRIZ

void WINAPI traspMAT(double **M, int N1, int N2, double **Res) {

	int i, j;

	for (i = 0; i < N1;i++) {
		for (j = 0;j < N2;j++) {
			Res[j][i] = M[i][j];
		}
```

```
	}
}


//FUNCIONES DE PRODUCTO ENTRE MATRICES Y ENTRE VECTORES (RESPECTIVAMENTE)

void WINAPI multiMAT(double **M1, double **M2, int N1, int N2, int N3, double **Res) {
	//N1 = FILAS M1-----------N2 = COLUMNAS M1-------------N3 = COLUMNAS M2

	//N2 debe ser el número compartido por las matrices para ser operable (COLUMNAS de 
M1, FILAS de M2)

	int i, j, k;
	double aux;

	for (i = 0; i < N1;i++) {//i para las filas de la matriz Resultado
		for (j = 0;j < N3;j++) {//j para las columnas de la matriz Resultado
			aux = 0;

			for (k = 0;k < N2;k++) {//k para apoyarnos a la hora de hacer la 
multiplicación
				aux = aux + (M1[i][k] * M2[k][j]);
			}
			Res[i][j] = aux;
		}
	}
}

double WINAPI multiVEC(double *V1, double *V2, int N) {

	//En esto caso únicamente se necesita una dimensión para ser multiplicables y el 
resultado es un escalar

	int i;
	double Res = 0;

	for (i = 0;i < N;i++) {
		Res = Res + (V1[i] * V2[i]);
	}
	return Res;
}

//FUNCIONES DE PRODUCTO ENTRE MATRICES-VECTORES y VECTORES-MATRICES (RESPECTIVAMENTE)

void WINAPI MATxVEC(double **M, int N1, int N2, double *V, double *Res) {

	//En este caso sólo se necesitarán las dimensiones de la matriz ya que las 
dimensiones del vector serán N1

	int i, j;
	double aux = 0;

	for (i = 0; i < N1;i++) {
		for (j = 0;j < N2;j++) {
			aux = aux + (M[i][j] * V[j]);
		}
		Res[i] = aux;
		aux = 0;
	}
}
```

```c
void WINAPI VECxMAT(double **M, int N1, int N2, double *V, double *Res) {

	//En este caso sólo se necesitarán las dimensiones de la matriz ya que las dimensiones del vector serán N2

	int i, j;
	double aux = 0;

	for (j = 0;j < N2;j++) {
		for (i = 0;i < N1;i++) {
			aux = aux + (V[i] * M[i][j]);
		}
		Res[j] = aux;
		aux = 0;
	}
}


//FUNCION PARA LA LINEALIZACIÓN Y RECONSTRUCCION DE MATRICES POR FILAS

void WINAPI LinMat(double **M, int N1, int N2, double *Res) {

	int i, j;
	int k = 0;

	for (i = 0;i < N1;i++) {
		for (j = 0;j < N2;j++) {
			Res[k] = M[i][j];
			k += 1;
		}
	}

}

void WINAPI ConstMat(double *V, int N1, int N2, double **Res) {

	int i, j;
	int k = 0;

	for (i = 0;i < N1;i++) {
		for (j = 0;j < N2;j++) {
			Res[i][j] = V[k];
			k += 1;
		}
	}

}
```

```
//FUNCIONES DE RESERVA DE ESPACIO EN MEMORIA PARA MATRICES Y VECTORES RESPECTIVAMENTE

double** WINAPI reservarMAT(int filas, int columnas) {

	double **mat;
	int i;

	mat = (double**)calloc(filas, sizeof(double*));
	if (mat == NULL) {
		printf("No se ha podido reservar memoria.");
		exit(1);
	}
	for (i = 0; i < filas; i++) {
		mat[i] = (double*)calloc(columnas, sizeof(double));
		if (mat[i] == NULL) {
			printf("No se ha podido reservar memoria.");
			exit(1);
		}
	}

	return mat;
}

double* WINAPI reservarVEC(int filas) {

	double *vec;

	vec = (double*)calloc(filas, sizeof(double));
	if (vec == NULL) {
		printf("No se ha podido reservar memoria.");
		exit(1);
	}


	return vec;
}



//FUNCION DE COMPROBACIÓN DE ERRORES GENERAL DE GUROBI

void WINAPI Erroracion(GRBenv *env) {

	printf("ERROR = %s\n", GRBgeterrormsg(env));
	printf("PUES HUBO ERRORAZION\n");
	system("pause");
	//exit(1);

}
```

ANEXO G: CÓDIGO COMPLETO CON LA ADICIÓN DE RESTRICCIONES DE ENTRADA Y SALIDA (GPCRESTRICT)

```
//Se incluyen todas las librerías que necesitará el código
#include "stdafx.h"
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include "GPCrestrict.h"

#include <gurobi_c++.h>


// FUNCIONES DE LA LIBRERÍA, SIEMPRE EJECUTADAS CON WINAPI \\

//Funciones para el cálculo de las matrices necesarias invariantes en el control
predictivo
void WINAPI calcG(double *AA, double *BB, int N2, int Nu, int na, int nB, double** GG);
void WINAPI calcGF(double *AA, double *BB, int N2, int Nu, int na, int nB, double** Gf);


//Funciones para las operaciones necesarias con vectores y matrices:
void WINAPI convolucion(int NA, int NB, double A[], double B[], double *Res);
void WINAPI traspMAT(double **M, int N1, int N2, double **Res);
void WINAPI multiMAT(double **M1, double **M2, int N1, int N2, int N3, double **Res);
double WINAPI multiVEC(double *V1, double *V2, int N);
void WINAPI MATxVEC(double **M, int N1, int N2, double *V, double *Res);
void WINAPI VECxMAT(double **M, int N1, int N2, double *V, double *Res);


//Funciones de linealización y reconstruccion por filas de una matriz:
void WINAPI LinMat(double **M, int N1, int N2, double *Res);
void WINAPI ConstMat(double *V, int N1, int N2, double **Res);


//Funciones de reserva de espacio en memoria para matrices y vectores
double** WINAPI reservarMAT(int filas, int columnas);
double* WINAPI reservarVEC(int filas);


//Funciones propias relativas a la resolución de Gurobi
void WINAPI Erroracion(GRBenv *env);


void WINAPI DevuelveG(double* A, double *B, int nA, int nB, int N2, int Nu, int Ntotal,
double* Glin) {

/*//////////////////////////////////////////////////////
FUNCIÓN QUE CALCULA Y DEVUELVE LA MATRIZ G LINEALIZADA//
/*/////////////////////////////////////////////////////

	//Reserva de memoria y posterior calculo de la matriz G
	double **G;
	G = reservarMAT(N2, Nu);

	calcG(A, B, N2, Nu, nA, nB, G);
```

```
	//Linealización de G por filas para poder ser enviada a través de la dll en
labview como Vector
	LinMat(G, N2, Nu, Glin);

}


void WINAPI DevuelveGf(double* A, double *B, int nA, int nB, int N2, int Nu, int Ntotal,
double* Gflin) {

/*/////////////////////////////////////////////////////////
FUNCIÓN QUE CALCULA Y DEVUELVE LA MATRIZ Gf LINEALIZADA //
/*////////////////////////////////////////////////////////

	//Debemos saber las dimensiones de esta matriz de antemano, y serán: (N2) filas y
(nB-1)+(nA-1)
	double **Gf;
	Gf = reservarMAT(N2, (nA + nB - 1));

	calcGF(A, B, N2, Nu, nA, nB, Gf);

	//Linealización de Gf por filas
	LinMat(Gf, N2, (nA + nB - 1), Gflin);

}


void WINAPI BucleGPC(double *Glin, double *Gflin, int nA, int nB, int N2, int Nu, double
lambda, double W, double Y, double U, int Modo, double *Ugurobi, int *error, double
*Usal,   /*Variables restricciones*/    double Ulim, double Ylim,   /*Displays
auxiliares*/ double *ukpast0, double *ukpast1, double *deltaupast0, double *ypast0,
double *ypast1, double *deltaupast1, double *ukpast2) {


/*//////////////////////////////////////////////////////////////////////
	FUNCIÓN QUE ENGLOBA EL CÁLCULO COMPLETO DE LA FUNCIÓN OBJETIVO
	JUNTO CON LA RESOLUCIÓN DEL PROBLEMA DE OPTIMIZACIÓN CON GUROBI
/*//////////////////////////////////////////////////////////////////////



	int i, j, k;

	/*
	INIALIZACIÓN Y RECONSTRUCCIÓN DE LAS MATRICES PARA EL CÁLCULO DEL GCP
	*/

	//Reserva de memoria para las matrices
	double **G;
	G = reservarMAT(N2, Nu);
	double **Gf;
	Gf = reservarMAT(N2, (nA + nB));

	//Reconstrucción de las matrices linealizadas por filas

	ConstMat(Glin, N2, Nu, G);
	ConstMat(Gflin, N2, (nA + nB), Gf);



	/*
	INIALIZACIÓN DE VARIABLES DE CONTROL NECESARIAS
	*/
```

```
	static int startup = 0;//Variable para la iniciación de los valores para el primer
bucle

	static double *deltaUpast;//Incremento de la señal de control pasadas
	static double *ukpast;//Vector de las salidas pasadas necesarios (se inicializa en
el primer bucle en cero)
	static double *ypast;//Vector con las entradas pasadas necesarias

	//Reserva de memoria de las variables estáticas en la primera ejecución del bucle
	if (startup == 0) {

		deltaUpast = reservarVEC(nB);
		ukpast = reservarVEC(nB + 1);//Nesario para calcular la actualizacion del
vector anterior
		ypast = reservarVEC(nA);//si estamos en el primer bucle, el segundo valor
será cero,hay que inicializarlo igualandolo al primero

		startup = 1;
	}

	//Actualización de las salidas pasadas del sistema
	for (i = nA - 1;i > 0;i--) {

		ypast[i] = ypast[i - 1];

	}ypast[0] = Y;//Actualización del vector salida


	//Inicialización del vector de referencias futuras
	double *w;
	w = reservarVEC(N2);
	for (i = 0;i < N2;i++) {
		w[i] = W;//Se aplica la convención de referencias constantes en el
horizonte
	}


/*//////////////////////////////////////////////////////////////////
	FORMULACION COMPLETA DE LA FUNCIÓN OBJETIVO
///////////////////////////////////////////////////////////////////


	SIENDO EL VECTOR U, VECTOR CON LAS VARIACIONES DE CONTROL DEL HORIZONTE DE
PREDICCION NU

	Minimizar:      J    =      1/2 * UT * 2(GT * G + lambda*I(NU,NU)) * U    +   2(f -
W)T * G * U    +   (f - W)T*  (f - W)

				J      =      1/2 * UT *            H                 *
U    +    bT     * G * U      +         f0

				     (------------TERMINO CUADRÁTICO------------)
(--TERMINO LINEAL--)    (--TERMINO INDEPENDIENTE--)

	*/
```

```
	//En primer lugar se va a calcular el valor del término "(f-w)" que está presente 
en los términos LINEAL e INDEPENDIENTE
	//La matriz Gf se divide por columnas en primer lugar, (nB - 1) columnas 
multiplicadas por un término del vector ukpast
	for (i = 0;i < (nB - 1);i++) {
		for (j = 0;j < N2;j++) {
			Gf[j][i] = Gf[j][i] * ukpast[i];
		}
	}

	//Las demás columnas (nA) deben ser multiplicadas por las entradas pasadas, ykpast
	//en primera instancia éstas serán siempre el valor Y

	for (i = (nB - 1);i < (nA + nB - 1);i++) {
		for (j = 0;j < N2;j++) {
			Gf[j][i] = Gf[j][i] * ypast[i - 1];
		}
	}

	//Para finalizar el cálculo se debe sumar entre sí todos los términos de las filas
	//Y cada uno restar el valor correspondiente de la referencia, W
	double *Gfaux;
	Gfaux = reservarVEC(N2);

	double f_reslib;
	for (i = 0;i < N2;i++) {
		f_reslib = 0;
		for (j = 0;j < (nA + nB - 1);j++) {
			f_reslib += Gf[i][j];
		}
		Gfaux[i] = f_reslib - w[i];
	}


	/*//////////////////////////////////////////////////
	CALCULO DE CADA UNO DE LOS TÉRMINOS
	DE FORMA INDEPENDIENTE PARA LA FUNCiÓN OBJETIVO
	*//////////////////////////////////////////////////


	/*
	CALCULO DEL TÉRMINO INDEPENDIENTE
	*/

	double f0 = multiVEC(Gfaux, Gfaux, N2);


	/*
	CALCULO DE LOS TÉRMINOS LINEALES
	*/

	//Primero se multiplica Gfaux por 2 y luego lo multiplicaremos por G


	for (i = 0;i < N2;i++) {
		Gfaux[i] = Gfaux[i] * 2;//Sobreescribir los valores en el propio vector
	}

	double *linval;//el vector resultado contendrá los valores que acompañan los 
términos lineales en orden creciente
	linval = reservarVEC(Nu);

	VECxMAT(G, N2, Nu, Gfaux, linval);
```

```
	/*
	CALCULO DE LOS TÉRMINOS CUADRÁTICOS
	*/

	//En primer lugar se realizará el cálculo matricial de H

	//Se crea y guarda G traspuesta
	double **GT;
	GT = reservarMAT(Nu, N2);

	traspMAT(G, N2, Nu, GT);


	//Creamos la matriz H y en ella guardamos el resultado de la multiplicación de G trapuesta por G
	double **H;
	H = reservarMAT(Nu, Nu);

	multiMAT(GT, G, Nu, N2, Nu, H);


	//Como la matriz identidad multiplicada por lambda va a ser una matriz con ÚNICAMENTE LAMBDAS EN LA DIAGONAL
	//Se suman directamente el valor de lambda a la diagonal de la matriz H.

	for (i = 0, j = 0;i < Nu, j < Nu;i++, j++) {
		H[i][j] += lambda;
	}

	//Se suman los términos que ocupan el lugar por encima de la diagonal
	//en los términos correspondientes de la matriz triangular inferior
	//Al representar a las mismas variables cuadráticas

	for (i = 0;i < Nu;i++) {
		for (j = 0;j < Nu;j++) {
			if (i < j) {
				H[i][j] += H[j][i];
			}
		}
	}


	/*//////////////////////////////////////////////////////////////////
	PROCESO DE OPTIMIZACIÓN DE LA FUNCIÓN OBJETIVO
	Y UTILIZACIÓN DE LAS LIBRERÍAS GUROBI
	*//////////////////////////////////////////////////////////////////


	/*
		CÁLCULO PREVIO DE VARIABLES CUADRÁTICAS NECESARIO PARA LA OPTIMIZACIÓN EN GUROBI
	*/

	//Calculo del número de variables cuadráticas
	int Nqvar;
	Nqvar = ((Nu*Nu - Nu) / 2) + Nu;

	//Calculo genérico de la combinación de los índices de las variables cuadráticas en vectores
	//Deben estar en formato "int" para que pueda ser utilizable a posteriori por la función de Gurobi
```

```
	//Por lo que reservamos exceptionalmente espacio en memoria dentro del código
	int *qfil; int *qcol;

	qfil = (int*)calloc(Nqvar, sizeof(int));
	if (qfil == NULL) {
		printf("No se ha podido reservar memoria.");
		exit(1);
	}

	qcol = (int*)calloc(Nqvar, sizeof(int));
	if (qcol == NULL) {
		printf("No se ha podido reservar memoria.");
		exit(1);
	}


	//Bucles de asignación de los índices para las variables cuadráticas en dos
vectores
	k = 0;
	for (i = 0;i < Nu;i++) {
		for (j = i;j < Nu;j++) {
			qfil[k] = i;
			qcol[k] = j;
			k++;
		}
	}

	//Se inicializa el vector que contendrá los valores de todas las variables
cuadráticas en orden
	double *qval;
	qval = reservarVEC(Nqvar);

	k = 0;
	for (i = 0;i < Nu;i++) {
		for (j = 0;j < Nu;j++) {
			if (i <= j) {
				qval[k] = H[i][j];
				k++;
			}
		}
	}


	/*
	DECLARACIÓN DE VARIABLES NECESARIAS PARA LA OPTIMIZACION EN GUROBI
	*/

	GRBenv *env = NULL;
	GRBmodel *model = NULL;


	double *deltaU;
	deltaU = reservarVEC(Nu);

	/*
	INICIALIZACIÓN DEL ENTORNO Y EL MODELO
	*/

	//CREACION DEL ENTORNO\\

	*error = GRBloadenv(&env, "NuevaUK.log");
	if (*error != 0) { Erroracion(env); }

	//CREACION DEL MODELO VACIO\\
```

```
	*error = GRBnewmodel(env, &model, "NuevaUK", 0, NULL, NULL, NULL, NULL, NULL);
	if (*error != 0) { Erroracion(env); }


	/*
	ADICIÓN DE LOS TODAS LAS VARIABLES AL MODELO VACÍO
	*/

	//ADICIÓN DE VARIABLES LINEALES\\
	//Tendremos tantas variables lineales iguales a Nu

	*error = GRBaddvars(model, Nu, 0, NULL, NULL, NULL, linval, NULL, NULL, NULL,
NULL);
	if (*error != 0) { Erroracion(env); }

	//ADICIÓN DE VARIABLES CUADRÁTRICAS\\

	*error = GRBaddqpterms(model, Nqvar, qfil, qcol, qval);
	if (*error != 0) { Erroracion(env); }

	//ADICIÓN DEL TÉRMINO INDEPENDIENTE\\

	*error = GRBsetdblattr(model, GRB_DBL_ATTR_OBJCON, f0);
	if (*error != 0) { Erroracion(env); }



	//ADICIÓN DE RESTRICCIONES DE VARIABLES DE CONTROL\\

	int indcU[] = { 0 };
	double linvalU[] = { 1 };

	for (i = 0;i < Nu;i++) {

		*error = GRBaddconstr(model, 1, indcU, linvalU, GRB_LESS_EQUAL, Ulim,
NULL);
		if (*error != 0) { Erroracion(env); }

		*error = GRBaddconstr(model, 1, indcU, linvalU, GRB_GREATER_EQUAL, -Ulim,
NULL);
		if (*error != 0) { Erroracion(env); }

		indcU[0] += 1;
	}


	//ADICIÓN DE RESTRICCIONES A LA SEÑAL DE SALIDA\\
	//En primer lugar debemos realizar los cálculos despejando al vector U, que es en
torno al que se hace la restricción

	//Inicializamos y reservamos espacio para las variables que contiene los indices
de las variables lineales
	int *indcY;
	indcY = (int*)calloc(Nu, sizeof(int));
	if (indcY == NULL) {
		printf("No se ha podido reservar memoria.");
		exit(1);
	}

	for (i = 0;i < Nu;i++) {
		indcY[i] = i;
	}
```

```
	//Y además los valores de G ordenados por filas
	double* linvalY;
	linvalY = reservarVEC(Nu);


	for (j = 0;j < N2;j++) {

		for (i = 0;i < Nu;i++) {
			linvalY[i] = G[j][i];
		}

		*error = GRBaddconstr(model, Nu, indcY, linvalY, GRB_LESS_EQUAL, Ylim,
NULL);
		if (*error != 0) { Erroracion(env); }

	}


	/*
	COMPROBACIÓN DEL SENTIDO DE LA OPTIMIZACIÓN
	*/

	//Para control predictivo siempre se elegirá minimización
	*error = GRBsetintattr(model, GRB_INT_ATTR_MODELSENSE, GRB_MINIMIZE);
	if (*error != 0) { Erroracion(env); }


	/*
		OPTIMIZACION DEL MODELO
	*/

	*error = GRBoptimize(model);
	if (*error != 0) { Erroracion(env); }

	/*
		CAPTURA DE SOLUCION
	*/

	*error = GRBgetdblattrarray(model, GRB_DBL_ATTR_X, 0, Nu, deltaU);
	if (*error != 0) { Erroracion(env); }


	/*
		LIBERACIÓN DEL MODELO Y DEL ENTORNO (en este orden específico)
	*/

	GRBfreemodel(model);
	GRBfreeenv(env);
```

```
/*///////////////////////////
       SELECCION DE MODO    //
*///////////////////////////


       //Actualizamos el valor de la variable de salida según el modo elegido
       if (Modo == 0) {

              *Ugurobi = deltaU[0];
              *Usal = U;

       }
       else {

              *Ugurobi = deltaU[0];
              *Usal = U + deltaU[0];

       }


       /*
       ACTUALIZACION DE VALORES DEL BUCLE DE CONTROL
       */

       //Se deben actualizar los dos vectores referentes a las señales de control pasadas


       //En primer lugar se actualiza el vector variaciones de control pasadas
       for (i = nB - 1; i > 0; i--) {
              deltaUpast[i] = deltaUpast[i - 1];
       }
       deltaUpast[0] = deltaU[0];

       //A continuación se actualiza el vector de salidas de control pasadas
       for (i = nB; i > 0;i--) {
              ukpast[i] = ukpast[i - 1];
       }
       ukpast[0] = *Usal;

       *ukpast0 = ukpast[0];
       *ukpast1 = ukpast[1];
       *ukpast2 = ukpast[2];
       *deltaupast0 = deltaUpast[0];
       *deltaupast1 = deltaUpast[1];
       *ypast1 = ypast[1];
       *ypast0 = ypast[0];

}


//FUNCIONES PARA EL CALCULO DE LAS MATRICES NECESARIAS PARA EL GPC

void WINAPI calcG(double *AA, double *BB, int N2, int Nu, int na, int nB, double** GG) {

       int i, j, k;

       int nA = na + 1;

       double delta[] = { 1, -1 };

       double *AD;
       AD = reservarVEC(nA);

       convolucion(na, 2, AA, delta, AD);
```

```
        //CALCULO DE LA MATRIZ F

        double **FF;
        FF = reservarMAT(N2, nA - 1);

        //Calculo del valor de la primera fila de F (obtenido de dividir AD entre 1)

        for (i = 1;i < nA;i++) {
               FF[0][i - 1] = -AD[i];
        }


        for (i = 0;i < N2 - 1;i++) {
               for (j = 0;j <= nA - 2;j++) {
                      FF[i + 1][j] = FF[i][j + 1] - FF[i][j] * AD[j + 1];

               }
               FF[i + 1][1] = -FF[i][0] * AD[nA - 1];
        }



        //CALCULO DE LA MATRIZ E

        double **EE;
        EE = reservarMAT(N2, Nu);

        //Inicializamos el primer valor del vector que coincide con toda la primera
columna
        for (i = 0;i < N2;i++) {
               EE[i][0] = 1;
        }

        k = 0;
        for (i = 1;i < N2;i++) {
               for (j = 1 + k;j < Nu;j++) {
                      EE[j][i] = FF[k][0];
               }
               k += 1;
        }


        //CALCULO DE LA MATRIZ G
        //se reserva espacio para la variable que contendrá la convolucion para calcular G
y el vector máximo de E
        double *convG;
        convG = reservarVEC(Nu + nB - 1);

        double *Eaux;
        Eaux = reservarVEC(Nu);

        //Extraccion de la componente con todos los valores que tendrá E en todo el
horizonte
        for (i = 0;i < Nu;i++) {
               Eaux[i] = EE[Nu - 1][i];
        }
```

```
	//Calculo de la convolucion de E y B
	convolucion(Nu, nB, Eaux, BB, convG);

	//Y por ultimo colocamos los términos de la convolución en las diagonales para
obtener G

	for (i = 0;i < N2;i++) {
		for (j = 0;j < Nu;j++) {
			if (i - j >= 0) {
				GG[i][j] = convG[i - j];
			}
		}
	}
}

void WINAPI calcGF(double *AA, double *BB, int N2, int Nu, int na, int nB, double** Gf) {

	int i, j, k;

	int nA = na + 1;

	double delta[] = { 1, -1 };

	double *AD;
	AD = reservarVEC(nA);

	convolucion(na, 2, AA, delta, AD);


	//CALCULO DE LA MATRIZ F

	double **FF;
	FF = reservarMAT(N2, nA - 1);

	//Calculo del valor de la primera fila de F (obtenido de dividir AD entre 1)

	for (i = 1;i < nA;i++) {
		FF[0][i - 1] = -AD[i];
	}


	for (i = 0;i < N2 - 1;i++) {
		for (j = 0;j <= nA - 2;j++) {
			FF[i + 1][j] = FF[i][j + 1] - FF[i][j] * AD[j + 1];

		}
		FF[i + 1][1] = -FF[i][0] * AD[nA - 1];
	}
```

```
	//CALCULO DE LA MATRIZ E

	double **EE;
	EE = reservarMAT(N2, Nu);

	//Inicializamos el primer valor del vector que coincide con toda la primera
columna
	for (i = 0;i < N2;i++) {
		EE[i][0] = 1;
	}

	k = 0;
	for (i = 1;i < N2;i++) {
		for (j = 1 + k;j < Nu;j++) {
			EE[j][i] = FF[k][0];
		}
		k += 1;
	}

	double *Eaux;//En este vector se guardará las filas sucesivas de E
	Eaux = reservarVEC(Nu);

	//CÁLCULO DE Gf ORIGINAL BASADO EN EL LIBRO, MODEL PREDICTIVE CONTROL

	double *aux;//En este vector guardaremos el valor de la convolución sucesivamente
	aux = reservarVEC(Nu + (nB - 1));

	double **Gfaux;
	Gfaux = reservarMAT(N2, (nB - 1));

	for (i = 0;i < N2;i++) {
		for (j = 0;j < Nu;j++) {
			//Extraemos en cada bucle la componente de E correspondiente
			Eaux[j] = EE[i][j];
		}
		convolucion(i + nB, nB, Eaux, BB, aux);

		for (j = 0;j < (nB - 1);j++) {
			Gfaux[i][j] = aux[i + 1];
		}
	}

	//Colcamos en la matriz definitiva para Gf primero Gfaux y luego F

	for (i = 0;i < N2;i++) {
		for (j = 0;j < (nB - 1);j++) {
			Gf[i][j] = Gfaux[i][j];
		}

	}

	for (i = 0;i < N2;i++) {
		for (j = nB - 1, k = 0;j < (nA + nB - 2), k < (nA - 1);j++, k++) {
			Gf[i][j] = FF[i][k];
		}
	}

}
```

```
//FUNCIÓN DE CONVOLUCIÓN DE VECTORES

void WINAPI convolucion(int NA, int NB, double A[], double B[], double *Res) {

	int Naux = NA + NB - 1;
	int i, j, k;

	double **Maux;
	Maux = reservarMAT(NB + 1, Naux);


	k = 0;
	for (i = 0;i < NB;i++) {
		for (j = 0;j < NA;j++) {
			Maux[i][j + k] = B[i] * A[j];
		}
		k += 1;
	}


	for (i = 0;i < NB;i++) {
		for (j = 0;j < Naux;j++) {
			Maux[i + 1][j] = Maux[i + 1][j] + Maux[i][j];
		}
	}


	for (i = 0;i < Naux;i++) {
		Res[i] = Maux[NB][i];
	}

}


//FUNCION TRASPUESTA DE UNA MATRIZ

void WINAPI traspMAT(double **M, int N1, int N2, double **Res) {

	int i, j;

	for (i = 0; i < N1;i++) {
		for (j = 0;j < N2;j++) {
			Res[j][i] = M[i][j];
		}
	}
}
```

```
//FUNCIONES DE PRODUCTO ENTRE MATRICES Y ENTRE VECTORES (RESPECTIVAMENTE)

void WINAPI multiMAT(double **M1, double **M2, int N1, int N2, int N3, double **Res) {
	//N1 = FILAS M1-----------N2 = COLUMNAS M1-------------N3 = COLUMNAS M2

	//N2 debe ser el número compartido por las matrices para ser operable (COLUMNAS de
M1, FILAS de M2)

	int i, j, k;
	double aux;

	for (i = 0; i < N1;i++) {//i para las filas de la matriz Resultado
		for (j = 0;j < N3;j++) {//j para las columnas de la matriz Resultado
			aux = 0;

			for (k = 0;k < N2;k++) {//k para apoyarnos a la hora de hacer la
multiplicación
				aux = aux + (M1[i][k] * M2[k][j]);
			}
			Res[i][j] = aux;
		}
	}
}

double WINAPI multiVEC(double *V1, double *V2, int N) {

	//En esto caso únicamente se necesita una dimensión para ser multiplicables y el
resultado es un escalar

	int i;
	double Res = 0;

	for (i = 0;i < N;i++) {
		Res = Res + (V1[i] * V2[i]);
	}
	return Res;
}

//FUNCIONES DE PRODUCTO ENTRE MATRICES-VECTORES y VECTORES-MATRICES (RESPECTIVAMENTE)

void WINAPI MATxVEC(double **M, int N1, int N2, double *V, double *Res) {

	//En este caso sólo se necesitarán las dimensiones de la matriz ya que las
dimensiones del vector serán N1

	int i, j;
	double aux = 0;

	for (i = 0; i < N1;i++) {
		for (j = 0;j < N2;j++) {
			aux = aux + (M[i][j] * V[j]);
		}
		Res[i] = aux;
		aux = 0;
	}
}

void WINAPI VECxMAT(double **M, int N1, int N2, double *V, double *Res) {
```

```
	//En este caso sólo se necesitarán las dimensiones de la matriz ya que las 
dimensiones del vector serán N2

	int i, j;
	double aux = 0;

	for (j = 0;j < N2;j++) {
		for (i = 0;i < N1;i++) {
			aux = aux + (V[i] * M[i][j]);
		}
		Res[j] = aux;
		aux = 0;
	}
}


//FUNCION PARA LA LINEALIZACIÓN Y RECONSTRUCCION DE MATRICES POR FILAS

void WINAPI LinMat(double **M, int N1, int N2, double *Res) {

	int i, j;
	int k = 0;

	for (i = 0;i < N1;i++) {
		for (j = 0;j < N2;j++) {
			Res[k] = M[i][j];
			k += 1;
		}
	}

}

void WINAPI ConstMat(double *V, int N1, int N2, double **Res) {

	int i, j;
	int k = 0;

	for (i = 0;i < N1;i++) {
		for (j = 0;j < N2;j++) {
			Res[i][j] = V[k];
			k += 1;
		}
	}

}
```

```
//FUNCIONES DE RESERVA DE ESPACIO EN MEMORIA PARA MATRICES Y VECTORES RESPECTIVAMENTE

double** WINAPI reservarMAT(int filas, int columnas) {

	double **mat;
	int i;

	mat = (double**)calloc(filas, sizeof(double*));
	if (mat == NULL) {
		printf("No se ha podido reservar memoria.");
		exit(1);
	}
	for (i = 0; i < filas; i++) {
		mat[i] = (double*)calloc(columnas, sizeof(double));
		if (mat[i] == NULL) {
			printf("No se ha podido reservar memoria.");
			exit(1);
		}
	}

	return mat;
}

double* WINAPI reservarVEC(int filas) {

	double *vec;

	vec = (double*)calloc(filas, sizeof(double));
	if (vec == NULL) {
		printf("No se ha podido reservar memoria.");
		exit(1);
	}


	return vec;
}



//FUNCION DE COMPROBACIÓN DE ERRORES GENERAL DE GUROBI

void WINAPI Erroracion(GRBenv *env) {

	printf("ERROR = %s\n", GRBgeterrormsg(env));
	printf("PUES HUBO ERRORAZION\n");
	system("pause");
	//exit(1);

}
```

REFERENCIAS

[1] Gurobi Optimization Group, Equipo de Gurobi, *http://www.gurobi.com/company/management-team*.

[2] Gurobi Optimization Group, Aplicación industrial, *http://www.gurobi.com/products/industries/industry-overview*.

[3] Gurobi Optimization Group, Gurobi Compute Server, *http://www.gurobi.com/products/gurobi-compute-server/gurobi-compute-server*.

[4] Gurobi Optimization Group, Gubori Cloud, *http://www.gurobi.com/products/gurobi-cloud*.

[5] Gurobi Optimization Group, Gurobi Optimizer, *http://www.gurobi.com/products/gurobi-optimizer*.

[6] Gurobi Optimization Group, Academia Center, *http://www.gurobi.com/academia/academia-center*.

[7] Gurobi Optimization Group, Servicio de soporte, *support.gurobi.com*.

[8] Gurobi Optimization Group, Descarga de Gurobi Optimizer, *http://www.gurobi.com/downloads/gurobi-optimizer*.

[9] Gurobi Optimization Group, Documentación de Gurobi Optimizer, *http://www.gurobi.com/documentation*.

[10] Gurobi Optimization Group, Referencia de Atributos Gurobi, *https://www.gurobi.com/documentation/7.5/refman/attributes.html#sec:Attributes*.

[11] Gurobi Optimization Group, Referencia de Parámetros Gurobi, *https://www.gurobi.com/documentation/7.5/refman/parameters.html*.

[12] Microsoft, Acceso a la plataforma Visual Studio, *https://visualstudio.microsoft.com/*.

[13] E. F. Camacho and C. Bordons, Model predictive control, Springer-Verlag London Limited, 2004.

[14] E. F. Camacho, Carlos Bordons, Model Predictive Control, curso Máster Automática, Robótica y Telemática, Sevilla 2016.

[15] D.W. Clarke, C. Mohtadi, and P.S. Tuffs, Generalized Predictive Control. Part I. The basic algorithm. Automatica, 23(2):137-148, 1987.

[16] D.W. Clarke, C. Mohtadi, and P.S. Tuffs. Generalized Predictive Control. Part II. Extensions and Interpretations. Automatica, 23(2):149–160, 1987.

[17] Sadhana CHIDRAWAR, Balasaheb PATRE. Generalized Predictive Control and Neural Generalized

Predictive Control. Issue 13, 133-152, July-December 2008.

[18] National Instruments, LabVIEW User Manual. Edición Enero 1998.

[19] National Instruments, Building a DLL with Visual C++, 30 Novembre 2018, *http://www.ni.com/tutorial/3056/en/*.

[20] National Instruments, Using external code in LabVIEW, April 2003 Edition, *http://www.ni.com/pdf/manuals/370109b.pdf*.

[21] Brian W. Kernighan, Dennis M. Ritchie, The C programming language. Second Edition.