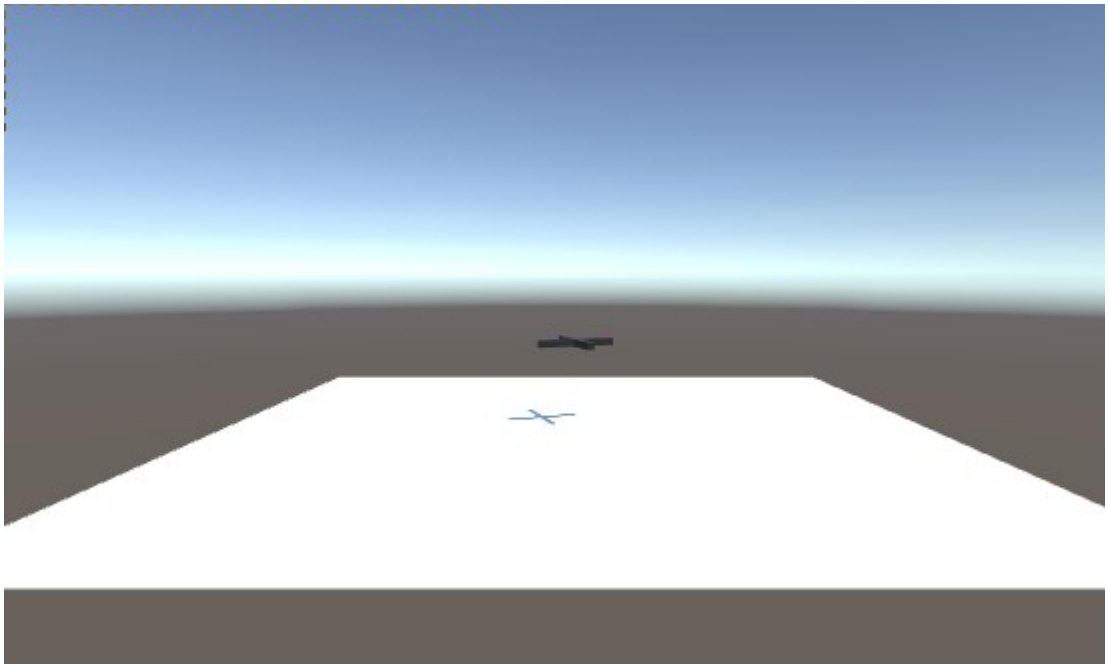
Memoria de Sistemas Inteligentes

Obtención de los valores PID de un sistema de control por medio de algoritmos genéticos



Realizado por:

Miguel Jiménez Arribas

Índice

1. Objetivo del proyecto.

La elección de la temática del proyecto se debió mi interés personal por todo tipo de sistema de control y especialmente por los sistemas de navegación inerciales aéreos. Por ello se decidió aprovechar esta práctica para aprender sobre los sistemas de control PID. Además, puesto que se trata de un proyecto para la asignatura de Sistemas Inteligentes y para ello era necesario algún tipo de "inteligencia" se decidió optar por la optimización de los valores PID mediante la utilización de algoritmos genéticos, algo para lo cuál se encuentran un gran número de *papers* en Internet[1].

De esta manera el proyecto comenzó con dos objetivos principales:

En primer lugar realizar una simulación lo más realista posible de manera en un futuro hipotético para la realización del proyecto en hardware real, el controlador PID y la mayoría del código pudieran reutilizarse.

En segundo lugar obtener mediante algoritmos genéticos unos valores PID que obtuvieran un comportamiento mejor que aquellos valores obtenidos mediante ensayo y error.

Con estos objetivos en mente, el proyecto consiste en la realización de una simulación en Unity3D de un quadcóptero estabilizado mediante tres controladores PID, uno para la altura, otro para la actitud y otro para la velocidad.

2. Explicación detallada del código.

Para la realización del proyecto de han creado 8 scripts en Unity: Altimeter.cs, CamControl.cs, FlightController.cs, GA.cs, Individual.cs, PIDController.cs, RotationSensor.cs y Rotor.cs. Ahora se analizará la funcionalidad de cada uno de ellos.

2.1. Altimeter.cs

Puesto que, como ya se ha explicado, uno de los objetivos era realizar una simulación lo más realista posible, se ha obtado por externalizar la obtención de datos de manera que actúe de forma más realista, puesto que en un controlador real se tendría por un lado la MCU (microcontrolador) y por otro cada uno de los sensores a los cuáles accedería desde la memoria principal.

De esta manera la clase Altimeter se actuliza en cada *frame* con la altura a la que se encuentra el quadcóptero y proveé a FlightController con una función con la cuál obtener dicho dato.

2.2. CamControl.cs

Este script controla la camara principal de Unity de manera que siga al quadcóptero en su movimiento.

1 Para más información sobre la bibliografía utilizada ir al Anexo 2.

2.3. FlightController.cs

Esta clase se encarga de realizar toda la simulación del quadcóptero. Cada *frame* llama a los métodos HeightStabiliser y AttitudeStabiliser, tras lo cuál realiza unos cuantos cálculos para el fitness de controlador, algo sobre lo que hablaremos en el próximo script.

El método HeightStabiliser obtiene la altura por medio de la clase Altimeter tras lo cuál pide a la clase PIDController que le devuelva el próximo valor de control, ajustando a partir de este la potencia mínima y máxima a la que puede funcionar cada uno de los rotores.

El método AttitudeStabiliser obtiene la actitud actual mediante la clase RotationSensor a partir de la cuál realiza los cálculos de cualquier controlador PID y establece la potencia exacta que deberán producir cada uno de los rotores, teniendo en cuenta las potencias mínimas y máximas calculadas en HeightStabiliser.

Para hacer que todo lo mencionado funcione es necesario que primero de todo se hayan inicializado todas las variables utilizadas por dichos métodos, para lo cuál existen dos métodos init() diferentes, uno para ejecutar en caso de que se encuentre en modo simulación y otro en caso de que esté en modo de aprendizaje, como se explicará en los próximos apartados.

2.4. GA.cs

Esta clase es la encargada de realizar el algoritmo genético. Nada más empezar comprueba si el programa está en modo simulación o en modo de aprendizaje (puede cambiarse seleccionando o no el valor booleano *simulation* desde el editor) llamando en cada caso a los métodos de inicialización correspondientes.

En el caso en que esté en modo simulación únicamente llamará al método de inicialización de FlightController y este se encargará de recoger los valores necesarios del editor.

En caso de que esté en modo de aprendizaje se crean aleatoriamente las poblaciones tras lo cual comienza la evolución. En el método *evolution* para cada generación se calcula el fitness y las posibilidades de selección de cada uno de los individuos. Tras esto se escoje un individuo de la población al azar según las probabilidades calculadas, tras lo cuál muta. Una vez hecho esto se le compara consigo mismo antes de la mutación y se añade el mejor de los dos a la nueva población.

Como puede verse, se ha obtado por una sola operación (mutar) puesto que al tratarse de números decimales, no tendría sentido que hubiera una operación de combinación.

Todo lo referente a la creación de individuos y al cálculo del *fitness* se expondrá en el apartado 4.

2.5. Individual.cs

Esta clase auxiliar implementa diferentes operaciones relacionadas con cada uno de los individuos que forman parte de la evolución llevada a cabo en el script GA.cs. Así tenemos operaciones como la creación aleatoria de individuos, la creación por copia, guardar el fitness calculado en dicho script, obtener dicho valor de vuelta y mutar.

El método mutar consiste en la obtención de dos números aleatorios, uno que decide si se suma o resta y otro la cantidad.

2.6. PIDController.cs

Esta clase centraliza todos los cálculos necesarios para actualizar un controlador PID de un momento al siguiente. En esta implementación se calcula el valor de salida a partir de un valor de entrada siguiendo la fórmula:

$$out = [bias] + [P * e(t)] + [I * \int e(t) * dt] + [D * de(t)/dt]$$

$$\text{siendo} \quad e(t) = valor\ objetivo - valor\ actual$$

Desafortunadamente, por motivos de falta de tiempo, esta clase solo se ha utilizado para el controlador de altitud (en FlightController.cs), mientras que en el controlador de actitud se ha tenido que implementar la misma funcionalidad *hardcoded* en FlightController.cs puesto que para este segundo se utilizan cuaterniones (*quaternions*) y no disponía del tiempo necesario para modificar PIDController de manera que soportara a estos.

2.7. RotationSensor.cs

De la misma forma que en Altimeter.cs, la clase RotationSensor se actualiza en cada *frame* con la actitud (rotación) del quadcóptero y proveé al controlador de una función con la cuál obtener esta.

2.8. Rotor.cs

Este script es el encargado de simular el comportamiento de cada uno de los rotores de un quadcóptero. Así genera la fuerza en cada uno de los cuatro vértices del quadcóptero y el par motor responsible del giro en el eje vertical (*yaw*).

Además, implementa las funciones utilizadas en la clase FlightController para establecer la potencia de cada uno de los motores.

3. Consideraciones a tener en cuenta.

El proyecto entregado cuenta con dos modo de operación: modo de aprendizaje y modo de simulación. Se puede cambiar entre ellos (des)marcando la variable booleana *simulation* desde el inspector del editor de Unity del script GA.cs, dentro del *GameObject Floor*.

Dependiendo del modo en el que se encuentre, los valores PID de cada controlador se leen de un sitios diferentes. En caso del modo simulación se pueden cambiar directamente desde el inspector del *GameObject Quadcopter*. Sin embargo, en modo aprendizaje dichos valores son generados aleatoriamente como se explicará en el siguiente apartado.

Además del modo, existen otras propiedades de la clase GA que pueden ser de interés modificar, como son el número de generaciónes o el de individuos, la probabilidad de mutación, el máximo cambio que puede darse un una sola mutación, el tiempo que dura la simulación de cada individuo y la escala del tiempo, para acelerar la ejecución (valores mayores que tres pueden dar problemas).

4. Resultados obtenidos.

Teniendo en cuenta los objetivos del proyecto establecidos anteriormente, puede verse como el transcurso de la práctica se ha dividido en dos partes bien diferenciadas. Por un lado la programación y testeo de la simulación con unos valores PID obtenidos mediante ensayo y error. En caso de que se quiera comprobar dichos valores fueron para la actitud PID: 50, 1, 3000, respectivamente, y para la altura, PID: 10, 0, 300. Y por otro lado la programación y aprendizaje de dichos valores por técnicas genéticas. Es sobre esto segundo sobre lo que se van a presentar los resultados a continuación.

En una primera aproximación se pensó en establecer inicialmente los valores PID del controlador de manera aleatoria, sin ningún tipo de conocimiento previo. Además se calculaba el *fitness* de cada individuo teniendo en cuenta únicamente lo cercano que estaban la velocidad (lineal y angular) a 0. Así tras tan solo una generación el algoritmo ya había encontrado que la población más estable era el suelo, por lo que establecia valores PID cercanos a 0 de manera que la potencia de los motores no era lo suficiente como para elevarlo.

Después se modificó el cálculo de *fitness* de manera que se tuvieran en cuenta tres cosas: la diferencia con la altura establecidad como objetivo (desde el inspector en FlightController.cs), la velocidad angular y la diferencia con la posición objetivo (no solo la altura). Se divide entre altura y posición para intentar prioritizar primero aquellas soluciones que se estabilizaran a la altura objetivo más rápidamente.

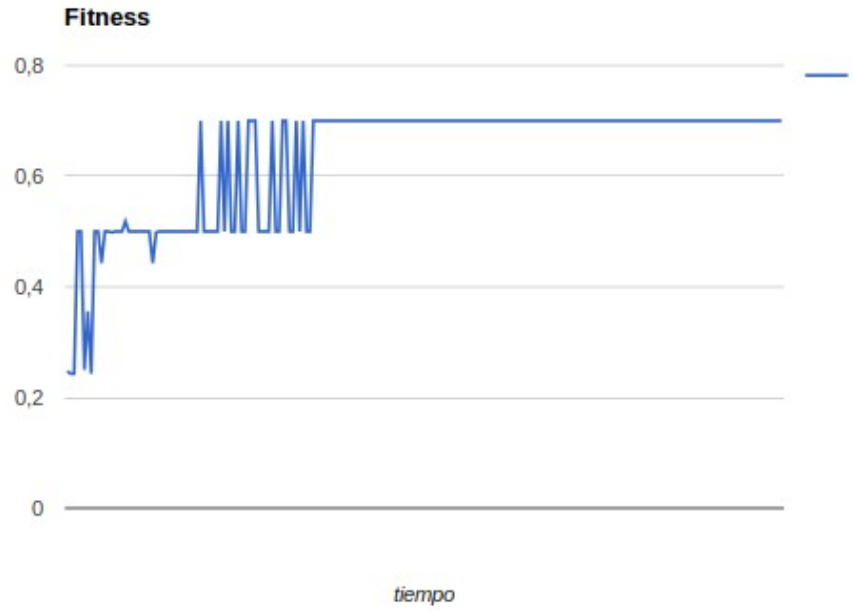
Sin embargo esta solución tampoco fue efectiva, pues tras haberlo dejado entrenando más de 12 horas no había llegado a ninguna solución más o menos efectiva. La mayoría de estas solian sufrir de lo que se conoce en inglés como *overshot*, en la que el quadcóptero aceleraba demasiado en una dirección tuviendo que decelerar a su vez más, etc, hasta que o salía despedido o simplemente se quedaba oscilando sin parar.

Finalmente se optó por una tercera opción que consiste en dos modificaciones diferentes. En primer lugar se decide utilizar conocimiento previo, puesto que como había hecho un gran número de pruebas con ensayo y error conocía que el valor D solía ser mucho más grando que el P y este a su vez del I. De esta manera en la creación aleatoria de inviduos en Individual.cs puede verse como como primer valor es de un orden 10 veces más grande que el segundo y este así a su vez del tercero.

Además respecto del cálculo del *fitness*, se decide tener en cuenta las oscilaciones anteriormente mencionadas, de manera que tras unos pequeños cambios en FlightController.cs tenemos que el *fitness* utiliza la suma ponderada del inverso del número de oscilaciones realizadas, la distancia a la altura especificada, la velocidad angular y la velocidad de ascenso inicial. Este último dato se tiene en cuenta para favorecer las soluciones que llegan a estabilizar el aparato de la manera más rápidamente posible, pues sino habían casos en los que no había oscilaciones pero que sin embargo tardaba más de 10 segundos en llegar hasta la posición deseada.

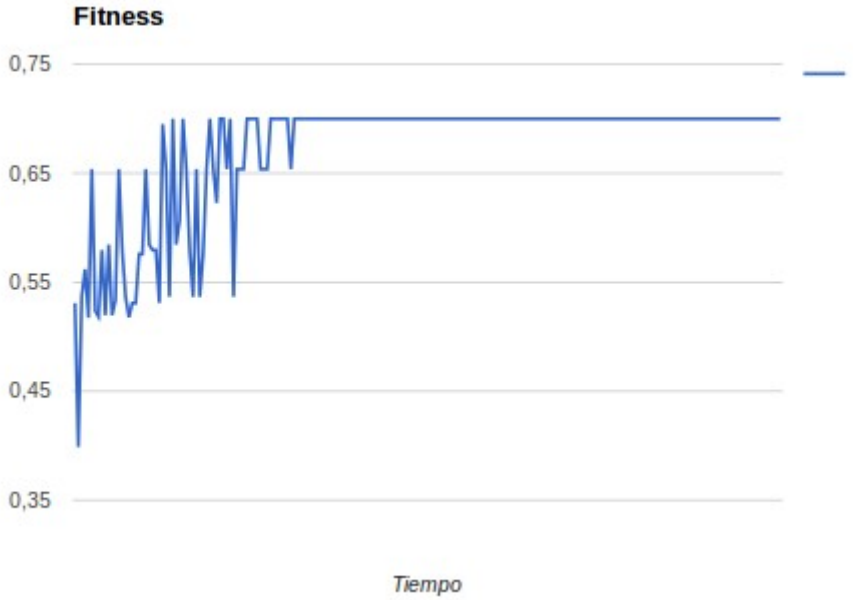
También cabe mencionar que en esta tercera aproximación, con el objetivo de que evolucionen aquellas soluciones más estables en relación con la actitud, a los tres cuartos del tiempo de simulación se aplica un par de fuerzas sobre el quadcóptero de manera que aquellos individuos más estables consigan estabilizarse antes de que termine el periodo de simulación.

A continuación se presentan dos gráficas para demostrar la mejora con el tiempo del *fitness* de las soluciones.



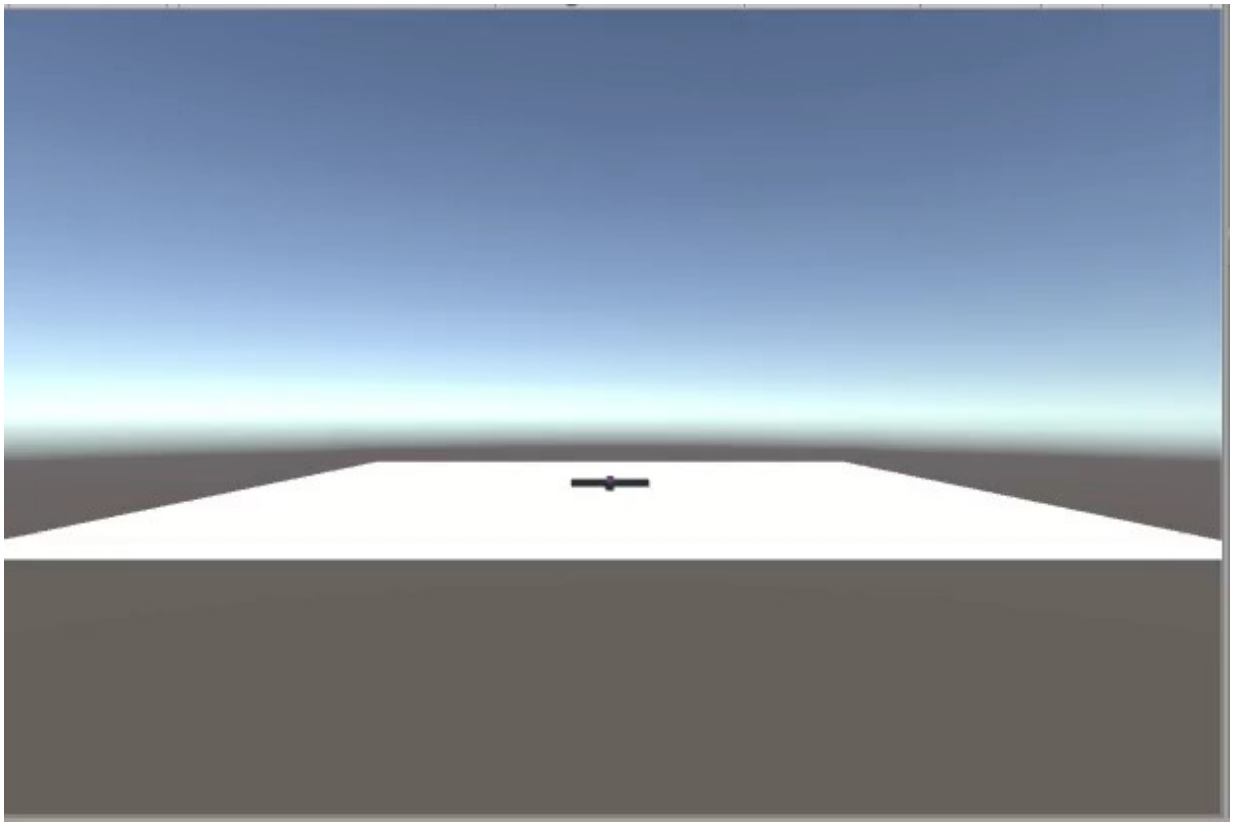
Este ejemplo está hecho utilizando la tercera aproximación explicada más arriba con generación de individuos aleatoria y un cambio máximo de 10.

Por otro lado también se han hecho pruebas partiendo de unos valores PID iniciales y disminuyendo el cambio máximo posible de manera que basandonos en una solución que sabemos es buena se vaya poco a poco acercando a una solución óptima. Es el caso de este segundo gráfico.

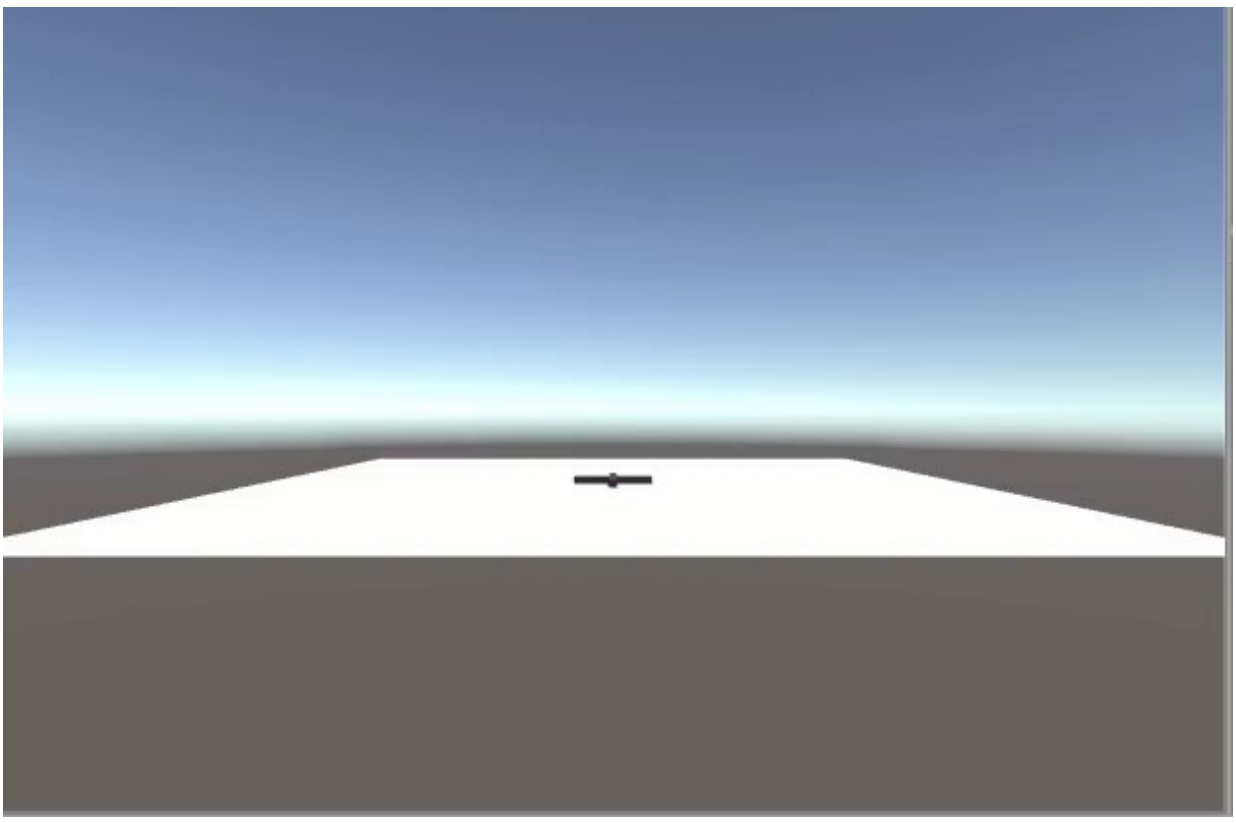


Puede verse como el *fitness* desde el que empieza es mayor. Sin embargo en este caso no se estaban utilizando la última forma de calcular el *fitness* dando como resultado un *fitness* y un comportamiento peor que el anterior.

Por último se presentan dos videos (en caso de que no puedan reproducirse pueden verse en /Course/Videos/). En el primero (out1.mkv) está simulándose un individuo durante la primera generación. Como puede verse no consigue estabilizarse ni en altitud ni en actitud.



En segundo lugar (out2.mkv) se trata de la mejor solución de la primera gráfica, con un *fitness* de 0,7 sobre 1. En este caso, pese a no ser todavía perfecto, puede observarse como resulta considerablemente más estable.



5. Conclusiones y mejoras.

Tras esta exposición llegamos finalmente a la conclusión. Mis sensaciones personales con el proyecto son satisfactorias. Desde mi punto de vista, el objetivo principal era el aprendizaje personal sobre métodos de control “tradicionales” y eso ha sido completamente conseguido. Así puede verse en la implementación de la simulación.

Sin embargo, es cierto que no se han llegado a cumplir todos los objetivos planteados en un principio. Por ejemplo, tras los grandes problemas que sufrí al intentar realizar el controlador PID de actitud me quedé sin tiempo para implementar el tercer controlador, el de velocidad, lo que quizá se manifiesta en la inestabilidad pitch-roll que presenta el quadcóptero, siendo el único eje de rotación completamente estable el eje vertical (yaw).

Respecto de la parte de aprendizaje mediante algoritmos genéticos se ha conseguido la obtención de unos valores con un comportamiento sensiblemente mejor en mantenimiento de la actitud y muy similar en altitud comparados a aquellos obtenidos mediante ensayo y error.

No obstante si hubiera dispuesto de más tiempo también hubiera realizado unas cuantas mejoras siendo las dos principales la detección temprana de una solución no válida, puesto que ahora mismo incluso aunque se estrelle contra el suelo todavía se debe esperar todo el tiempo de simulación para desechar al individuo; y la modificación de la función de fitness por una que tenga en cuenta también la velocidad de estabilización respecto de la actitud, puesto que ahora mismo mientras que al final de la simulación se encuentra parado recibe una puntación perfecta (respecto de su peso).

Por último un objetivo personal de cara al futuro es implementar este proyecto pero en hardware mediante un microcontrolador.

Anexo 1: Entrega.

Junto con esta memoria se entrega un paquete de Unity con todo lo necesario para importar el proyecto. Únicamente es necesario abrir un nuevo proyecto de Unity e ir a la pestaña de Assets de la barra de tareas. Ahí se escoge *Import Package* > *Custom Package* y se escoge el proyecto. Tras esto se hace doble click sobre la escena *scene1* y ya está todo listo.

Además el proyecto también se encuentra en Github, pudiendo clonarlo o descargar en zip desde: https://github.com/migueljiarr/quadcopterController, encontrándose la memoria en la carpeta "/Course/Memoria.pdf".

Anexo 2: Bibliografía.

- http://wordpress.richardhannah.ninja/honours-project/
- http://www.alanzucconi.com/2016/04/27/evolutionary-computation-4/
- PID Parameters Optimization by Using Genetic Algorithm. Andri Mirzal, Shinichiro Yoshii, Masashi Furukawa. https://arxiv.org/pdf/1204.0885.pdf
- PID-Controller Tuning Optimization with Genetic Algorithms in Servo Systems. Arturo Y. Jaen-Cuellar, Rene de J. Romero-Troncoso, Luis Morales-Velazquez, Roque A. Osornio-Rios. http://journals.sagepub.com/doi/pdf/10.5772/56697
- Controller Tuning Using Genetic Algorithms. Ecaterina Emilia Vladu, Toma Leonida Dragomi. http://uni-obuda.hu/conferences/saci04/vladu.pdf
- https://docs.unity3d.com/ScriptReference/