Trabajo Fin de Máster
Máster universitario en Ingeniería Aeronáutica

Optimización por estrategia evolutiva y su aplicación en el inicio del diseño preliminar de aeronaves

Autor: Germán Jimena de la Resurrección
Tutor: Francisco Javier Ros Padilla

Dpto. Matemática aplicada II
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2018

Trabajo Fin de Máster
Máster universitario en Ingeniería Aeronáutica

Optimización por estrategia evolutiva y su aplicación en el inicio del diseño preliminar de aeronaves

Autor:
Germán Jimena de la Resurrección

Tutor:
Francisco Javier Ros Padilla
Profesor titular

Dpto. de Matemática Aplicada II
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2018

Trabajo fin de máster: Optimización por estrategia evolutiva y su aplicación en el inicio del diseño preliminar de aeronaves

Autor: Germán Jimena de la Resurrección

Tutor: Francisco Javier Ros Padilla

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

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2018

Agradecimientos

Este trabajo es el último proyecto de una vida universitaria que cumple ya 7 años, y muchas personas han influenciado ese camino. Sería muy extenso mencionarlos a todos, de modo que seré breve.

Gracias a mi familia que ha servido de apoyo durante todos estos años.

Gracias al 'sofi' y al 'patito', los mejores profesores del Liceo por empujarme aún más hacia las ciencias.

Gracias a los grandes profesores del departamento de aeronáutica, por hacer que me haya vuelto un poco 'friki' de los aviones.

Gracias a mis compañeros de carrera por toda la ayuda que nos hemos prestado mutuamente.

Y gracias a mi tutor de este TFM por ayudarme a llevarlo a cabo.

Espero que este trabajo sea de vuestro agrado.

Resumen

En este trabajo hacemos un estudio sobre los algoritmos evolutivos, centrándonos en la estrategia evolutiva, para lo que nos serviremos de programas desarrollados en *Matlab 2016*. Hablaremos también del diseño de aeronaves y trataremos de aplicar lo estudiado en un caso práctico de desarrollo de algoritmos evolutivos para automatizar parte del diseño preliminar de una aeronave convencional de transporte, procedimiento que a día de hoy es un tanto 'artesanal'.

El diseño de una aeronave es un proceso complejo, que empieza con un diseño conceptual muy básico, prosigue con un diseño preliminar, del que tiene que salir un avión muy bien definido, y finaliza con un diseño de detalle, del que salen diseños de todos los elementos listos para fabricar. El programa que se desarrollará nos permite iniciar el diseño preliminar, en un punto en el que tenemos poca información y en el que un diseño convencional utilizaría datos de aeronaves semejantes. Para ello, el programa empleará estrategias evolutivas.

Una estrategia evolutiva es un sistema de optimización basado en la teoría de la evolución darwinista. El concepto general es crear una población de individuos con unas características aleatorias. Posteriormente se evalúan numéricamente en la función a optimizar cómo de 'buenos' son estos elementos y se eliminan los que presenten una peor puntuación. Los elementos restantes se mezclan, se reproducen y se mutan, para crear una población nueva que tendrá, en general, mejores características que la anterior. Comúnmente este procedimiento converge al máximo absoluto de la función a optimizar, evitando quedarse atascado en máximos locales y de forma relativamente rápida si los parámetros del algoritmo están bien escogidos.

En el caso concreto del problema de diseño de avión nos encontramos con un tema interesante y complejo. Esto es, el problema de cómo evaluar numéricamente lo 'bueno' que es un avión. Para ello se valorarán diversos parámetros, se adimensionalizarán para poder compararlos, y se fusionarán dando más importancia a lo que el usuario del código prefiera, llegando a una especie de función de optimización 'hecha a medida' para el usuario del programa, denominada función de *fitness*.

Además, es importante escoger adecuadamente los elementos de la población. El elemento 'avión' a optimizar es un vector que contiene información de la aeronave. Esta información debe ser adecuada para el algoritmo evolutivo, así como relevante en la evaluación de la aeronave. El problema de encontrar un grupo de parámetros que definan un avión de forma preliminar, que sean relevantes en la evaluación de la aeronave y que permitan funcionar correctamente el algoritmo evolutivo se tratará en un apartado dedicado a ello.

Finalmente, el funcionamiento del programa completo, de forma resumida, es el siguiente:

- El usuario del programa introduce requerimientos del diseño del avión.
- El programa crea una población de elementos 'avión' al azar.
- La población de elementos se optimiza mediante algoritmo evolutivo, usando una función de optimización de acuerdo a los criterios introducidos por el usuario.
- Cuando se cumple un criterio de parada, se detiene el algoritmo de optimización.
- Se presenta al usuario el mejor elemento creado.

Por último, con el código desarrollado haremos distintos ejemplos para llegar a conclusiones.

Índice

Índice de Tablas

Índice de Figuras

Notación

b	Envergadura
C_{D0}	Coeficiente de resistencia parásita
C_e	Consumo específico
C_L	Coeficiente de sustentación
DT	Desviación típica
E_{max}	Eficiencia aerodinámica máxima
HTP	Horizontal tail plane – Estabilizador horizontal
k	Coeficiente de resistencia inducido
MTOW	Maximum take-off weight
Re	Número de Reynolds
S_{wet}	Superficie mojada
T	Empuje
V	Velocidad
VTP	Vertical tail plane – Estabilizador vertical
W_{empty}	Peso en vacío de la aeronave
W_F	Peso de combustible
ν	Viscosidad cinemática
ρ	Densidad
Φ	Factor de Oswald

1 INTRODUCCIÓN

Optimización por estrategia evolutiva y su aplicación en el inicio del diseño preliminar de aeronaves. Tal es el título de este trabajo, y explicándolo es como vamos a comenzar esta introducción. Abordemos esta explicación desgranándolo en las dos partes que lo conforman, primero la 'optimización por estrategia evolutiva' y, posteriormente, el 'inicio del diseño preliminar de aeronaves'.

En la ingeniería siempre nos encontraremos con problemas de los que tenemos que buscar una solución óptima, y las matemáticas han desarrollado numerosos métodos analíticos para hallar dichos óptimos. Sin embargo, estos métodos no siempre funcionan en problemas complejos, muchas veces con máximos relativos, que para su solución requieren múltiples iteraciones y, en muchas ocasiones, la subjetividad de la experiencia.

Una alternativa a los métodos tradicionales de optimización son los algoritmos evolutivos, marco en el cual se encuentra la estrategia evolutiva. Para llevar a cabo este proceso de optimización debemos simplificar el problema llegando a condensarlo en una única función, que será la que optimicemos.

El proceso de optimización por estrategia evolutiva será explicado en detalle en su apartado correspondiente. De momento, adelantamos que se va a utilizar por su versatilidad, la capacidad de optimizar funciones complejas y de adaptarse a modificaciones.

El problema complejo que estamos diciendo de optimizar no es otro que la segunda parte del título del trabajo, el diseño preliminar de aeronaves.

El diseño preliminar de aeronaves es una ciencia mezclada con artesanía. Es una tarea multidisciplinar que conlleva a la colaboración entre ingenieros de distintas ramas, conocida como ingeniería concurrente.

El *input* en el diseño de aeronaves es, en general, un RFP (request for proposal) lanzado por algún cliente que necesita una solución a un determinado problema. Este RFP contiene las características esperadas, límites a elementos físicos, y, sobre todo, la misión que debe realizar la aeronave. Si no hubiera RFP como *input* de diseño, sería un estudio de mercado. Además de ello, también es un *input* importante el marco legal.

Partiendo de estos requerimientos y necesidades, un equipo multifuncional se sienta en una mesa y comienza a bocetar como sería el avión que cumpla esos requisitos. Es el comienzo de la fase de diseño conceptual, del cual el *output* es uno o varios diseños posibles, con una geometría caracterizada de forma básica y con unos objetivos de diseño planteados.

Una vez se tiene el diseño conceptual se pasa al siguiente nivel, en el que se trata de hacer el diseño preliminar. En esta fase se tienen muchos más *inputs*, no solo la geometría general del nivel anterior, sino que se empiezan a tener en cuenta los requerimientos legales, cargas, deflexiones, etc. El resultado de esta fase de diseño es un diseño maduro, que incluye la configuración externa completa, una configuración interna básica, principales cargas y tensiones, etc.

La última fase del diseño es el diseño de detalle. En esta fase es cuando se gasta la mayor parte del presupuesto, y el resultado es un diseño listo para ser fabricado, con todo lo que ello conlleva.

El objetivo del programa que se va a realizar es que sirva como apoyo en las primeras fases del diseño conceptual, cuando la mayor parte de los datos se obtiene de una simple comparación cualitativa con aeronaves semejantes. En lugar de ello, queremos poder obtener datos cuantitativos mediante un análisis automatizado de los requisitos del RFP.

Finalizamos esta introducción explicando la estructura del trabajo.

En primer lugar, hablaremos con más profundidad sobre el diseño de aeronaves.

Una vez abordado ese tema, pasaremos a estudiar los algoritmos evolutivos. En este tema nos detendremos de

forma importante, dado que estos algoritmos constituyen el eje fundamental del trabajo. Veremos ejemplos de su funcionamiento más sencillos que el diseño de la aeronave, para comprobar su validez.

A continuación, pasaremos a una sección particular de los algoritmos evolutivos, que es el diseño de la función de optimización y de los elementos avión a optimizar.

Continuaremos la aplicación de todo lo anterior a nuestro caso concreto, obteniendo como resultado el código MATLAB del programa en cuestión.

Tras tener operativo este código haremos algunos ejemplos para explicar más claramente el funcionamiento y estudiaremos los resultados.

Por último, cerraremos el trabajo con unas conclusiones al respecto.

Al final del trabajo se podrá consultar la bibliografía utilizada, así como el código completo utilizado en los anexos.

2 DISEÑO PRELIMINAR DE AERONAVES

Comenzamos explicando las dos ideas fundamentales detrás del diseño de aeronaves, basándonos en este apartado en la bibliografía consultada [2] y [9]

En primer lugar, el diseño de una aeronave es una tarea multidisciplinar, que implica la colaboración de distintas ramas. Ésta es la idea de ingeniería concurrente.

Además, nunca es un proceso directo. No podemos, a partir de unos requisitos, diseñar directamente el avión, sino que tendremos que hacer un diseño básico con poca información y, sobre dicho diseño, iterar repetidas veces hasta converger en el diseño final.

Estas dos ideas están representadas de forma gráfica en las siguientes imágenes cómicas:

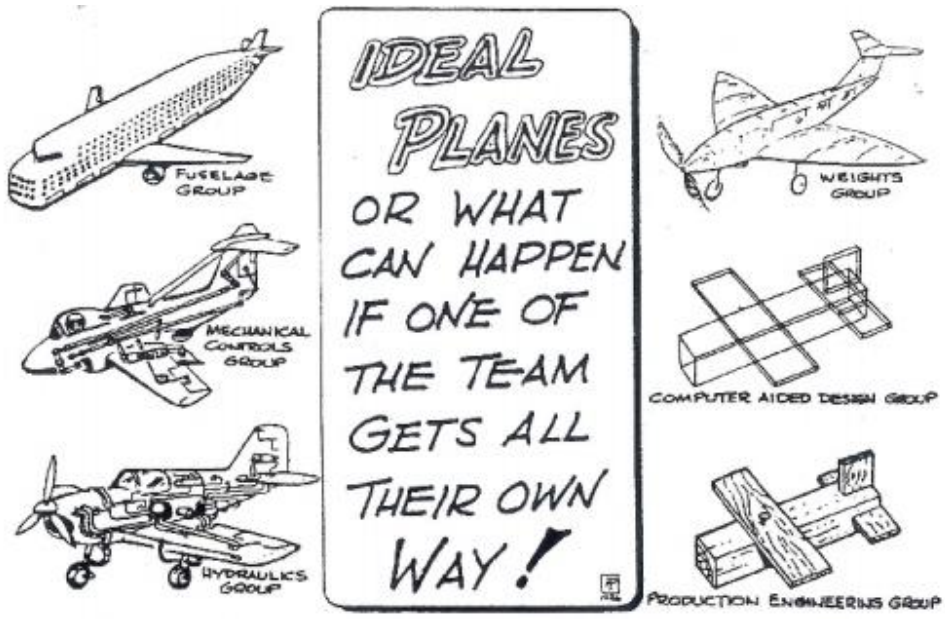


Ilustración 1: Sátira sobre la necesidad de la concurrencia en el diseño [9]

Ilustración 2: Sátira sobre el proceso iterativo del diseño [9]

Para abordar el problema del diseño hay que comenzar por definir los requerimientos de la aeronave y su misión. Estos requreimientos se encuentran en el Request for proposal (los nuevos diseños de aeronaves en general vienen como respuesta a un RFP lanzado por un determinado cliente o grupo de clientes), en la normativa (MIL, FAR, JAR, …) o a partir de un estudio de mercado que te dicte sus necesidades.

En general, el diseño se suele dividir en tres fases. Fase conceptual, fase preliminar y fase de detalle.

Las dos primeras fases se resumen bastante bien en el siguiente diagrama de flujo:

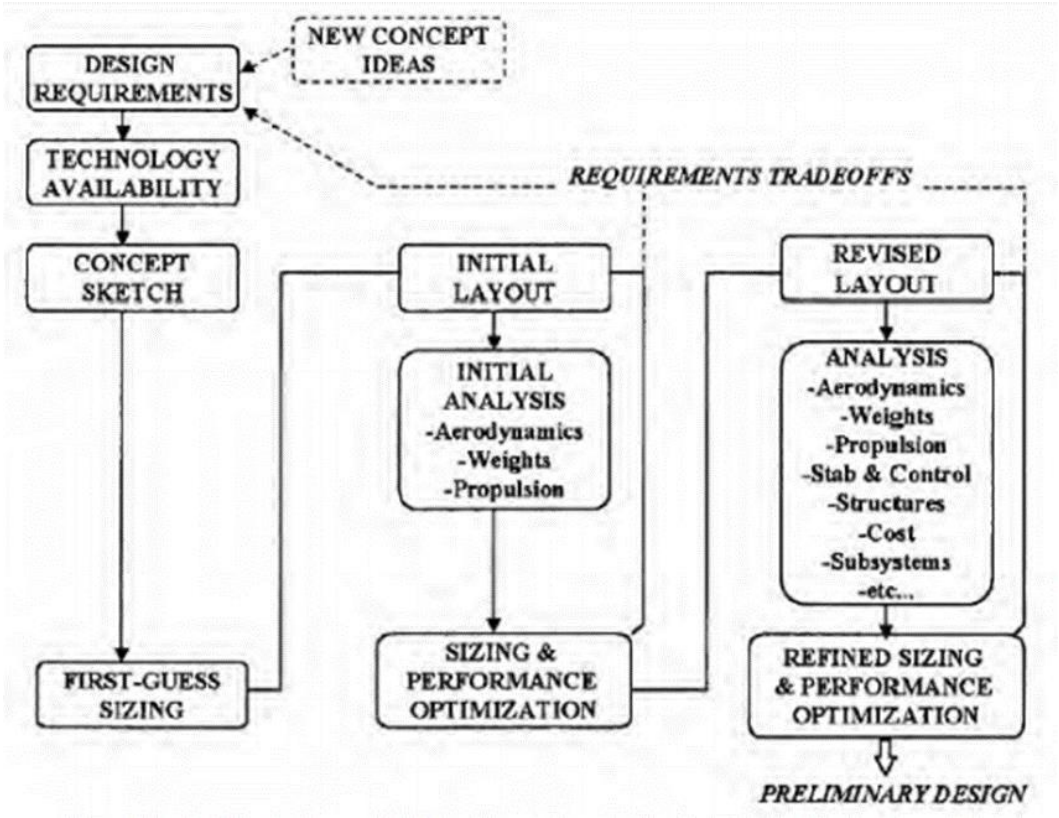


Ilustración 3: Flujo del diseño conceptual y preliminar [9]

En general, podemos decir que partimos de unos requerimientos. Conociendo cuál es la tecnología existente, se hace un *brainstorming* del que sale una serie de bocetos de diseños conceptuales.

De estos diseños, se hace una criba basándose en criterios cualitativos, quedándonos con una pequeña cantidad de opciones distintas (dos o tres generalmente).

En este momento, hacemos una primera estimación del tamaño de las aeronaves, y una primera distribución de los elementos. Con esta información hacemos un análisis más cuantitativo de los diseños, y nos quedamos con el que parezca más prometedor en este punto.

Apoyándonos en el análisis anterior, se revisa la distribución y el diseño de los elementos y se repite un análisis más profundo de cada área. A partir de este análisis se vuelve a revisar, y se itera hasta que lleguemos a un óptimo. En ese momento podemos determinar que tenemos un diseño preliminar.

La primera estimación de tamaño y distribución de elementos se hace en base a la idea de aviones semejantes. En general, las compañías fabricantes de aeronaves no toman el riesgo de hacer un diseño rompedor (que puede presentar problemas de todo tipo, desde fabricación hasta certificación, poniendo en peligro la estabilidad económica de la empresa fabricante), por lo que para estimar inicialmente simplemente se buscan en el mercado aviones con características similares y se supone, en primer análisis, que nuestro avión tendrá esas características. Este diseño, como hemos dicho, será refinado en análisis posteriores, pero sirve como punto de partida.

La última fase, el diseño de detalle, es la que más esfuerzo supone (en tiempo y dinero). Se deben diseñar todos los elementos de la aeronave, dejándola lista para comenzar la fabricación. El diseño global de la aeronave debe modificarse lo mínimo posible, pues cambios en el diseño a estas alturas provocarán grandes gastos económicos.

No vamos a entrar en mayor profundidad, pero es necesario ubicar la aplicación práctica del presente proyecto, así como distinguir lo que es de lo que no es.

Lo que sí, es una herramienta que facilita el trabajo, porque proporciona una alternativa al método de los aviones semejantes para dar un punto de partida en el diseño. Pero, como se ha explicado, sobre este primer diseño habrá que iterar para llegar a un diseño más maduro.

También es un mejor punto de partida, porque está basado en ideas cuantitativas, no en una simple copia que no entra en criterios profundos.

Asimismo, es un estudio importante sobre un método de optimización de funciones complejas que no se aplica frecuentemente.

Por último, también es un interesante ejemplo del uso de herramientas tecnológicas en áreas tan difíciles de automatizar como el diseño. Las estrategias evolutivas son un concepto parte del *machine learning* que tiene interesantes aplicaciones en todo tipo de áreas de diseño.

Es importante remarcar que el programa desarrollado **no** es una herramienta que diseña por sí sola el avión. Aún queda un tiempo para poder llegar al punto que marca la siguiente imagen.

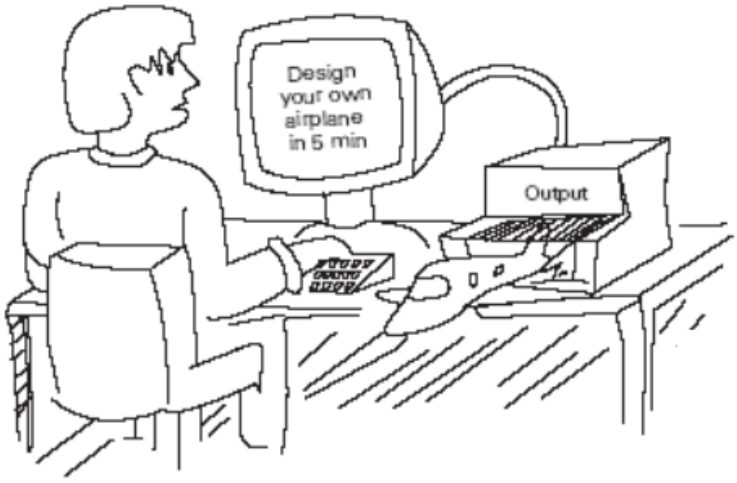


Ilustración 4: Sátira sobre la automatización del diseño [9]

3 ALGORITMOS EVOLUTIVOS

La ingeniería ha copiado a la naturaleza en múltiples aspectos tecnológicos. Es por ello que no debemos sorprendernos al encontrar que podemos optimizar problemas imitando el mayor proceso de optimización natural que ha existido nunca: La evolución.

La computación evolutiva, tal como estudiamos en [11] y [12] (de donde obtendremos la mayor parte de la información empleada), engloba los algoritmos de optimización inspirados por la evolución biológica, así como subcampos de inteligencia artificial y *soft computing*.

Hay una enorme variedad de técnicas de computación evolutiva, que se basan en múltiples aspectos de la biología (comportamiento de insectos, evolución gramatical, comportamiento del sistema inmunológico, …). De entre todas ellas, nos centramos en los algoritmos evolutivos.

Los algoritmos evolutivos son un subconjunto de algoritmos de optimización inspirados en la evolución biológica, con mecanismos como reproducción, mutación, mezcla de genes y selección natural.

En resumen, se caracterizan por la creación de una población de soluciones candidatas al problema que se trata de optimizar. Se valora la calidad de estas soluciones y, las que tengan mejores puntuaciones, tendrán mayor probabilidad de pasar sus genes a la siguiente generación. Esta siguiente generación será distinta de la anterior debido a la recombinación de los genes de los padres y a las mutaciones aleatorias, y tendrá, en general, mejor valoración respecto a la función a optimizar que la generación anterior.

Dentro de los algoritmos evolutivos también podemos distinguir distintos tipos de técnicas. La más común son los algoritmos genéticos, donde los genes están codificados en binario. Otras técnicas son programación genética, en la que la solución que se va a determinar es un programa de ordenador cuya calidad se determina por su capacidad para resolver un problema concreto, o la estrategia evolutiva. Es sobre esta última sobre la que estamos realizando el presente proyecto.

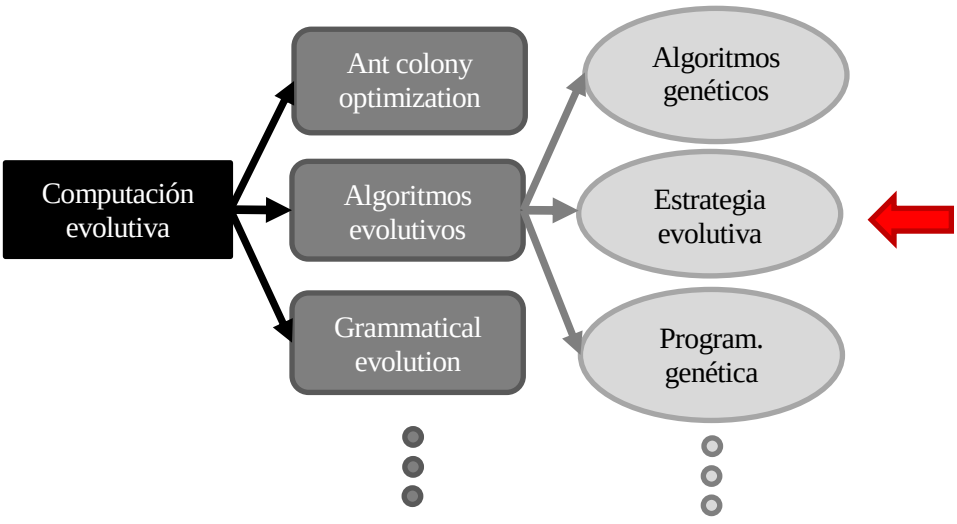


Ilustración 5: Ramas de la computación evolutiva

Hay muchas formas distintas de gestionar una optimización por estrategia evolutiva. En el presente capítulo nos vamos a centrar en la forma escogida, justificándola y aplicándola a los siguientes casos concretos de optimización de funciones con múltiples máximos y mínimos relativos:

- Caso 1: $f(x)=x\cdot\sin(x)$, $x \in [0, 100]$
- Caso 2: $f(x,y)=x\cdot y\cdot\cos(x\cdot y)$, x e $y \in [-5, 5]$

En el caso objetivo del trabajo completo, que es la optimización del diseño preliminar de una aeronave, el fondo será idéntico, pero la elección de los parámetros del algoritmo variará, dado que estos afectan de manera muy importante a la velocidad de convergencia y tienen que ser cuidadosamente escogidos según el problema concreto.

Es importante entender por qué estamos utilizando algoritmos evolutivos:

- Porque son un concepto de *machine learning* interesante de profundizar.
- Porque son versátiles y pueden aplicarse a casi cualquier problema de optimización.
- Porque son capaces de superar máximos y mínimos relativos, y esto es fundamental cuando desconocemos la forma de la función.
- Porque son rápidos cuando se trata de funciones que, siendo computacionalmente difícil encontrar su óptimo analíticamente, es muy sencillo encontrar el valor de la función para una solución concreta – problemas NP.

3.1. Resumen

El proceso completo puede ser observado en el siguiente diagrama de flujo:

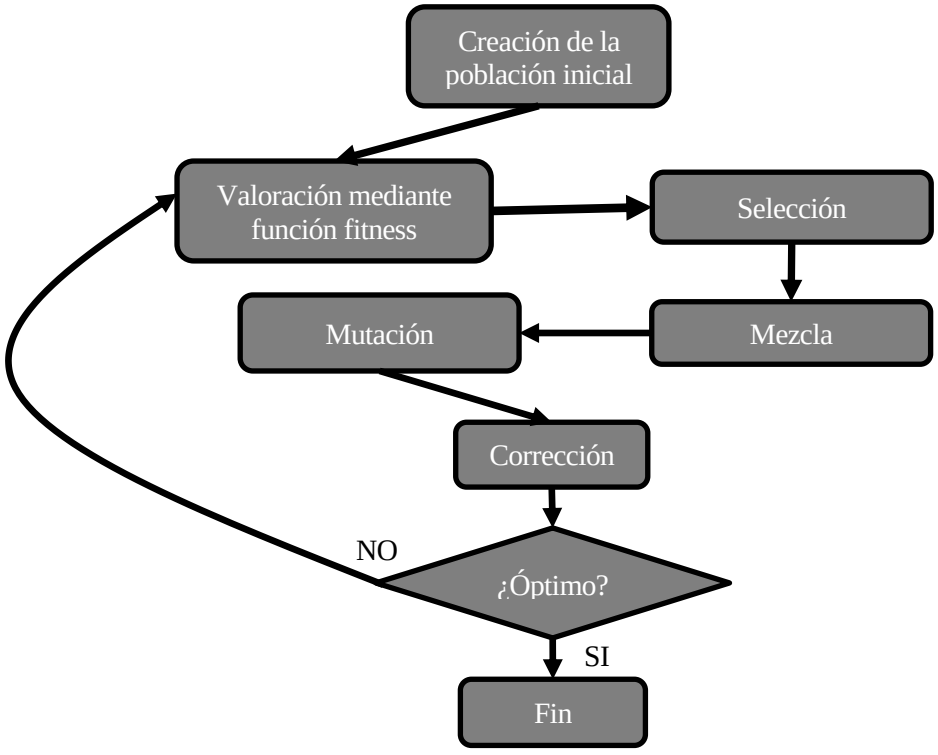


Ilustración 6: Diagrama de flujo explicativo de la estrategia evolutiva

A continuación, entraremos en detalle en cada uno de los pasos, pero en líneas generales es [11] [12]:

- Paso 1: Creación de una población de soluciones con parámetros al azar dentro de los límites establecidos.
- Paso 2: Valoración de la calidad de las soluciones.
- Paso 3: Creación de un conjunto donde cada solución esté repetida un número de veces tanto mayor cuanto mejor calidad tenga.
- Paso 4: Creación de una nueva población mediante mezcla de soluciones cogidas al azar del conjunto creado durante el paso tres.
- Paso 5: Mutación de esta nueva población.
- Paso 6: Corrección de la nueva población para que esté en los límites adecuados. También es habitual poner criterios de extinción, por los que se elimina una solución si no cumple alguna exigencia concreta.
- Paso 7: Valoración de si hemos llegado al óptimo, en cuyo caso detenemos el bucle. En caso contrario se repite desde el paso 2.

3.2. Creación de la población inicial

La población consiste en un conjunto de soluciones candidatas al problema que buscamos optimizar.

Cada solución candidata consiste en un **vector**. Este vector tiene un número de elementos denominados **genes**, que son números reales. Esta es la principal diferencia con los algoritmos genéticos, en los cuales estos genes serían *bits*.

En el caso 1 hay un único gen, el parámetro *x*, por lo que los vectores tienen dimensión 1 (son simples números reales).

En el caso 2 hay dos genes, los parámetros *x* e *y*. Por tanto, los vectores tendrán dimensión 2. En este caso concreto no nos importa el orden, dado que se están multiplicando y por ello el orden no afecta, pero en general hay que establecer un orden determinado. Nosotros diremos que el primer elemento es *x* mientras que el segundo es *y*, [x, y].

Nuestro espacio de búsqueda son los números reales dentro de las restricciones impuestas, por lo que la población inicial se generará de forma aleatoria cumpliendo con que se encuentren en los intervalos especificados - recordamos que para el caso 1 el intervalo es [0, 100], y para el caso 2 es [-5, 5]-.

El número de elementos de la población es un importante parámetro a escoger en este tipo de algoritmos porque se produce el siguiente efecto. Cuanto mayor sea el número de elementos en la población, más exploramos el espacio de búsqueda en cada generación y por tanto tardamos menos generaciones en llegar al óptimo. Sin embargo, la computación de cada generación será más costosa y, por consiguiente, más lenta. En el caso opuesto, cuantos menos elementos tenga una población, más rápido se computa cada generación, pero más generaciones tardaremos en llegar a la solución. La elección de este parámetro se realiza durante el tuneado del algoritmo.

A continuación, mostramos, como caso ilustrativo, un ejemplo gráfico de lo que sería una población inicial de 10 elementos. Hemos representado con un número decimal, pero podrían tener cualquier número de decimales (cualquier número real dentro de los intervalos es aceptable).

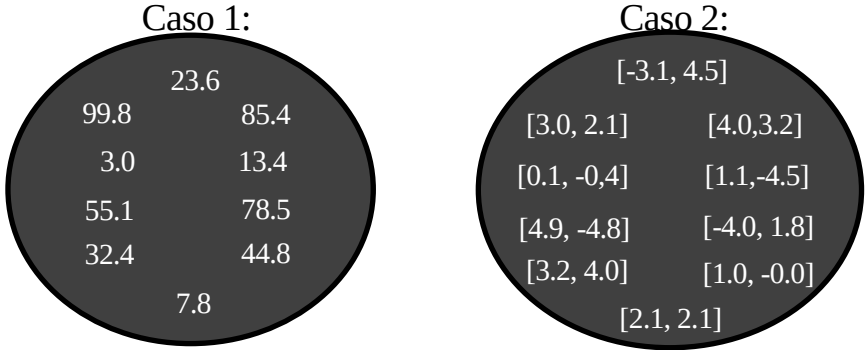


Ilustración 7: Ejemplos de población

Para el caso del problema del diseño de aeronave, el número de genes será mayor, el tamaño de la población será también mucho mayor y habrá unos límites concretos para cada gen. Veremos ese problema concreto en su apartado correspondiente.

3.3. Valoración mediante función *fitness*

Una vez tenemos una población de soluciones candidatas, tenemos que asignar un valor numérico a cómo de buenas son dichas soluciones.

Para ello necesitamos una función en la que las entradas sean **todos** los genes de las soluciones y la salida sea **un número real** (*fitness*) que nos indique cómo de buena es la solución introducida. Si algún gen no forma parte de la función de *fitness*, este gen no será optimizado y será siempre aleatorio, por ello es importante que todos los genes tengan algún efecto en el valor del *fitness*.

En los casos de ejemplo esta función es inmediata. La función *fitness* serán las funciones trigonométricas que estamos tratando de optimizar.

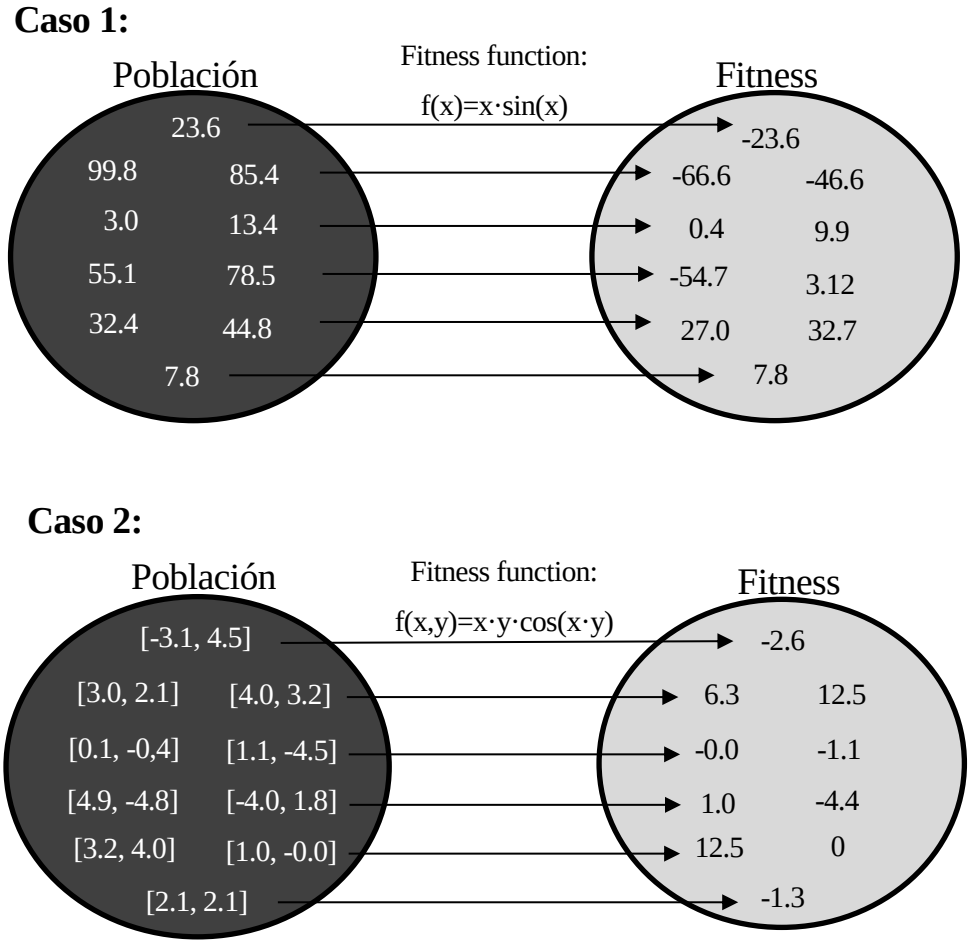


Ilustración 8: Ejemplos de *fitness function*

Destacamos varias ideas. En primer lugar, no importa cuántos genes tiene la entrada, la salida es un único número real. Este número real valora cómo de buena es la solución en el problema que estamos optimizando.

Como ejemplo, podemos ver que para el caso 2, las soluciones [3.2, 4] y [4, 3.2] son las mejores de toda la población, dado que llegan a un *fitness* de 12.5. En el extremo opuesto, la peor de todas es la solución [-3.1, 4.5], que tiene un *fitness* de -2.6.

En estos casos particulares la función *fitness* es muy sencilla. En el caso de la optimización del diseño preliminar de una aeronave, la función será uno de los elementos más complicados de decidir por dos razones. En primer lugar, porque la pregunta '¿Cuán bueno es este diseño?' puede ser muy subjetiva y difícil de valorar y, en segundo lugar, porque las valoraciones numéricas que se puedan hacer son computacionalmente muy exigentes.

3.4. Selección

Llegados a este punto, tenemos una población de soluciones candidatas y un *fitness* de cada solución. El siguiente paso es seleccionar soluciones para que sean padres de la siguiente generación.

La característica fundamental es que cuanto mayor *fitness* tenga una solución, mayor probabilidad debe tener para ser escogida para ser padre.

Hay muchas formas de llevar a cabo la selección. La más sencilla es truncando la población, de forma que se elimine un porcentaje de las peores soluciones (por ejemplo, eliminar la mitad peor de las soluciones) y escoger al azar entre las supervivientes. Sin embargo, esta técnica no discrimina entre la calidad de las soluciones supervivientes y elimina completamente la posibilidad de estudiar soluciones en zonas menos prometedoras (en otras palabras, limita la exploración).

Para solventar este problema, se va a seguir la técnica de la creación de una *mating pool*. Esto consiste en lo siguiente:

- Se asigna un número natural, *mating pool number*, a cada solución. o describimos más adelante, pero las principales características son que es mayor cuanto mayor sea el *fitness* y que se debe obtener a partir del *fitness* de la solución relativo al *fitness* del resto de soluciones y del número de generación.
- Se crea una *mating pool*, que es un conjunto en el que cada solución estará repetida tantas veces como indique su *mating pool number*.
- Finalmente, la selección se hace de forma aleatoria entre los elementos de la *mating pool*, por lo que habrá más probabilidades de escoger los elementos que estén más veces repetidos. Los elementos más veces repetidos son aquellos con mayor *mating pool number*, debido a que tienen mayor *fitness*, gracias a que es una mejor solución.

Respecto a la obtención del *mating pool number* hacemos lo siguiente:

- Sumamos el valor absoluto de la *fitness* más negativa, más uno. De este modo tendremos valores positivos mayores o iguales a uno. Si ninguna *fitness* es negativa, dividimos todas entre la menor para, del mismo modo, tener valores mayores o iguales a uno.
- Elevamos los valores absolutos de los números resultantes a $1 + \frac{generacion}{divisor}$. De este modo amplificamos las diferencias conforme pasan las generaciones. Esto es importante debido a que, en generaciones avanzadas, todos los *fitness* son muy parecidos, y eso llevaría a que la diferencia de probabilidad entre escoger las mejores soluciones o las peores sea muy pequeña. Se necesita, por lo tanto, amplificar dichas diferencias. El parámetro 'divisor' se escoge durante el tuneado del algoritmo, para regular el ritmo al que se amplifican las diferencias (cuanto mayor sea, menos se amplifican y más se explora el espacio de búsqueda, pero más lento es el algoritmo).
- Multiplicamos el resultado por 100 y redondeamos al número entero más cercano.
- Para evitar problemas cuando estamos en generaciones muy altas, verificamos si *Matlab* reconoce algún número como infinito. En dicho caso, los infinitos se vuelven 1, el resto de números se vuelven 0 e igualamos divisor a generación para evitar infinitos en las próximas generaciones.
- Para evitar *mating pool* excesivamente grandes, dividimos los *mating pool number* entre 3 y redondeamos, tantas veces como sea necesario para que la suma de todos los mating pool number sea menor que un cierto número de veces la población.

La forma más sencilla de ver este proceso es mediante un ejemplo. Hemos escogido los *fitness* del caso 2. En cada columna vemos cada una de las operaciones. Lo hacemos con un valor de divisor 100 y un valor de generación 50. Cuanto mayor sea el ratio generación/divisor, más se amplificarán las diferencias de los *mating*

pool number. Ignoramos el penúltimo paso, dado que no encontramos infinitos. En el último paso reduciremos hasta que la suma de *mating pool number* sea menor de 10000.

Fitness	Suma	Elevado	Mult. + redondeo	Reducción
-2.6	2,8	4,685	469	156
6.3	11,7	40,020	4002	1334
12.5	17,9	75,732	7573	2524
-0.0	5,4	12,548	1255	418
-1.1	4,3	8,916	892	297
1.0	6,4	16,190	1619	540
-4.4	1	1	100	33
12.5	17,9	75,732	7573	2524
0	5,4	12,548	1255	418
-1.3	4,1	8,301	830	277

Tabla 1: Ejemplo de proceso para obtener el *mating pool number*.

3.5. Mezcla

Si recapitulamos, vemos que hemos creado una población, hemos evaluado su *fitness* y con dicho *fitness* hemos creado una *mating pool* en la que cada elemento está más veces repetido cuanto mejor solución sea.

Ahora llega el momento de crear una nueva población, mezclando los valores de elementos de la *mating pool*.

Al contrario que en la biología natural, en este caso podemos escoger libremente el número de padres que tendrá cada elemento de la nueva población (cualquier número entre 1 y el número total de elementos de la población). Sin embargo, lo más habitual, y lo que haremos nosotros, es imitar a la naturaleza y escoger dos padres para cada nuevo elemento.

Por lo tanto, se escogen al azar dos individuos de la *mating pool* y se mezclan. Como se ha explicado previamente, dado que los mejores elementos están más veces repetidos en la *mating pool*, tendrán más posibilidades de ser escogidos y por tanto pasar sus genes, sin que se pierda por completo la exploración de regiones menos prometedoras (al menos en las primeras generaciones, cuando las diferencias en los *mating pool number* no son tan grandes)

Hay distintas técnicas para hacer este mezclado, en concreto suelen dividir en cuatro clases: Convencionales, aritméticas, *direction-based*, y estocásticas. Van desde técnicas sencillas, como escoger parámetros de un padre o de otro al azar, a técnicas complejas, en las que se introduce el gradiente aproximado de la función que se está optimizando en el algoritmo de mezclado.

La técnica que nos resulta más interesante para nuestro caso es una mezcla aritmética, en la que usaremos la media de los padres. Nos resulta apropiado por su sencillez respecto a mezclas estocásticas o *direction-based*, mientras que mantiene una mayor capacidad de exploración que las técnicas convencionales.

Además, en este proyecto vamos a ir un paso mas allá y cada parámetro del elemento hijo estará distribuido como una gaussiana, caracterizada por una media que será la media de los parámetros de los padres y una desviación típica. El objetivo de esta caracterización es que la mezcla repetida de los mismos padres dé como resultados hijos parecidos pero distintos, con lo que mejoramos la exploración.

Para evitar que el hijo resulte muy distinto respecto a la media de los padres, la desviación típica considerada será baja, menor de un 5% del valor de la media. Este valor se obtiene de la experiencia propia en la programación del algoritmo evolutivo, dado que valores más grandes dificultan la convergencia cuando tenemos un criterio de finalización muy exigente.

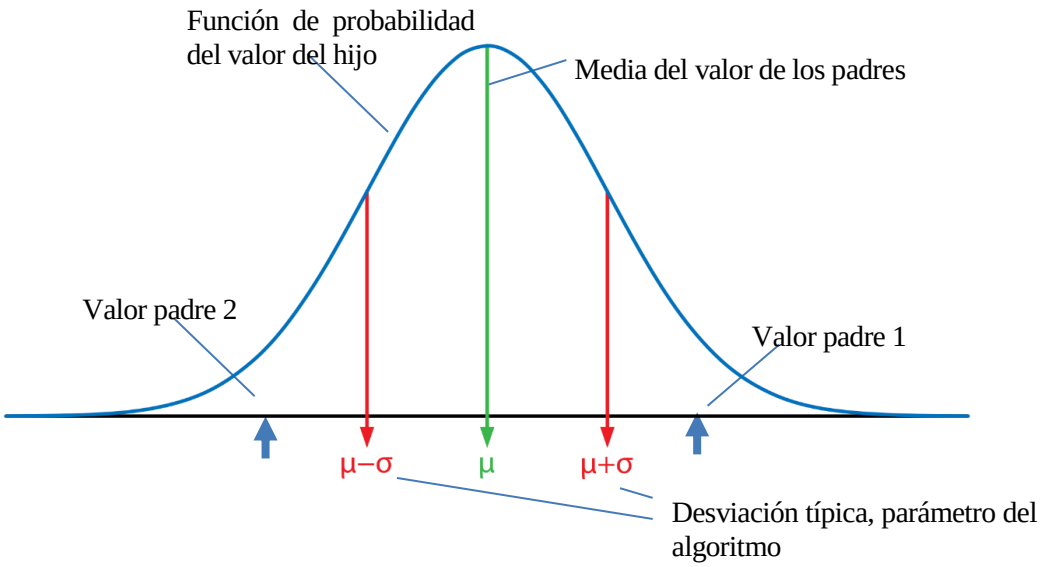


Ilustración 9: Mezcla mediante función de distribución de probabilidad gaussiana

La mezcla se hará tantas veces como grande sea la población (si es una población de 1000 elementos, haremos 1000 mezclas para obtener 1000 nuevos elementos).

Pongamos un ejemplo de una mezcla concreta:

Padre 1: [3.0, 2.1]; Padre 2: [3.2, 4]

Primer parámetro: Media → 3.1; Segundo parámetro: Media → 3.05

Desviación típica → 0.15; Parámetro del algoritmo

Hijo: [G~(3.1; 0.15), G~(3.05; 0.15)]

Hijo: [3.05, 2.9]; Es un ejemplo de resultado aleatorio, si se volviera a calcular con los mismos parámetros podrían dar un resultado distinto, pero siempre de acuerdo a la gaussiana establecida.

3.6. Mutación

La última modificación de la población de acuerdo a la estrategia evolutiva es crear mutaciones al azar. Las mutaciones son necesarias para no dejar nunca de explorar el espacio de búsqueda y tienen diversas características y requerimientos.

Las características de estas mutaciones son dos. En primer lugar, son muy poco habituales (la mayor parte de la población no sufrirá mutaciones, y las que lo sufran lo normal será que sea solo uno de sus genes). En segundo lugar, producen cambios muy severos.

Para que un operador de mutación sea adecuado, tiene que cumplir tres requerimientos. El primero es el alcance, por el que cualquier punto del espacio de búsqueda tiene que tener una probabilidad de ser alcanzado mediante la mutación de un elemento. En segundo lugar, la imparcialidad, que determina que el operador no debe introducir un sesgo en la dirección de la mutación. En tercer lugar, la escalabilidad, que nos indica la necesidad de ofrecer al operador algún grado de libertad por el que la severidad sea adaptable.

Hay varias formas típicas de llevar a cabo lo descrito anteriormente, siendo la más popular para representaciones contínuas como la nuestra el sustituir el parámetro por otro distribuido como una función gaussiana de media el valor de dicho parámetro y una desviación típica determinada.

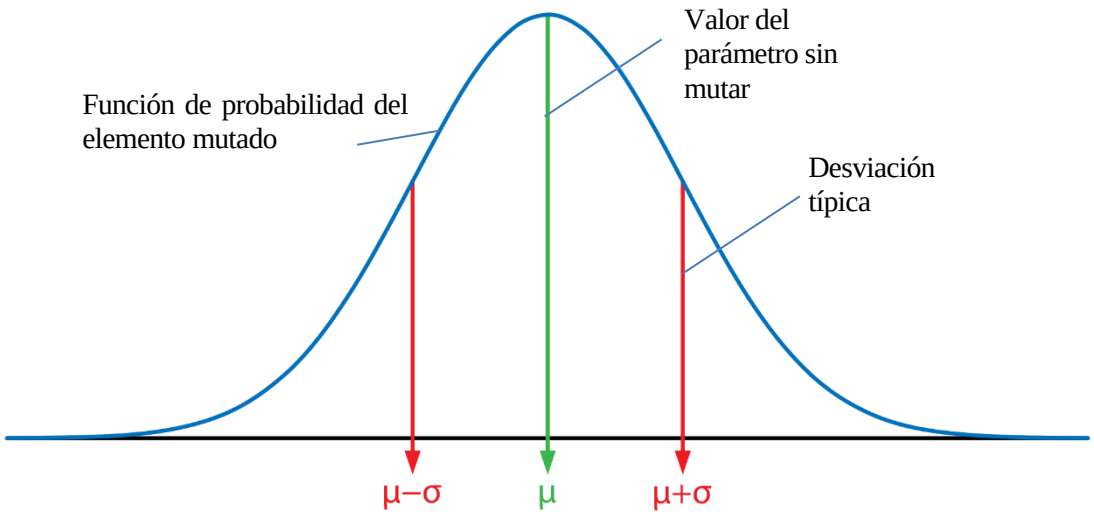


Ilustración 10: Mutación mediante función de probabilidad gaussiana

Tomamos como probabilidad de mutación un 5%. La severidad de la mutación (la severidad es mayor cuanto mayor sea la desviación típica aplicada), se ajusta a lo largo del proceso de optimización de acuerdo al número de generaciones y al éxito de cada generación, y hay distintas técnicas para ese ajuste. El punto inicial debe ser de una desviación típica grande, mayor del 100% del valor del parámetro, y este valor se irá adaptando a lo largo de las generaciones. Este valor de 100% lo tomamos de la experiencia con las funciones tratadas, dado que las referencias sobre el tema se centran en la adaptación de este valor, dándole poca importancia al valor inicial.

Nosotros utilizaremos una estrategia de control adaptativo basándonos en la regla de Rechenberg [11], aunque con modificaciones (por la experiencia en las funciones con las que tratamos). La idea es incrementar la tasa de mutación cuando sea posible para acelerar la búsqueda. Sin embargo, es razonable disminuir la tasa de mutación si la búsqueda se queda atascada. Por lo tanto, mediante este algoritmo de adaptación mediremos el éxito de la evolución y, si tiene un éxito mayor de determinada cantidad, incrementaremos la tasa de mutación. Si tiene menor éxito que una determinada cantidad, se reducirá. En otro caso, se mantiene. Con esto se pretende mantenerse en la llamada ventana de evolución, que garantiza tasas de progreso óptimas.

La variación de la tasa de mutación se hace en incrementos o decrementos de 5%, y las barreras seleccionadas son si el éxito es menor, igual, o mayor a 1/50 (en cuyo caso se multiplica por 0.95, 1 y 1.05 respectivamente).

Se debe poner un límite inferior al valor de mutación, para que ésta no desaparezca nunca. En el caso que estamos tratando, seleccionamos un 0.0001.

Pongamos un ejemplo de una mutación concreta:

Probabilidad de mutación: 0.05; Desviación típica: 5 → Aunque la desviación típica va cambiando a lo largo de las generaciones, durante una generación concreta es constante.

Elemento: [3.05, 2.7]

Primer parámetro: Se prueba suerte y resulta que no hay mutación.

Segundo parámetro: Ocurre mutación.

Elemento mutado: [3.05, G~(2.7, 5)] → [3.05, -1.3]

3.7. Corrección

El último paso para finalizar la nueva población es revisar que todos los parámetros de los elementos están dentro de los límites aceptables, dado que la mezcla y mutación pueden haberlos sacado de los intervalos especificados inicialmente.

Si algún elemento no se encuentra en los límites hay varias formas de lidiar con ello. Se puede mantener el

elemento, pero asignarle un valor muy bajo a su *fitness* para que no pase los genes a la siguiente generación. Otra opción más sencilla es borrar el parámetro y darle un valor al azar o un valor predeterminado para estos casos. También se puede mutar el parámetro hasta que entre dentro de los límites aceptables.

En nuestro caso, la opción tomada es hacer que el valor del parámetro sea igual al límite más cercano.

Por ejemplo, en el caso 2 donde los límites eran [-5, 5] para ambos parámetros, se haría la siguiente transformación:

Elemento: [-6, 1.3] → Elemento corregido: [-5, 1.3]

Otra operación que se suele realizar en este momento es comprobar criterios de extinción. Estos criterios son tales que, si no se cumplen, se le da un valor muy bajo al *fitness* para que el elemento se extinga. En los casos de optimización de funciones trigonométricas que estamos tratando no haremos uso de esto, pero sí que lo haremos en el caso de optimización del diseño de una aeronave.

3.8. Salida del bucle

El algoritmo de optimización se debe detener en algún momento. En los casos de ejemplo, dado que conocemos de antemano cuál es el óptimo, detendremos el algoritmo cuando la mejor solución encontrada sea un cierto porcentaje del óptimo (un porcentaje muy alto, sobre un 99.99%).

Se debe mantener también un límite en tiempo y número de generaciones para evitar que fallos en el código o en el tuneado del algoritmo nos lleven a no alcanzar nunca el criterio de finalización.

Para el problema de este proyecto, el de optimización del diseño preliminar, no sabemos de antemano cuál es la solución, por lo que no podemos poner ese criterio.

Para detener el algoritmo en ese caso nos basaremos en la evolución de la mejor respuesta. En el momento en que no consigamos mejora durante un amplio número de generaciones, a pesar de los ajustes en la mutación, detendremos el algoritmo.

3.9. Tuneado o determinación de parámetros

Ya conocemos el algoritmo completo. En concreto, resumimos los siguientes elementos a determinar:

- Tamaño de población: Cuanto mayor, menos generaciones en llegar al óptimo, pero más lenta cada generación y viceversa.
- Divisor: Cuanto menor sea el divisor, más rápidamente se amplificarán las diferencias en los *mating pool number*. Amplificar esta diferencia ayuda a escoger las mejores soluciones, pero también nos impide explorar el espacio de búsqueda y nos puede llevar a atascarnos en máximos locales.
- Valor inicial de desviación típica en la mutación y probabilidad de mutación: Cuanto mayores, más exploración del espacio de búsqueda, pero convergencia más lenta. Es necesario que sean grandes para evitar quedarnos atascados en máximos locales. La desviación típica varía conforme pasan las generaciones, por lo que la elección del valor inicial no es tan trascendental, como sí lo es la elección de la probabilidad de mutación.
- Criterio de salida de bucle: En el caso básico, hablamos del porcentaje del óptimo que queremos alcanzar. En el caso de la optimización del diseño de aeronaves, hablamos de cuántas generaciones tienen que pasar sin que el máximo mejore. En ambos casos, cuanto más exigente sea el criterio, más tardamos en converger, pero mejor óptimo obtendremos.

Hay muchas maneras de determinar los parámetros. Desde formas complejas que utilizan algoritmos evolutivos para optimizar los propios parámetros del algoritmo (algoritmos metagenéticos) hasta ideas sencillas como consultar expertos.

Nosotros simplificaremos este punto. Primero pensaremos razonadamente el intervalo en que debería moverse cada parámetro, y luego haremos pruebas sencillas en las que variemos solamente dicho parámetro.

Los intervalos para los casos sencillos planteados serían los siguientes (para la optimización del algoritmo completo lo veremos más adelante):

- Población: Dado que ambas funciones son muy sencillas, la computación de cada generación será muy rápida. Esto nos va a permitir poblaciones grandes. Además, dado que son funciones con muchos máximos relativos necesitamos una gran población para que la exploración del espacio de búsqueda sea contundente. Hablamos del orden desde decenas a cientos de miles de elementos.

- Divisor: Si sabemos que la optimización va a dirigirse rápido hacia la zona del máximo absoluto y por tanto tenemos menos peligro de perdernos en un máximo local (porque tengamos una población grande o una función con pocos máximos locales), se puede seleccionar un número más pequeño (20-50). Por el contrario, si no conocemos bien la función o tenemos poblaciones pequeñas que tardan más en explorar el espacio de búsqueda, debe ser un número grande (300-1000). En nuestro caso, tener una gran población nos permitirá explorar el espacio de búsqueda muy rápido, por lo que podemos seleccionar un divisor pequeño, en torno a 50.

- Valores iniciales de desviación típica en la mutación y probabilidad de mutación: En algoritmos de este tipo, generalmente se considera que menos de un 5% de parámetros deben sufrir mutación. Respecto a la desviación típica, debe estar relacionada con los gradientes de la función *fitness* en el espacio de búsqueda. Cuanto mayores sean, menor será la desviación típica. En nuestro caso utilizaremos como valor inicial 5, sin tunearlo, dado que es adaptativo.

- Criterio de salida de bucle: Dado que la computación es muy sencilla, y la exploración es muy extensa gracias a la enorme población, podemos poner un criterio de máximo muy exigente, por ejemplo, llegar al 99.9999% del valor del óptimo de la función.

Ya que conocemos los rangos, vamos a hacer distintas pruebas.

El *output* de las pruebas que vemos en los próximos apartados es el tiempo en segundos que tarda en salir del bucle, ejecutando el programa en un ordenador con un procesador Intel®Core™ i7-5500U, 2.4GHz, 2 núcleos, 4 procesadores lógicos. Cada prueba la repetiremos 10 veces, para observar los distintos valores que obtenemos.

Respecto a los límites del caso 2, son modificados a [-6, 5] para *x* e *y*. Esto es debido a que en el intervalo originalmente especificado el máximo se encuentra en [x, y] = [-5, -5], en uno de los bordes del espacio de búsqueda, lo que facilitaba enormemente encontrar el máximo y la convergencia era siempre muy rápida. Eso nos demuestra la calidad del algoritmo también para máximos que se encuentran en los límites del espacio de búsqueda pero, dado que daba poca información para el tuneado, hacemos la modificación explicada.

3.9.1 Tuneado de la población.

Divisor	Prob. Mutación	Criterio de salida
50	5%	99.999%

- Población 10.000:

Caso 1	28.44	28.88	29.41	28.78	0.22	0.22	27.60	0.22	29.74	0.22
Caso 2	53.68	60.74	55.62	25.09	0.54	3.99	59.88	54.09	16.03	71.94

- Población 100.000:

Caso 1	1.67	1.67	1.62	1.60	1.62	1.63	1.60	1.62	1.63	1.61
Caso 2	39.16	5.47	46.49	77.06	5.27	5.12	26.93	5.40	5.34	27.44

- Población 1.000.000:

Caso 1	15.73	16.74	16.08	16.22	17.25	16.04	16.28	16.51	16.16	16.24
Caso 2	49.38	48.97	49.90	50.20	49.64	50.19	50.21	50.74	50.06	49.9

Tabla 2: Tuneado del parámetro población

Estas pruebas nos dan resultados, como se puede comprobar, muy dispares. Ello nos invita a tener una reflexión más profunda en este parámetro.

Después de numerosas pruebas llegamos a la siguiente conclusión. Para una función *fitness* con una computación tan sencilla como la que presentamos, y un criterio de salida que no sea muy exigente, una optimización por algoritmo evolutivo no es lo más eficiente. Esta conclusión se extrae de que los menores tiempos de cálculo se obtienen para valores de población del orden de 100.000. En ese orden, ya no existe evolución, puesto que la salida se hace en la primera o segunda generación, y el algoritmo se desvirtúa, por lo tanto, en una 'fuerza bruta'.

Dado que nuestro interés en estos ejemplos es ver la evolución de los elementos, escogeremos una población de 10.000 elementos para ambos casos, que es una cantidad que permite ver una convergencia de la solución a lo largo de las generaciones.

3.9.2 Tuneado del divisor.

Población	Prob. Mutación	Criterio de salida
10.000	5%	99.999%

- Divisor 10

Caso 1	15.40	0.21	15.56	0.20	15.68	15.38	0.37	0.20	0.21	15.59
Caso 2	9.81	74.57	21.94	3.11	74.9	77.29	76.5	76.68	0.58	77.68

- Divisor 50

Caso 1	0.20	29.48	30.2	0.20	0.20	0.40	29.45	0.20	0.19	0.20
Caso 2	60.29	93.11	56.60	76.32	54.20	65.70	12.59	12.72	74.49	92.9

- Divisor 100

Caso 1	0.20	0.19	0.19	37.50	38.11	0.20	37.97	38.19	38.51	37.80
Caso 2	72.16	63.58	67.59	3.23	4.88	66.54	71.20	0.53	69.56	0.52

Tabla 3: Tuneado del parámetro divisor

En el caso 1, vemos que las respuestas se dividen en dos grupos, uno en el que convergen extremadamente rápido (orden de décimas de segundo) y otro en el que convergen en un tiempo corto (orden de decenas de segundos). El primer grupo es debido a que una o varias de las soluciones en la población inicial cayeron muy cerca del óptimo (gracias a un tamaño de población grande) y por tanto apenas tarda 2 o 3 generaciones en converger. En este grupo no hay evolución, y por tanto no debemos guiarnos de esos resultados para determinar el valor del divisor, dado que éste no juega ningún papel. Guiándonos de los resultados del segundo grupo, el de convergencia más lenta, que realmente sí ha evolucionado, comprobamos que un divisor de valor 10 da mejores resultados. Esto tiene sentido, puesto que un divisor pequeño lleva a una convergencia más rápida, sacrificando exploración. Sin embargo, con 10.000 elementos distribuidos entre 1 y 100, sabemos que algún elemento de la población inicial se encontrará en la zona del óptimo y la pérdida de exploración no nos da problemas importantes.

Respecto al caso 2, vemos mucha mayor variabilidad, sin poder llegar a conclusiones claras. Tomamos por tanto el mismo divisor que para el caso 1.

3.9.3 Tuneado de la probabilidad de mutación

Población	Divisor	Criterio de salida
10.000	10	99.999%

- Probabilidad 1%

Caso 1	0.2	15.9	30.9	38.2	59.7	30.0	0.16	0.17	0.17	0.17
Caso 2	26.0	31.1	22.9	30.5	31.6	25.9	26.4	30.4	28.3	30.4

- Probabilidad 5%

Caso 1	0.2	14.9	0.2	0.2	0.2	0.2	27.7	36.2	0.2	0.2
Caso 2	74.8	13.9	4.0	0.6	107.1	133.7	16.5	143.9	152.8	91.2

- Probabilidad 10%

Caso 1	26.12	0.2	0.2	0.2	0.5	0.2	36.6	44.8	0.2	30.9
Caso 2	3.1	4.0	73.3	108.3	17.8	127.3	143.7	147.2	56.0	157.0

Tabla 4: Tuneado del parámetro probabilidad de mutación

No encontramos una diferencia apreciable, utilizaremos el 5% de mutación.

3.9.4 Tuneado del criterio de salida del bucle

Población	Divisor	Prob. De mutación
10.000	10	5%

- Criterio de salida 99.9%

Caso 1	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2
Caso 2	6.1	1.7	2.1	0.6	3.8	0.6	0.6	11.4	0.6	0.6

- Criterio de salida 99.999%

Caso 1	0.2	0.2	0.2	15.1	0.2	28.5	36.6	0.2	42.8	14.2
Caso 2	73.0	110.2	2.2	134.3	146.8	5.8	160.3	167.2	37.4	180.2

- Criterio de salida 99.99999%

Caso 1	15.4	29.3	0.3	37.1	58.9	76.3	71.9	70.5	0.16	72.1
Caso 2	73.1	105.2	132.8	159.9	172.7	173.1	176.2	169.4	183.1	163.2

Tabla 5: Tuneado del criterio de salida del bucle

En la determinación de este parámetro, es evidente que cuanto menos exigente sea el criterio, más rápida será la salida del bucle. Por lo tanto, el criterio para escoger el parámetro no será el que proporcione una salida más rápida, sino el más exigente del que resulte un tiempo aceptable.

Para unos casos tan sencillos, el criterio más exigente sigue sin ser demasiado lento (en general menos de 3 minutos para el caso 2, que es el más complejo) y, por consiguiente, lo utilizaremos.

3.9.5 Resultados

Finalmente hacemos una revisión de lo obtenido en este apartado del trabajo.

Se ha descrito en profundidad la estrategia evolutiva, explicando el marco en el que se encuentra y se han planteado distintas técnicas y criterios, explicando y justificando los que han sido escogidos. Además, se han hecho pruebas para tunear la optimización del algoritmo, llegando al siguiente resultado:

Población	Divisor	Prob. De mutación	Criterio parada
10.000	10	5%	99.99999%

Tabla 6: Parámetros escogidos

Con esos parámetros hacemos pruebas más extensas, 30 para cada tipo, obteniendo los siguientes resultados:

Caso 1			Caso 2		
0.2429	16.0069	29.7081	80.1706	113.7799	151.6411
39.2932	46.1757	70.1521	163.3122	175.0509	10.2149
87.8782	84.7972	72.1522	184.0973	15.1143	190.3047
74.0794	101.8346	87.9717	156.5536	186.7007	199.8173
0.1752	26.5759	77.1159	197.4075	175.2310	179.0977
84.3328	73.7255	86.0229	175.4200	184.1144	144.6407
32.2946	81.5534	91.2465	201.6298	188.8469	190.8310
0.1739	67.5240	56.1977	198.8686	201.4654	197.8100
77.2389	87.9547	87.9547	202.1819	118.6956	176.8760
65.9429	65.7867	76.3482	161.5701	148.6814	148.6814

Tabla 7: Resultados exhaustivos

Descartamos el 10% superior e inferior. Con los 24 resultados restantes, vemos cuál es el resultado estadístico caracterizado por la media y la desviación típica:

Caso 1		Caso 2	
Media	DT	Media	DT
64.4354	21.3513	173.9161	25.0964

Tabla 8: Resultados estadísticos

Ya tenemos el algoritmo completamente definido. Por último, vamos a optimizar otra función distinta pero similar, manteniendo los parámetros para comprobar que también se comporta de forma adecuada.

$f(x,y)=(x+y)\cdot\sin(x)\cdot\cos(y)$, x e y $\in$ [-5, 5]

Utilizaremos los parámetros del caso 2, que es el más similar. Haciendo el mismo análisis que acabamos de hacer, obtenemos el siguiente resultado:

f(x,y)=(x+y)*sin(x)*cos(y)									
68.1491	21.0818	128.0851	147.7921	208.5346	158.9136	251.9074	91.7409	270.2390	316.2542
364.7034	391.6010	128.3102	458.6840	489.7237	532.1442	583.7206	623.0997	151.6579	671.4385
176.9019	365.8960	36738482	719.2218	209.9271	764.7607	44.2276	170.3999	174.3061	798.7020

Media	DT
309.7429	173.4986

Tabla 9: Resultados función 3

También resulta muy ilustrativo ver cómo mejora la solución a lo largo de las generaciones. En la siguiente gráfica se observa un ejemplo genérico de cómo tienen lugar estas mejoras, donde podemos ver un perfil escalonado, en el que la mayor parte de mejoras se dan en saltos entre los cuales hay una gran cantidad de generaciones con pocas mejoras. Destacamos que el eje X está en escala logarítmica

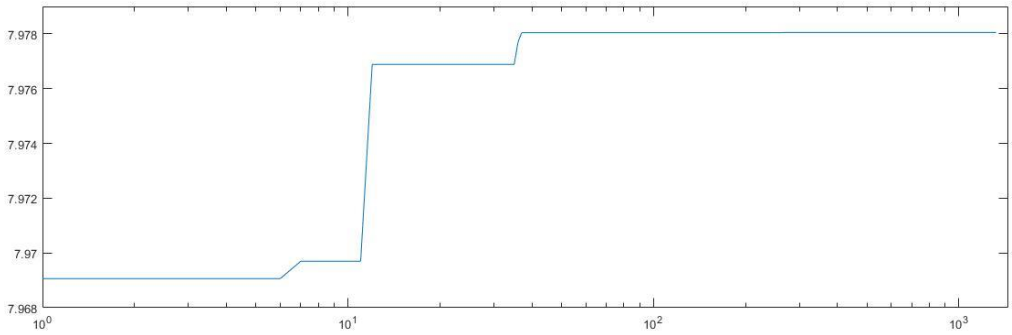


Ilustración 11: Mejora del fitness a lo largo de las generaciones.

Llegamos a las siguientes conclusiones:

- Aunque no hay una demostración matemática que compruebe que el funcionamiento de una optimización por estrategia evolutiva siempre obtiene una solución, llegamos a la conclusión de que en los casos prácticos el algoritmo funciona. Llega a soluciones muy exactas en un tiempo razonable. Por tanto, podemos pasar al siguiente paso que es utilizar este algoritmo para la optimización de un diseño preliminar de aeronaves, para el cual tendremos primero que desarrollar una función *fitness* y un elemento a optimizar.
- La inmensa mayor parte de la mejora sucede en las primeras generaciones. Si no buscamos un resultado muy exacto, el algoritmo podrá ser mucho más rápido. Esto nos favorece en el siguiente caso de estudio, puesto que las ecuaciones están ya muy simplificadas y no tiene sentido buscar el máximo de la función más allá del orden de las unidades.
- La determinación de los parámetros (tuneado) es importante, dado que produce cambios importantes en la mejora de la búsqueda del óptimo de la función.

4 DISEÑO DE LA FUNCIÓN DE OPTIMIZACIÓN Y LOS ELEMENTOS AVIÓN

En este apartado vamos a desarrollar cómo escoger la función que mida lo adecuado que es un avión, es decir, la función *fitness*, y cuál sería el elemento más adecuado para ella. Hacemos notar que el segundo paso está supeditado al primero, debemos escoger la función *fitness* que se adecúe a nuestro problema y, posteriormente, escoger el elemento que contenga únicamente los parámetros necesarios para dicha función.

El problema de escoger una función *fitness* es muy delicado. En primer lugar, porque definir 'cómo de bueno es un avión' es algo subjetivo en general y que siempre depende de la misión del avión. Además, los cálculos en este campo pueden llegar a ser muy complejos. Otro problema que se suma es que en esta etapa de diseño no se disponen de muchos datos como para hacer un análisis profundo.

En primer lugar, calcularemos dos parámetros de la aeronave, su peso y su coeficiente de resistencia parásita. Estos dos datos no son interesantes desde el punto de vista de evaluar el *fitness*, dado que directamente no nos aportan información que nos pueda servir para evaluar la aeronave. Sin embargo, nos serán necesarios para el resto de cálculos.

A continuación, calcularemos distintas actuaciones y valores de la aeronave que sí resultan relevantes para el cálculo de *fitness*. Con los datos disponibles en este diseño preliminar, podemos calcular velocidad, rango, autonomía, coste y CASM.

Por último, mezclando estos últimos cinco valores crearemos una función *fitness* adecuada.

4.1. Peso

Existen muchas formas de estimar el peso de una aeronave [2], desde métodos estadísticos extremadamente simples (método Kundu) hasta métodos muy complejos (método Sadrey, Torenbeek,…), o incluso la fusión de distintos métodos para obtener un resultado más fiable.

En nuestro caso, necesitamos un método que use varios parámetros de la aeronave -puesto que, si no usamos parámetros, no habrá nada que optimizar-, pero que no utilice ninguno que no tengamos fácilmente a estas alturas del diseño. El método escogido que cumple con todo esto es el método de los factores lineales.

Nos basamos en la descripción del método empleada en [8], apoyándonos también en la información de [10]. El método de los factores lineales es un método estadístico, relativamente sencillo -aunque no tan básico como el método Kundu-, en el que se estima el peso de cada elemento de la aeronave a partir de un parámetro característico de dicho elemento y un factor estadístico distinto según el tipo de aeronave.

El siguiente cuadro expone los distintos elementos, los parámetros característicos y los factores estadísticos. Destacamos que la tabla presenta un error, el parámetro 0.43 realmente es 0.043.

ELEMENTO	Datos Característico	Factor para cazas	Factor para transporte y bombarderos	Factor para aviación general
Ala	S alar expuesta	44	49	12
Estabilizador Horizontal	S HTP	20	27	10
Estabilizador Vertical	S VTP	26	27	10
Cola en V	S Cola V	26	27	10
Canard	S Canard	20	27	10
Fuselaje	S Mojada fus	23	24	7
Tren de Aterrizaje	MTOW	0.033 Navy:0.045	0.43	0.057
Motores	Peso del motor	1.3	1.3	1.4
Sistemas	MTOW	0.17	0.17	0.10

Ilustración 12: Factores lineales [8]

Por ejemplo, si tenemos un avión caza con una superficie alar expuesta de 10 m^2, multiplicamos ese valor por el factor 44 y obtenemos que el ala de dicho avión pesará unos 440 kg.

Sumando todos los elementos, tendremos el peso en vacío de la aeronave (W_{empty}). Sumando la tripulación, la carga de pago y el combustible tenemos el MTOW.

No es difícil observar que el *output* de este método es el MTOW. Sin embargo, uno de los *inputs* también es el MTOW, puesto que es un factor característico para estimar el peso de los sistemas y del tren de aterrizaje. Para solucionar este problema, empleamos el teorema del punto fijo.

De acuerdo al teorema del punto fijo, podemos resolver la ecuación F(x)=0 expresándola como x=f(x), si escogemos adecuadamente f(x) para que sea contractiva –debe ser continua, derivable y su derivada debe ser menor que uno-. En dicho caso, se puede demostrar que si iteramos $x_{n+1} = f(x_n)$ converge a la solución, a mayor velocidad cuanto menor sea la derivada de la función f(x).

Centrándonos en nuestro caso, nuestra función F(MTOW)=0 se define de forma natural como MTOW=f(MTOW) de acuerdo a:

MTOW= [parámetros conocidos] + MTOW · (Factor tren aterr. + Factor sistemas)

Y dicha función es contractiva – es continua, derivable, y su derivada es una constante menor que uno - y por lo tanto se demuestra que podemos encontrar la solución de forma iterativa.

De forma gráfica, podemos ver que la iteración se hace como muestra la siguiente imagen:

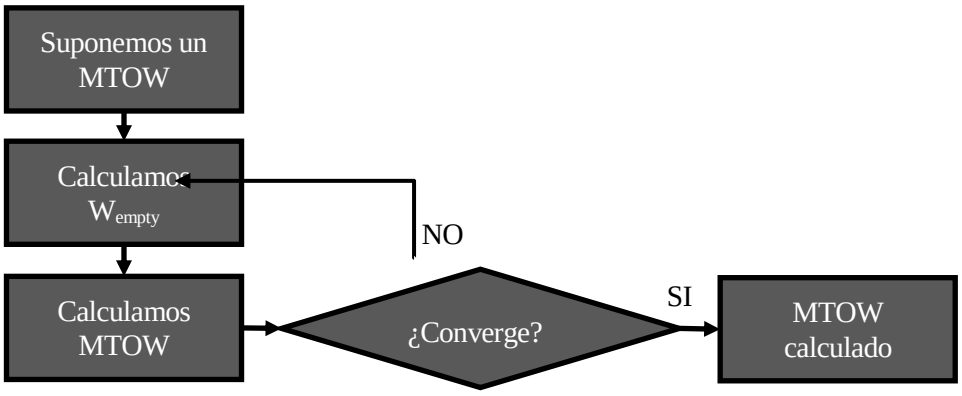


Ilustración 13: Método iterativo para el cálculo de peso

Como se ha explicado anteriormente, el peso no será empleado directamente en la función de *fitness*, pero es necesario para los siguientes cálculos.

4.2. Coeficiente de Resistencia parásita

El coeficiente de resistencia parásita es un parámetro adimensional que trata sobre la aerodinámica de una aeronave, y define cuanta resistencia genera la aeronave – aunque no esté creando ninguna sustentación-. Se emplea en el cálculo del coeficiente de resistencia global, que se define de acuerdo a la siguiente expresión:

$$C_D = C_{D0} + k\, C_L^2$$

Donde C_{D0} es el coeficiente de resistencia parásita, k es el coeficiente de resistencia inducida y C_L es el coeficiente de sustentación (de los dos últimos hablaremos en el apartado 4.3)

El coeficiente de resistencia parásita se puede estimar de forma preliminar de acuerdo a lo expuesto en [4], método que seguimos a continuación.

Este coeficiente depende de la superficie mojada, de la superficie alar y de un coeficiente dependiente de número de Reynolds conforme a la siguiente fórmula:

$$C_{D0} = C_{fe} \cdot (S_{wet}/S_{wing})$$

Donde C_{fe} se puede calcular de la siguiente forma, obtenida a partir de datos empíricos:

$$C_{fe} = 0.00258 + 0.00102 \cdot \exp(-6.28\, x\, 10^9\, Re) + 0.00295 \cdot \exp(-2.01 x 10^8\, Re)$$

A su vez, la longitud característica tomada para el número de Reynolds es la superficie mojada dividida entre la envergadura, dejando como fórmula para calcular el valor de Re la siguiente:

$$Re = \left(\frac{S_{wet}}{b}\right) \cdot \frac{V}{\nu}$$

Con esa fórmula podemos calcular una estimación del coeficiente de resistencia parásita, que será mejor cuanto más baja sea.

Nuevamente, este coeficiente no será empleado en la función *fitness*, pero será necesaria para posteriores cálculos.

4.3. Velocidad de crucero

En este apartado, utilizaremos las formulaciones obtenidas a partir de las ecuaciones de vuelo en crucero, como podemos observar en [1].

Suponemos que el avión realiza un *cruise climb*, que es el perfil de vuelo al que las aeronaves comerciales tratan de aproximarse. Este perfil consiste en un aumento lento y contínuo de la altitud conforme el peso disminuye por el consumo de combustible. Sin embargo, debido a restricciones por la gestión del tráfico aéreo, el perfil real suele ser un *stepped climb*, donde el aumento de altitud tiene lugar en escalones en lugar de ser lento y contínuo.

Para un *cruise climb* de altura inicial dada, la velocidad viene dada por la siguiente expresión:

$$V = \sqrt{\frac{2W}{\rho_i S\, C_L}}$$

Donde W es el MTOW.

El coeficiente de sustentación, C_L, es un coeficiente adimensional que cuantifica la sustentación generada por la aeronave. Éste depende del piloto, puesto que puede ser fácilmente modificado cambiando la actitud de la aeronave (mayor ángulo de ataque conlleva mayor coeficiente de sustentación, dentro de unos límites, puesto que a ángulos excesivamente grandes el coeficiente cae drásticamente). Generalmente, el coeficiente de sustentación C_L al que se vuela en crucero es el que maximiza el alcance, dado por la fórmula:

$$C_{L|(xf)max} = \sqrt{\frac{C_{D0}}{3k}}$$

Donde *k* es el coeficiente de resistencia inducida. Dicho coeficiente es un parámetro adimensional que trata sobre la aerodinámica del avión (define cuanta resistencia genera la aeronave cuando crea sustentación), y depende de la forma y la esbeltez de las alas. Se calcula mediante la siguiente fórmula:

$$k = \frac{S}{\pi \phi b^2}$$

Donde ϕ es el factor de Oswald, un valor relacionado con la forma de las alas (se aproxima a 1 cuanto más elíptica sea). En general, un compromiso entre la dificultad de fabricar un ala elíptica y la mejora de la aerodinámica es un valor del factor de Oswald de entre 0.65 y 0.85. En este trabajo supondremos 0.7, que es bastante común en aeronaves comerciales [7].

Destacamos el efecto poco intuitivo de que a mayor peso mayor velocidad. Si lo pensamos tiene sentido, pues para levantar mayor peso hace falta mayor sustentación, y para una mayor sustentación a una altura dada necesitamos mayor velocidad.

También es importante notar que para el cálculo de C_{D0} hace falta V, y viceversa. La forma de solucionar este problema es similar al peso, siguiendo iteraciones de acuerdo al siguiente diagrama:

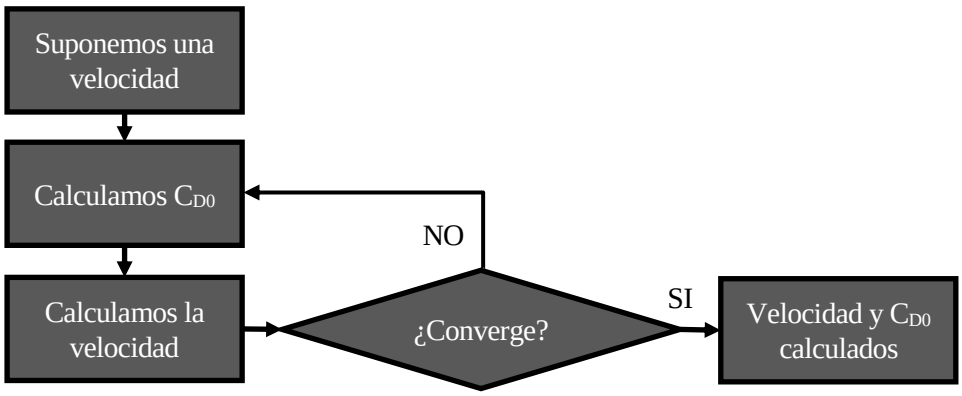


Ilustración 14: Método iterativo para el cálculo de velocidad y resistencia parásita

De la misma forma que para el cálculo de peso, este problema converge gracias al teorema del punto fijo. Aunque no sea evidente demostrarlo para cualquier valor de los parámetros, para valores típicos de los elementos la derivada de la función velocidad es mucho menor que la unidad, por lo que converge y la velocidad de convergencia será buena.

4.4. Rango y autonomía

Un dato crítico para las aeronaves es el alcance y autonomía que tienen. Para calcularlo, se supone un *cruise climb* con altitud dada (al igual que con la velocidad). Nuevamente, recurrimos a las formulaciones derivadas en [1].

Primero se hace un cálculo de la capacidad de combustible de acuerdo al tamaño de las alas. Esto es porque en los aviones comerciales el combustible se almacena en las alas generalmente. En algunas ocasiones el estabilizador horizontal también tiene depósitos de combustible o puede haber tanques en otros sitios, pero no es lo habitual. Más adelante explicaremos cómo hacer este cálculo.

Una vez sabemos cuánto combustible puede almacenar la aeronave, se calcula alcance y autonomía con las ecuaciones de Breguet de mecánica de vuelo.

$$x_{f|max} = \frac{E_{max}}{C_E(\rho_i)} \sqrt{\frac{2\,W_i}{\rho_i S}} \left(\frac{k}{C_{D0}}\right)^{\frac{1}{4}} \frac{3^{\frac{3}{4}}}{2} \ln\left(\frac{1}{1-\frac{W_F}{W_i}}\right)$$

$$t_{f|max} = \frac{E_{max}}{C_E(\rho_i)} \ln\left(\frac{1}{1-\frac{W_F}{W_i}}\right)$$

Donde la eficiencia aerodinámica máxima se calcula de acuerdo a la siguiente expresión:

$$E_{max} = \frac{C_{Lopt}}{C_D}, \text{ donde } C_D = C_{D0} + k \cdot C_{Lopt}$$

Hay que tener en cuenta que C_{Lopt} es distinto para el caso de máximo rango y máxima autonomía, y se calculan de la siguiente forma:

$$máximo\ rango: C_{Lopt} = \sqrt{\frac{C_{D0}}{3k}} \quad ; \quad máxima\ autonomía: C_{Lopt} = \sqrt{\frac{C_{D0}}{k}}$$

Para estimar la cantidad de fuel lo hacemos con la siguiente función:

$$W_{fuel} = 328.71 \cdot S_{wing} - 18213$$

Esta función ha sido obtenida de forma estadística a partir de aeronaves comerciales de transporte, en concreto de los modelos comerciales de Airbus y Boeing [6] [15] [16], y conjuntamente suponen la inmensa mayoría de las aeronaves en los cielos hoy en día. Se ha escogido la superficie alar como variable explicativa, dado que es en las alas donde se guarda la mayor parte del combustible. En la siguiente gráfica, vemos que la dispersión respecto a la recta de mínimos cuadrados es pequeña, obteniendo una aproximación muy buena.

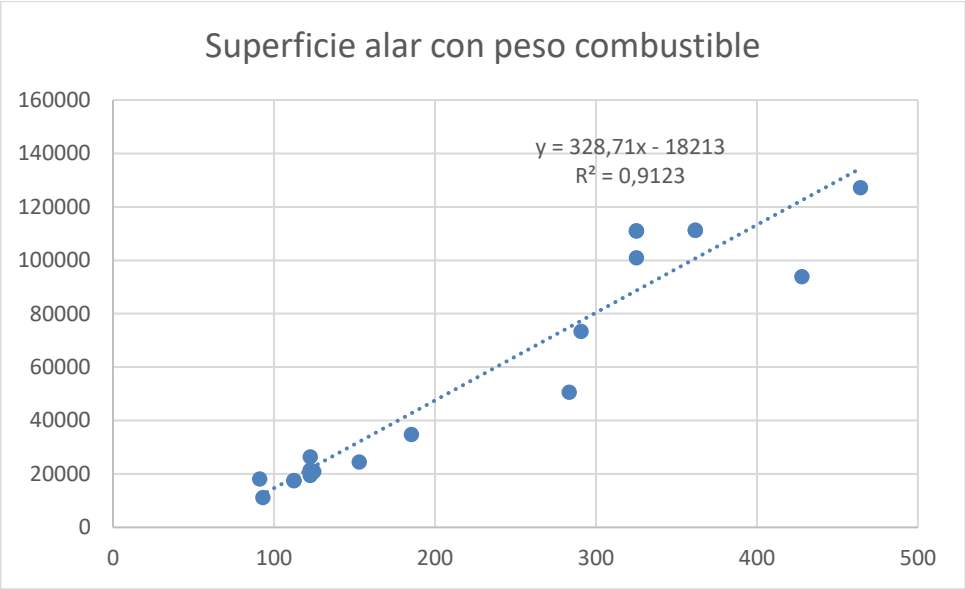


Ilustración 15: Ajuste del peso de combustible respecto a la superficie alar

4.5. Coste

La estimación de coste es un factor fundamental en el diseño de una aeronave, pero también es uno de los más difíciles de estimar.

En general, para estimar de forma convenientemente exacta el precio final de una aeronave hace falta una multitud de datos que no están presentes hasta la fase de diseño de detalle. Sin embargo, durante el comienzo de la fase preliminar se puede tener una idea aproximada mediante los modelos de estimación de coste RAND de ecuaciones paramétricas [3] [5]. Estos modelos fueron desarrollados para que el departamento de defensa de los Estados Unidos pudiera estimar el coste de los nuevos programas en una fase muy temprana de diseño. Aunque los resultados no siempre son muy exactos, no se han obtenido mejores estimaciones con los datos disponibles en un estado de diseño preliminar.

Destacamos que este coste es el de la estructura del avión. Muchas cosas no están incluidas como motores, aviónica, muebles, margen de beneficio del fabricante…, por lo que el coste estimado en este punto será muy inferior al coste final de venta.

Estos modelos buscan parámetros que cumplan dos condiciones. En primer lugar, ser datos disponibles desde un punto muy temprano del diseño. En segundo lugar, que ajusten de la forma más correcta posible los datos estadísticos de precio de diseños pasados.

Los únicos dos parámetros que cumplen esto son el peso y la velocidad. A partir de estos dos datos, se estiman las horas de trabajo en cada área (ingeniería, utillaje, fabricación, …), se multiplican las horas por el coste por hora asociado a dicha área, se pasa el dinero con la inflación al valor actual, se suman otros costes monetarios también estimados y con ello se tiene el valor total de una aeronave.

Destacamos que, aunque la técnica sobreestime o subestime el coste, lo importante es que lo hará con todos los elementos avión por igual, de forma que la optimización sigue funcionando.

Con los siguientes cálculos estimamos horas de trabajo:

$$engineering_{hours} = 0.023 \cdot Wempty_{pounds}^{0.66} \cdot V^{0.96};$$

$$tooling_{hours} = 0.47 \cdot Wempty_{pounds}^{0.64} \cdot V^{0.5};$$

$$manufacturing_{hours} = 0.35 \cdot Wempty_{pounds}^{0.79} \cdot V^{0.42};$$

$$quality_{hours} = manufacturing_{hours} \cdot \begin{Bmatrix} 0.085\ for\ cargo\ aircraft \\ 0.12\ for\ other \end{Bmatrix}$$

Transformamos las horas de trabajo a dólares a partir de la siguiente tabla:

COMPOSITE HOURLY RATES IN 1973 DOLLARS

Company	Engineering	Tooling	Quality Control	Manufacturing
A	21.13	21.86	16.19	15.39
B	21.75	16.76	16.26	15.39
C	18.00	17.84	19.61	16.19
D	19.88	18.93	16.12	17.81
E	19.55	17.80	15.95	14.06
Mean	20.06	18.63	16.83	15.77

Ilustración 16: Tabla con el coste horario de distintas áreas

A ese coste, sumamos los costes de materiales y de test de vuelo de acuerdo a las siguientes ecuaciones:

$$materials_{nonrecurring}(\$) = 0.000024 \cdot Wempty_{pounds}^{0.72} \cdot V^{1.92};$$

$$materials_{recurring}(\$) = 0.05 \cdot Wempty_{pounds}^{0.88} \cdot V^{0.87};$$

$$fligh\ test\ (\$) = 0.13 \cdot Wempty_{pounds}^{0.71} \cdot V^{0.59} \cdot \#testplanes^{0.72} \cdot \begin{Bmatrix} 2\ for\ cargo\ aircraft \\ 1\ for\ other \end{Bmatrix}^{-1.56};$$

Finalmente, se suman todos los costes y se multiplican por 5.8 para ajustar por la inflación.

4.6. Cost per available seat mile (CASM)

Este es el parámetro de eficiencia más usado por las aerolíneas. Se calcula el número de asientos posible de acuerdo a las dimensiones del fuselaje, y se divide el coste de operación para máximo rango entre las millas que puede recorrer y el número de asientos.

Es muy frecuente asumir que el mayor coste de operación será el combustible y simplificar el coste de operación por el coste de combustible. Dado que tenemos el peso de combustible de la aeronave y el precio obtenido de la página oficial de IATA actualizado diariamente, con facilidad calculamos el coste de combustible.

Este parámetro es muy interesante para las aerolíneas, dado que es de donde pueden estimar el beneficio que pueden sacar a la operación de la aeronave.

En primer lugar, el cálculo de asientos se hizo de forma similar al cálculo de combustible, pero siendo algo más complejo. En primer lugar, buscamos posibles variables explicativas, y a continuación vemos cuál de ellas correlaciona mejor con el número de asientos a partir de datos de aeronaves comerciales.

Respecto al número de asientos hay dos posibilidades, que sea según el número típico de asientos o según el número máximo de asientos. Se comprobarán ambos para verificar cuál correlaciona mejor.

Las variables planteadas son la longitud del fuselaje y el área de fuselaje (longitud por ancho). Esta última es la que *a priori* parece más explicativa, pero hay que considerar que las aerolíneas cambian mucho el ancho de los asientos y pasillos, por lo que puede que el ancho de la cabina no sea tan relevante.

Finalmente, otra variable a considerar será separar por *narrowbody* (aviones con un solo pasillo) o *widebody* (aviones con doble pasillo).

A continuación, comprobamos todas las posibles rectas de mejor ajuste para ver que la mejor opción es no discriminar el número de pasillos y ajustar el mayor número de asientos según el área del fuselaje. Obtenemos muy buena correlación, siendo la decisión final utilizar el área del fuselaje (ancho por largo) para obtener el máximo número de asientos, sin discriminar entre *widebody* y *narrowbody*. Datos obtenidos de [15] y [16].

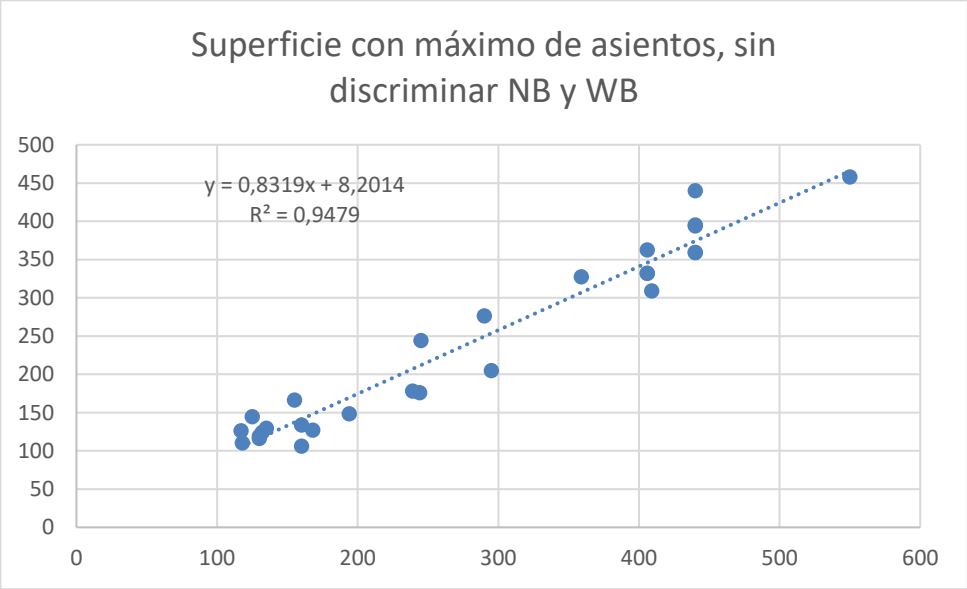


Ilustración 17: Ajuste del número de asientos en función del fuselaje

Sin embargo, aunque el ajuste es muy bueno, las variables son explicativas y, para cualquier avión comercial, predice muy bien el máximo número de asientos, no proporciona buenos resultados en el código.

Esto es debido a que el algoritmo evolutivo hace, mediante prueba y error, fuselajes con características que se salen por completo de lo normal. En particular, si ajustamos el número de asientos mediante esta recta obtenemos aviones extraordinariamente estrechos y largos que el código, de forma errónea, piensa que puede soportar muchos asientos cuando esto no es cierto, debido a que el pasillo quitaría mucha área.

Es por ello que debemos cambiar completamente de técnica para estimar el número de asientos. Pasamos a solicitar al usuario del código que introduzca el ancho de asientos, pasillo y espacio entre filas deseado de acuerdo al nivel de confort que desee para los pasajeros (de forma predeterminada toma valores estándar para clase turista).

Una vez sabemos el ancho de los asientos, vemos cuántos podemos poner en el ancho de la cabina

$$n_{seats_{row}} = floor(\frac{cabin_{width}}{seat_{width}})$$

Si el espacio restante no es suficiente como para situar el pasillo, restamos uno al número de asientos por fila.

Si, después de la comprobación anterior, tenemos más de 6 asientos por fila, tenemos que comprobar si tenemos espacio para situar un segundo pasillo. Si no es así, tenemos que eliminar otro asiento.

Si, después de la comprobación anterior, tenemos más de 12 asientos por fila, nuevamente comprobamos si tenemos espacio para situar un tercer pasillo. Si no es así, tenemos que eliminar otro asiento.

Tras todo esto sabemos cuál es el número de pasillos y el número de asientos por fila. Mediante el espacio entre filas y la longitud del fuselaje, sabemos cuántas filas podemos situar y con ello el número total de asientos.

4.7. Función completa

Esos son todos los parámetros que podemos calcular de forma sencilla con los datos de un diseño poco avanzado. Ahora tenemos que operar con dichos resultados hasta llegar a un único número real que mida lo adecuado que es el avión.

Los parámetros que realmente relevantes a la hora de evaluar el avión son los siguientes:

- Velocidad
- Coste
- Rango
- Autonomía
- CASM

El peso y la resistencia aerodinámica son necesarios para calcular dichos parámetros, pero realmente no son relevantes para la evaluación (lo que al cliente le importa es el CASM, no el factor de resistencia parásita, por ejemplo).

Antes de continuar, dedicamos un momento a verificar que los cálculos son aceptables. Para ello seleccionamos un avión característico de la misión de transporte, un B737, el avión más vendido de la historia, en concreto en su primer modelo que se fabricó, el B737-100.

Con los datos físicos del B737-100 calculamos las actuaciones con nuestras ecuaciones y comparamos los resultados obtenidos con los valores reales de las actuaciones de un B737-100.

Características físicas de un B737-100:

- Superficie alar: 102 m^2
- Superficie HTP: 28.99m^2
- Diámetro fuselaje: 3.76 m
- Longitud del fuselaje: 27.66 m
- Envergadura: 28.35 m

En la siguiente tabla se resumen las actuaciones calculadas y las reales, para comprobar cómo de justa es la función de *fitness*.

	Valores reales B737-100 [14] [15]	Valores calculados B737-100	Porcentaje error
Velocidad	216 m/s	293 m/s	26%
Coste	7.5 mill. $	5.6 mill. $	-34%
Autonomía	4.16 h	3.25 h	-28%
Alcance	3185 km	2984 km	-7%
CASM	0.050 $/seat·mile	0.057 $/seat·mile	12%

Tabla 10: Verificación de las cualidades de la function fitness

Tenemos que destacar los siguientes aspectos:

- La velocidad se sobreestima debido a que el peso se sobreestima. El método de los factores lineales, en este caso, calcula un peso excesivo y ello lleva a un cálculo de velocidad excesiva.
- La estimación de coste puede tener error, pero en este caso eso se complica, dado que no conocemos el coste de fabricación de la estructura de un B737-100. El valor propuesto es estimativo a partir del coste de venta de un B737-100 en 1972 [14], ajustando a la inflación, y descontando un porcentaje estimado del precio de los motores y otros elementos no tenidos en cuenta en el modelo RAND, pero es muy probable que el coste sea menor y, por lo tanto, la estimación sea más acertada.
- No hay datos concretos de autonomía en este tipo de aviones, pero se estima a partir del alcance y velocidad.

- En general, los valores obtenidos son similares a los valores reales, por lo que los cálculos que realiza la función de *fitness* quedan validados.

Una vez hecha esta verificación, proseguimos con el cálculo de la función de *fitness*.

En estos momentos tenemos una serie de valores, pero tenemos que destacar que los parámetros tienen unidades distintas y, para poder operar con todos ellos de forma justa vamos a adimensionarlos.

Para llevar a cabo esta adimensionalización vamos a dividirlos por los valores del avión B737-100 del que hemos hablado anteriormente.

Los valores adimensionales que obtenemos en el avión que estamos optimizando tendrán valores alrededor de la unidad. Un valor por encima de 1 implica que el avión calculado es mejor que un B737-100 en ese aspecto. Un valor menor de 1 implica lo contrario.

A continuación, multiplicamos cada uno de los valores obtenidos por un factor que dependerá de la importancia que el usuario le dé a esa característica. Para ello necesitamos que previo a todo este proceso, el usuario valore qué características prefiere.

Finalmente, se suman todos los valores y ese será el *fitness* del avión, que mide lo adecuado que es.

4.8. Elemento avión

Como ya hemos dicho, cada elemento de la población será un vector. Los parámetros físicos del avión necesarios para el cálculo de la función *fitness* son:

[Superficie alar, superficie HTP, diámetro fuselaje, longitud del fuselaje, envergadura]

Además de estos parámetros, daremos por supuesto desde el inicio una serie de valores típicos:

- Cargo: 1 si la aeronave no es de carga, 2 en caso contrario.
- Consumo específico de los motores: 4.0548e-04 1/s, equivalente al de dos motores JT8D, típicos en estas aeronaves.
- Peso de los motores: 4300 kg, equivalente al de dos motores JT8D, típicos en estas aeronaves.
- La superficie del VTP se considera un 70% la del HTP
- Diámetro vertical y horizontal del fuselaje se consideran iguales.
- El número de tripulantes, incluyendo piloto y copiloto, depende del número total de pasajeros de acuerdo a la siguiente fórmula.

  Tripulantes = 3+ pasajeros/50
- El peso de cada persona se considera 100 kg, e incluye las maletas.
- El taper ratio (relación entre la cuerda del ala en la punta y la raíz) es de 0.3.

Destacamos que partimos de estos valores para poder realizar cálculos concretos en el siguiente apartado. Para realizar otro análisis, podríamos modificar estos parámetros convenientemente.

4.9. Comentarios sobre la altitud

Durante la función *fitness* se da por supuesto que el vuelo tiene lugar a 10 km de altitud. Sin embargo, dada la aleatoriedad con que se generan los aviones (en particular las primeras generaciones) es perfectamente posible que se produzcan elementos con un tamaño de ala insuficiente para generar la sustentación necesaria a dicha altitud. Ante esto nos encontramos dos posibilidades:

- Descartar todos los elementos para los cuales se calcule que no podrían mantener un crucero a 10 km de altitud.
- Recalcular de forma iterativa la altitud óptima de cada avión y calcular sus características a dicha altitud.

Se toma la primera alternativa por las siguientes razones:

- Resolver de forma iterativa la altitud es muy costoso computacionalmente (el proceso se muestra abajo, y en él hay que tener en cuenta que la velocidad ya tiene interiormente un proceso iterativo en su cálculo).
- En la vida real, los aviones no vuelan a su altitud óptima, por motivos de gestión del tráfico aéreo.
- La suposición de unos 10 km de altitud es muy aceptable para un amplio espectro de aeronaves comerciales.

El cálculo de acuerdo a la segunda opción, el proceso iterativo sería como muestra el siguiente esquema:

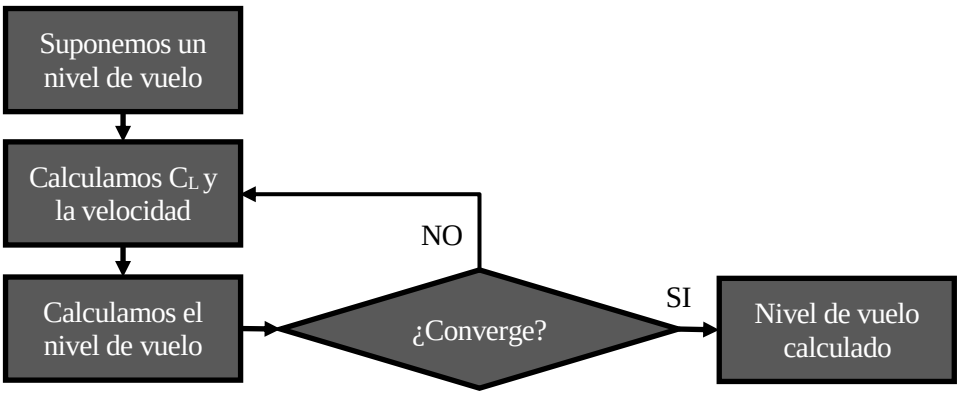


Ilustración 18: Método iterativo para el cálculo del nivel de vuelo.

Respecto al cálculo del nivel de vuelo a partir de C_L y la velocidad, tendríamos que hacer un equilibrio de fuerzas. Por un lado, el peso, y por el otro, la sustentación.

$$P = \frac{1}{2}\,\rho\, V^2 S_{wing} C_L$$

Despejando en esta ecuación la densidad y a partir del modelo de atmósfera estándar podemos obtener la altitud de vuelo.

4.10. Comentarios sobre el motor

En este programa se ha hecho una importante simplificación, que consiste en suponer que la aeronave va a tener dos motores JT8D.

Este tipo de motorización ha sido habitual en aviones comerciales como los que tratamos de optimizar. Sin embargo, para aviones especialmente grandes o pequeños es posible que esta motorización no sea adecuada. En particular, para aviones pequeños, los motores proporcionarían un exceso de empuje que obligaría a tenerlos

funcionando a baja capacidad, en la que no son tan eficientes y por tanto el consumo específico real sería mayor que el proporcionado. Para aviones grandes, los motores no proporcionarían suficiente empuje como para vencer la resistencia a la velocidad calculada, lo que obligaría al avión a disminuir de velocidad y, por ello, a bajar el nivel de vuelo, por lo que los resultados calculados tampoco serían correctos.

A pesar de ello, consideramos la motorización aceptable por ser una opción intermedia, válida para un amplio rango de aeronaves.

Una forma de mejorar el programa consistiría en incluir la motorización en los factores a optimizar. Sin embargo, no podemos tratarlo de la misma forma que el resto de parámetros, dado que los motores no se fabrican 'hechos a medida' para un avión, sino que se escogen entre el catálogo de los fabricantes de motores. Además, si dejáramos que los parámetros del motor fueran optimizados no sería ninguna sorpresa ver que el óptimo sería un motor de mínimo peso, mínimo consumo específico y máximo empuje, sin que eso sea realista. Es por ello que la forma más adecuada sería crear una estructura en Matlab en la que cada fila representaría un motor de catálogo (con el peso, empuje óptimo, consumo específico a empuje óptimo, precio y un número identificador). Una vez se tenga dicha estructura, se podría asignar al elemento avión un número identificador al azar y un número de motores también al azar. Este identificador podría evolucionar a través de la mutación (dado que la mezcla no tendría mucho sentido en este parámetro) y el número de motores podría evolucionar normalmente.

Habría que añadir finalmente una comprobación de que el motor tiene una potencia adecuada, ni demasiada para que durante el crucero funcione muy por debajo de la potencia óptima, ni demasiada poca como para que no pueda alcanzar la velocidad adecuada en el nivel de vuelo establecido. Si estamos evolucionando el nivel de vuelo, estas comprobaciones no serían necesarias, dado que la potencia de motor simplemente modificaría el nivel de vuelo.

En cualquier caso, tendríamos que ver que se cumple la siguiente ecuación, siendo el empuje T cualquier valor del rango de empujes al que el motor es eficiente:

$$T = \frac{1}{2}\rho V^2 S_{wing} C_{D0}$$

Si estamos optimizando el nivel de vuelo, tenemos que despejar el valor de la densidad para el T óptimo. Si no lo estamos optimizando, tenemos que comprobar que se cumple para algún T dentro de un rango aceptable y, si no es así, hacer cero el fitness del elemento para extinguirlo por tener una motorización inadecuada.

5 PROGRAMACIÓN DE NUESTRO CASO

En este apartado vamos a ir punto por punto explicando cómo se ha llevado a cabo la programación de la estrategia evolutiva aplicada al caso del diseño de aeronaves.

En todo el capítulo se van a recortar los comentarios en gran parte, dado que vamos a explicar lo que ellos dicen con aún más profundidad.

5.1. master.m

Este es el programa que se debe ejecutar para el funcionamiento del código. El resto de programas son módulos que son llamados desde el programa master. En este programa están todos los parámetros que deben ser ajustados para las distintas optimizaciones que se quieran llevar a cabo.

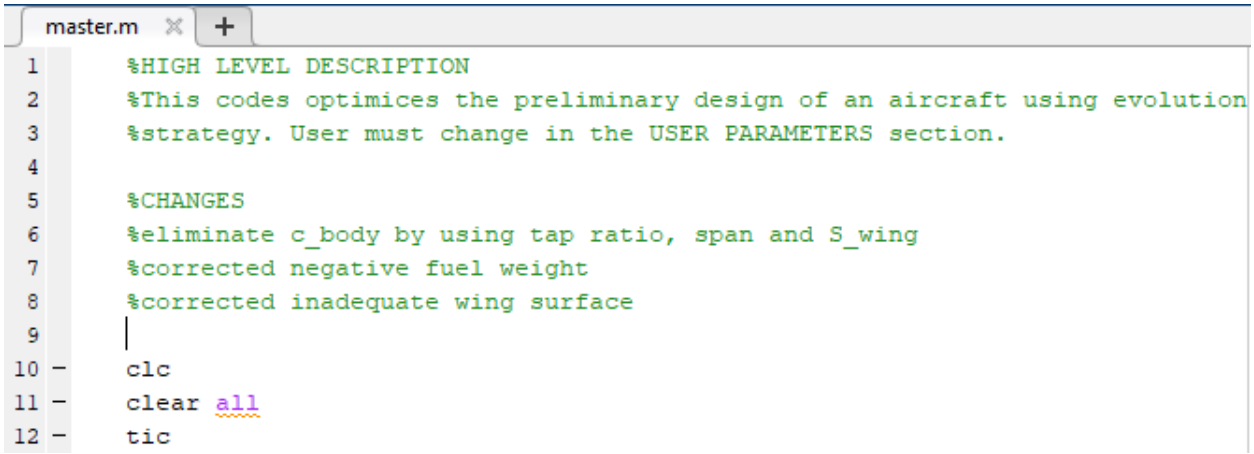


Ilustración 19: master.m imagen 1.

Comenzamos el programa con una sección de comentarios, en la que se describe el código y las correcciones respecto a previas versiones del código. Se limpia la memoria de Matlab para evitar que previas ejecuciones del programa puedan causar problemas y se inicia el cronómetro.

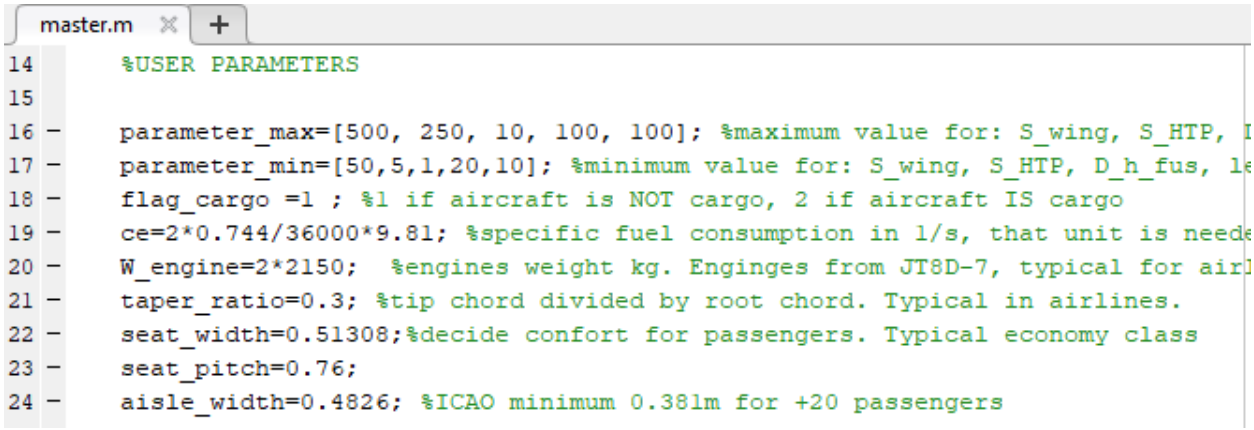


Ilustración 20: master.m imagen 2.

Este apartado es importantísimo, puesto que en él es donde el usuario pone sus condiciones.

En primer lugar (líneas 16 y 17), se ponen los máximos y mínimos valores aceptados para los parámetros que estamos optimizando, que son la superficie alar, superficie del HTP, diámetro del fuselaje, longitud del fuselaje y envergadura. Puede estar limitado por las características del aeropuerto en el que se requiera volar, legislación, etc.

Se especifica si el avión es de carga o no (línea 18).

Se escoge un motor, con su peso y consumo específico (líneas 20 y 21).

Se especifica el estrechamiento de las alas, siendo el valor 0.3 el típico en aeronaves comerciales (línea 21).

Se decide la comodidad de los pasajeros mediante el tamaño de asientos y pasillo. Los valores preseleccionados son los típicos en clase turista (líneas 22, 23 y 24).

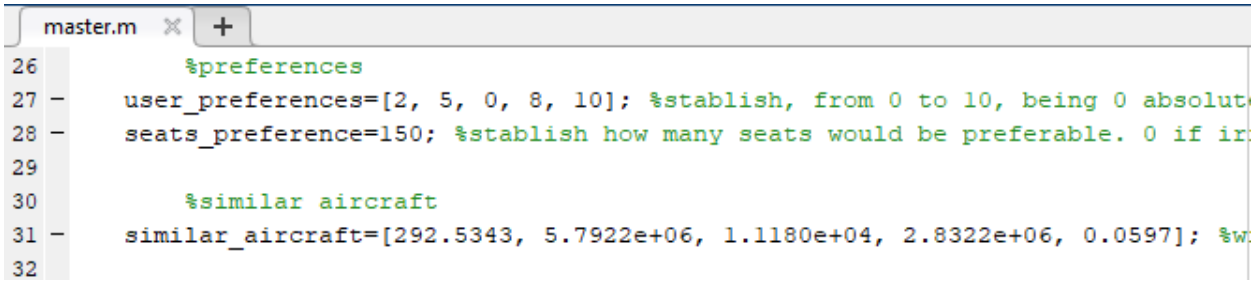


Ilustración 21: master.m imagen 3.

Se establecen cuáles son las prioridades de diseño, valorando de 0 a 10 la importancia de las características que son: velocidad, coste, autonomía, rango y CASM. También se especifica el número deseado de asientos, seleccionando 0 si el número de asientos es irrelevante.

Se establecen los datos de una aeronave similar para llevar a cabo la adimensionalización. Los datos preseleccionados son de un B737-100, que es muy aceptable para un amplio rango de aeronaves.

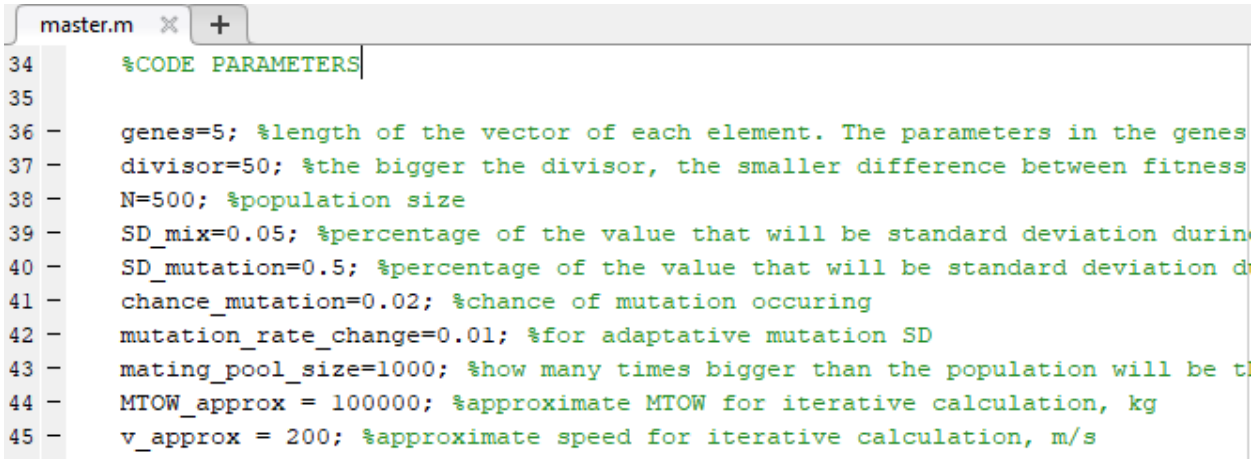


Ilustración 22: master.m imagen 4.

Se establecen distintos parámetros del código.

El número de genes en nuestro caso es 5, que es el número de parámetros que estamos optimizando.

El divisor, como explicado anteriormente, regula la velocidad a la que se maximizan las diferencias en la *mating pool* y, por tanto, modifica el equilibrio entre exploración y convergencia.

El tamaño de población de 500 es adecuado para que las generaciones sean suficientemente grandes para garantizar la exploración sin que se ralentice en exceso la ejecución del código.

La desviación estándar de la mezcla será pequeña relativa a los valores de los parámetros. Dado que cada parámetro tiene un rango distinto, el valor que se da de desviación estándar no es absoluto, sino que se refiere al porcentaje del valor de la media que será la desviación estándar. Por ejemplo, si un padre tiene un ala de 100 y el otro padre de 300, la media será 200 y el 5% de este valor, es decir 10, será la desviación típica de la mezcla. Por otro lado, si el diámetro de uno es 3 y del otro es 5, la media es 4 y la desviación típica será un 5% de este valor, es decir 0.2. Con esto conseguimos que, para distintos parámetros, la desviación estándar sea distinta, pero siempre aceptable en relación a los valores de los parámetros en sí.

Del mismo modo ocurre con la desviación típica de la mutación, pero que será mucho más grande, empezando con un 50% y ajustándose a lo largo de las generaciones de acuerdo a la técnica explicada anteriormente.

La tasa de variación de la desviación típica será un 1%, pequeña para que tarde bastantes generaciones en adaptarse y no disminuya demasiado rápido.

El tamaño de la *mating pool* será suficientemente grande como para dar diversidad, pero no tanto como para ralentizar en exceso.

Se dan valores aproximados de MTOW y velocidad, aunque la convergencia en la iteración es tan rápida que no importa que estén lejos de los valores correctos.

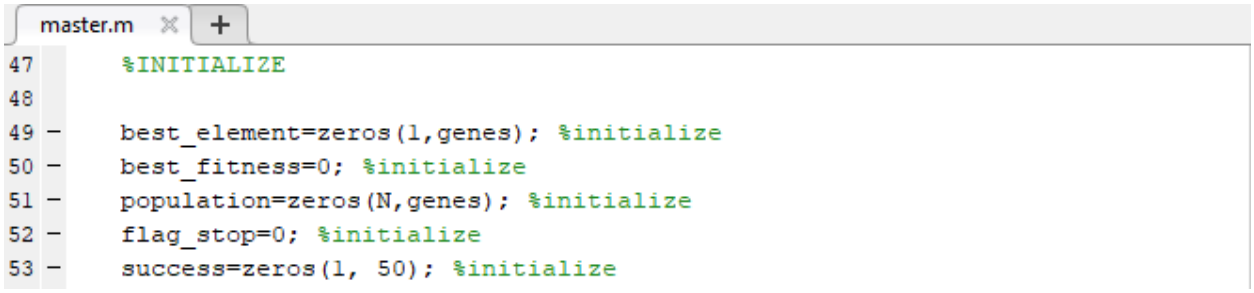


Ilustración 23: master.m imagen 5.

Se inicializan los elementos a valores adecuados.

```
55      %OPTIMIZATION ALGORITHM
56
57          %creation of original population
58          for i=1:N %for every member of the population
59              population(i,:)=rand(1,genes); %we create random elements
60              population(i,:)=(population(i,:)+parameter_min./...
61              (parameter_max-parameter_min)).*(parameter_max-parameter_min); %we
62          end
63          generation=1 %this random population is the first generation
```

Ilustración 24: master.m imagen 6.

Mediante un bucle *for* se genera la primera generación aleatoria. El primer paso es crear elementos aleatorios (línea 59) y, el segundo paso, es dimensionar estos elementos aleatorios a los máximos y mínimos establecidos (líneas 60 y 61).

```
64          while flag_stop==0 %while we have not decided to stop
65              %fitness calculation
66              fitness=zeros(N,1); %initialize
67              for i=1:N %for every member of the population
68                 fitness(i)=fitnessfunctionmaster_v1(population(i,:), ...
69                     MTOW_approx, v_approx, flag_cargo, taper_ratio, ce, ...
70                     W_engine, user_preferences, seats_preference, seat_width,...
71                     seat_pitch, aisle_width, similar_aircraft); %we calculate th
72              end
73              [max_fitness, position_max]=max(fitness); %we save the maximum valu
74              if max_fitness>best_fitness %if the maximun fitness of this generat
75                  best_fitness=max_fitness;%this is the new all-time maximum
76                  best_element=population(position_max,:); %this element must be
77              end
78              fitness_improvement(generation)=best_fitness; %in this vector we st
79              element_improvement(generation, :)=best_element; %in this vector we
```

Ilustración 25: master.m imagen 7.

En este bucle *while* comienza el algoritmo evolutivo. La variable *flag_stop* está inicializada en valor 0, por lo que el programa entra en el bucle. Esta variable cambiará a valor 1 cuando se decida que ya se ha acabado la optimización.

Se inicializa el vector de valores de *fitness* para acelerar el proceso. Posteriormente este vector se reescribe mediante el bucle *for*, en el que se llama a la función *fitnessfunctionmater_v1*, que explicaremos más adelante. Lo fundamental es conocer que esta función proporciona, dado un elemento y una serie de parámetros, un valor real, que es el *fitness* de dicho elemento.

Una vez acabado este bucle *for*, tenemos un vector de valores de *fitness*. En este vector buscamos el valor máximo. Si este valor máximo es el mayor que hemos encontrado hasta ahora (en la primera generación lo será, en el resto de generaciones dependerá de si ha habido mejora o no) guardamos como *best_fitness* y el elemento que proporciona ese valor lo guardamos como *best_element*.

Para poder ver la evolución del mejor *fitness* y el mejor elemento a lo largo de las generaciones, guardamos en cada generación estos dos elementos en un vector que podremos representar al final.

master.m

```
77          %creation of mating pool
78 -        if min(fitness)<=0
79 -            mating_pool_number=fitness+(abs(min(fitness)))+1;
80 -        else
81 -            mating_pool_number=fitness./min(fitness);
82 -        end
83 -        mating_pool_number=mating_pool_number.^(1+generation/divisor);
84 -        mating_pool_number=round(mating_pool_number.*100);
85 -        if sum(isinf(mating_pool_number))>=1
86 -            divisor=generation;
87 -            for i=1:length(mating_pool_number)
88 -                if isinf(mating_pool_number(i))==1
89 -                    mating_pool_number(i)=1;
90 -                else
91 -                    mating_pool_number(i)=0;
92 -                end
93 -            end
94 -        end
95 -        while sum(mating_pool_number)>mating_pool_size*N
96 -            mating_pool_number=round(mating_pool_number./3);
97 -        end
98 -        mating_pool=zeros(sum(mating_pool_number),genes);
99 -        k=1;
100 -        for i=1:N
101 -            for j=1:mating_pool_number(i,1)
102 -                mating_pool(k,:)=population(i,:);
103 -                k=k+1;
104 -            end
105 -        end
```

Ilustración 26: master.m imagen 8.

Ya que tenemos una población y los *fitness* asociados a esta población, vamos a crear la *mating pool* de la que sacaremos las parejas para la nueva generación. Recordamos que la *mating pool* es un conjunto en el que cada elemento está repetido más veces cuanto mayor sea su *fitness*.

Vamos a seguir los pasos explicados previamente, cuando se explicó en detenimiento el funcionamiento de la *mating pool.*

En las líneas de 78 a 82 se opera para que el menor de los *mating_pool_number* sea valor 1, y el resto sean proporcionales.

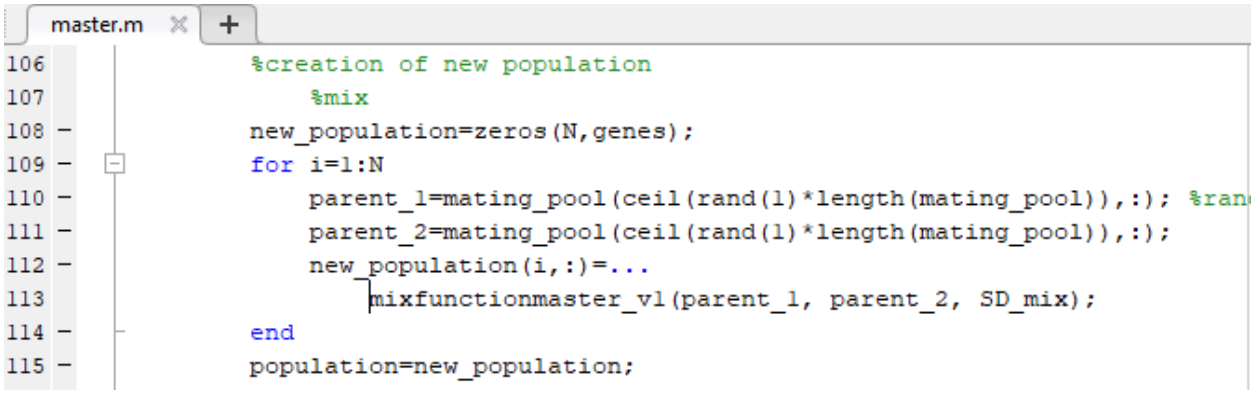
En la línea 83 se amplifican las diferencias entre los *mating_pool_number*.

En la línea 84 se multiplica por 100 y redondea.

En las líneas de 85 a 94 confirmamos que no hay infinitos. Si los hubiera, se iguala el parámetro 'divisor' a la generación para evitar infinitos en el resto de generaciones y se igualan los infinitos a 1 y el resto a 0.

En las líneas 95 a 97 se reduce el tamaño de la *mating_pool'* para que no sea inaceptablemente grande.

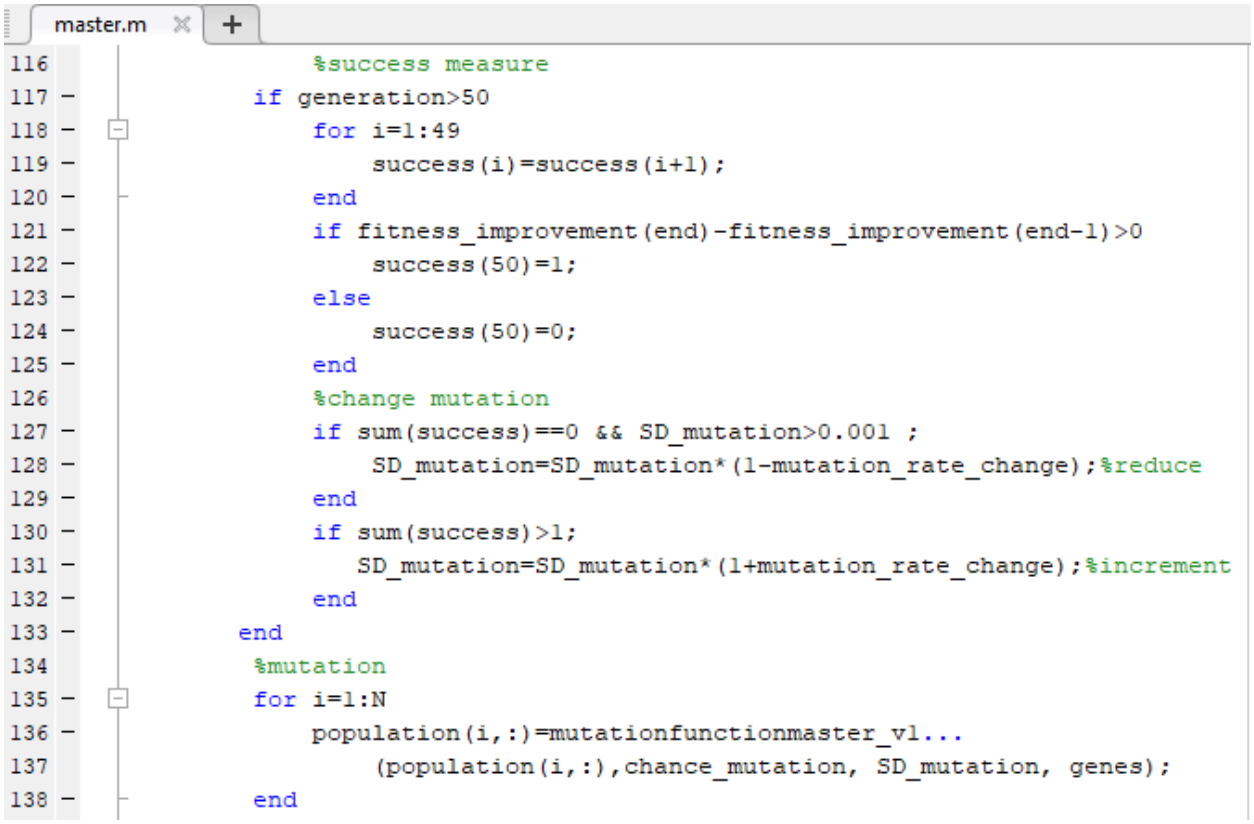
Finalmente, en las líneas 98 a 105 se crea la *mating_pool*, repitiendo cada elemento el número de veces que dice su *mating_pool_number*.



```
106          %creation of new population
107              %mix
108          new_population=zeros(N,genes);
109          for i=1:N
110              parent_1=mating_pool(ceil(rand(1)*length(mating_pool)),:); %ran
111              parent_2=mating_pool(ceil(rand(1)*length(mating_pool)),:);
112              new_population(i,:)=...
113                  mixfunctionmaster_v1(parent_1, parent_2, SD_mix);
114          end
115          population=new_population;
```

Ilustración 27: master.m imagen 9.

En este punto creamos la nueva población. Para ello, mediante un bucle *for* repetimos tantas veces como el número de población el proceso de mezcla. En él, se escoge al azar dos elementos de la *mating pool* y se mezclan mediante la función *mixfunctionmaster_v1*, que explicaremos más adelante.



```
116              %success measure
117          if generation>50
118              for i=1:49
119                  success(i)=success(i+1);
120              end
121              if fitness_improvement(end)-fitness_improvement(end-1)>0
122                  success(50)=1;
123              else
124                  success(50)=0;
125              end
126              %change mutation
127              if sum(success)==0 && SD_mutation>0.001 ;
128                  SD_mutation=SD_mutation*(1-mutation_rate_change);%reduce
129              end
130              if sum(success)>1;
131                 SD_mutation=SD_mutation*(1+mutation_rate_change);%increment
132              end
133          end
134          %mutation
135          for i=1:N
136              population(i,:)=mutationfunctionmaster_v1...
137                  (population(i,:),chance_mutation, SD_mutation, genes);
138          end
```

Ilustración 28: master.m imagen 10.

En este apartado hacemos la mutación, para lo cual primero tenemos que ajustar el valor de la tasa de mutación.

Medimos el éxito (líneas 117 a 125). Consideramos que una generación tiene éxito si el *best_fitness* se ha incrementado.

A continuación, cambiamos la mutación (líneas 127 a 133) de acuerdo al éxito que hemos medido. Se reduce o incrementa de acuerdo al *mutation_rate_change* que habíamos establecido.

Por último, se muta la población, mediante un bucle *for* y llamando a la función *mutationfunctionmaster_v1*, que explicaremos más adelante.

```
master.m
142          %check everything is within limits
143 -        for i=1:N
144 -            for j=1:genes
145 -                if population(i,j)>parameter_max(j)
146 -                    population(i,j)=parameter_max(j);
147 -                end
148 -                if population(i,j)<parameter_min(j)
149 -                    population(i,j)=parameter_min(j);
150 -                end
151 -            end
152 -        end
153 -        population(end,:)=best_element; %the best element must be saved for
154 -        generation=generation+1
155
156          %check stop criteria
157 -        time=toc;
158 -        flag_stop=...
159      stopcriteriamaster_v1(generation, fitness_improvement, time); %we check if
160 -    end
161
162      %FINAL SOLUTION
163 -    generation
164 -    time
165 -    best_fitness
166 -    best_element
```

Ilustración 29: master.m imagen 11.

Finalmente comprobamos que en la nueva población todo está dentro de los límites. Si algún parámetro no lo cumple, se mueve al límite más cercano.

En la línea 153 se añade el *best_element* a la población, para no perder el elemento óptimo hasta el momento.

Mediante la función *stopcriteriamaster_v1* se decide si se para (poniendo *flag_stop* igual a 1) o si continuamos el proceso (manteniendo *flag_stop* igual a 0).

Cerramos el bucle *while* (línea 160).

Finalmente presentamos la generación a la que se ha llegado, el tiempo, el mayor *fitness* y el mejor elemento.

5.2. Fitnessfunctionmaster_v1.m

Este es el módulo al que llama el programa *master.m* a la hora de evaluar el *fitness* de un elemento. Vamos a desgranarlo y comentarlo en distintas partes, al igual que en el apartado anterior. Destacamos que es un código casi tan largo como el programa *master.m* y más complejo en los cálculos.

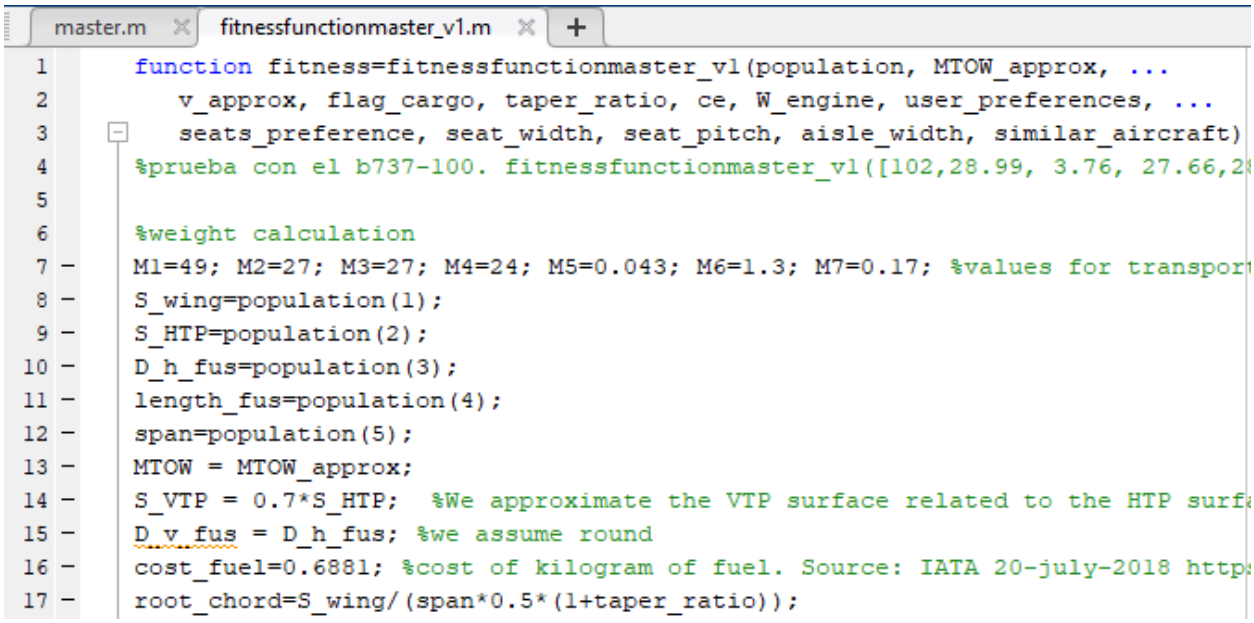


Ilustración 30: fitnessfunctionmaster_v1 imagen 1.

Vemos que este programa utiliza una enorme cantidad de argumentos para devolver un único resultado, que es el número real del *fitness* del elemento. Además del elemento en sí, tiene otros argumentos como son el peso y la velocidad aproximados, si es cargo o no, el estrechamiento de las alas, las características de los motores, las preferencias del usuario incluyendo la comodidad para los pasajeros y los datos de una aeronave similar. Todos estos datos se han obtenido durante la ejecución del programa *master.m*.

Los valores proporcionados en la línea 7 son los factores lineales para el cálculo del peso. Estos factores vienen dados por el método de factores lineales y son para aeronaves de transporte o bombarderos. En caso de cazas, UAVs o aeronaves de aviación general los valores serían distintos.

Después, extraemos los datos por separado del vector del elemento avión, para ser utilizados más cómodamente (líneas 8 a 12).

Establecemos la superficie del VTP como un 70% del valor de la superficie del HTP (un valor habitual en las aeronaves comerciales) y en la línea 15 hacemos la simplificación de que el fuselaje es redondo, por lo que el diámetro vertical y horizontal se hacen iguales.

Establecemos el precio del combustible de acuerdo a IATA (línea 16). Por último, se hace el cálculo de la cuerda del ala en la raíz, a partir del estrechamiento, de la envergadura y de la superficie alar.

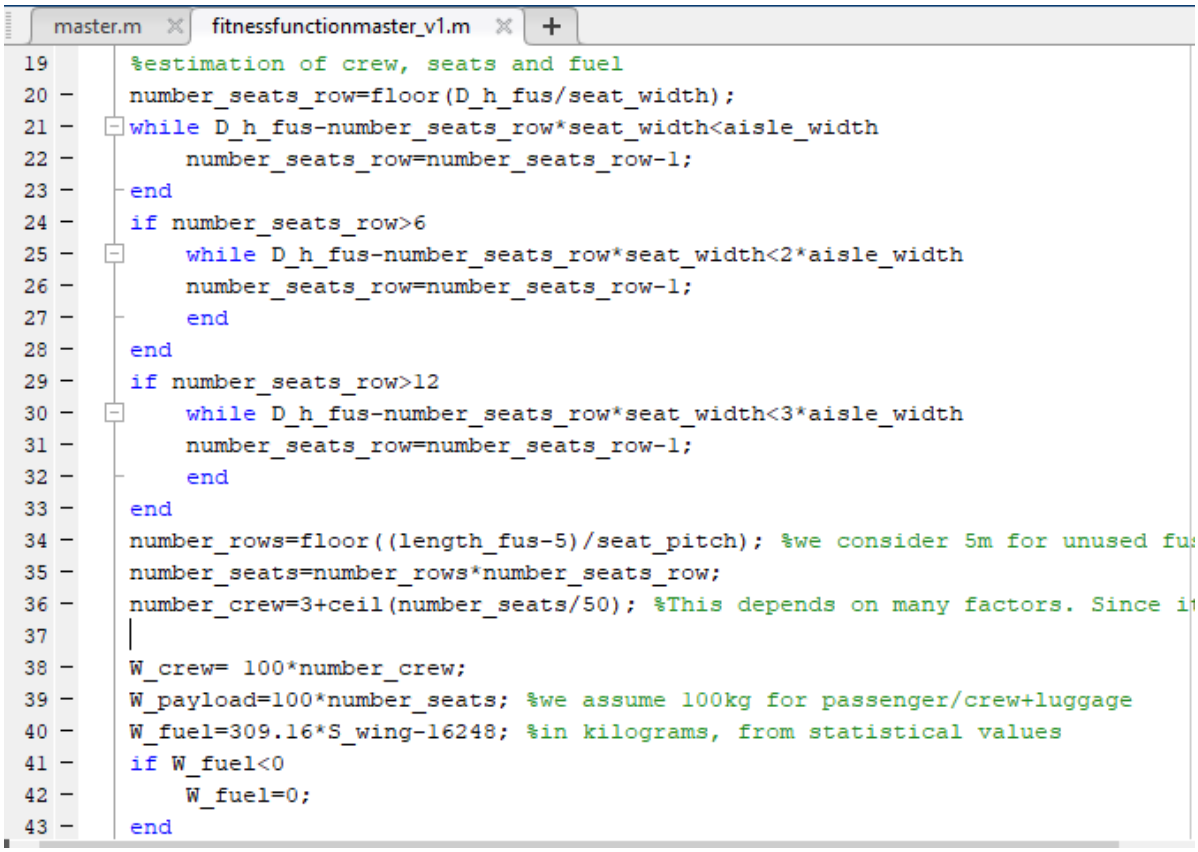


Ilustración 31: fitnessfunctionmaster_v1 imagen 2.

También es necesario estimar la tripulación, asientos y fuel para el cálculo de peso.

Entre las líneas 20 y 35 se hace el cálculo del número de asientos como se explicó en el apartado correspondiente. Recordamos que primero se divide la anchura del fuselaje por el ancho de asientos para obtener el número de asientos por fila (línea 20). Luego se comprueba si hay sitio para un pasillo y, si no es así, se eliminan asientos de la fila hasta que lo haya (en general bastaría con eliminar uno). A continuación, vemos si hay más de 6 asientos por fila, en cuyo caso necesitamos dos pasillos y se eliminarán más asientos de la fila si no hay espacio para el segundo pasillo. Después, se realiza la misma comprobación con más de 12 asientos por si fuera necesario un tercer pasillo. Finalmente se calculan cuántas filas hay de acuerdo a la longitud y distancia entre asientos (línea 34) y se multiplica el número de filas por el número de asientos en cada fila para tener el total de asientos.

En la línea 36 estimamos la tripulación, suponiendo que serán 3 tripulantes más un tripulante más por cada 50 pasajeros. Esta es una estimación general, pero según el nivel de confort que se quiera dar a los pasajeros podría haber más tripulación.

El peso de cada persona más equipaje se estima en 100 kg.

Por último, en la línea 40 se utiliza la recta de ajuste para el peso del fuel en función de la superficie alar. Si resultase un peso de fuel negativo, se iguala a cero. Esta aeronave evidentemente se extinguirá al no tener alcance y, por tanto, un *fitness* muy bajo.

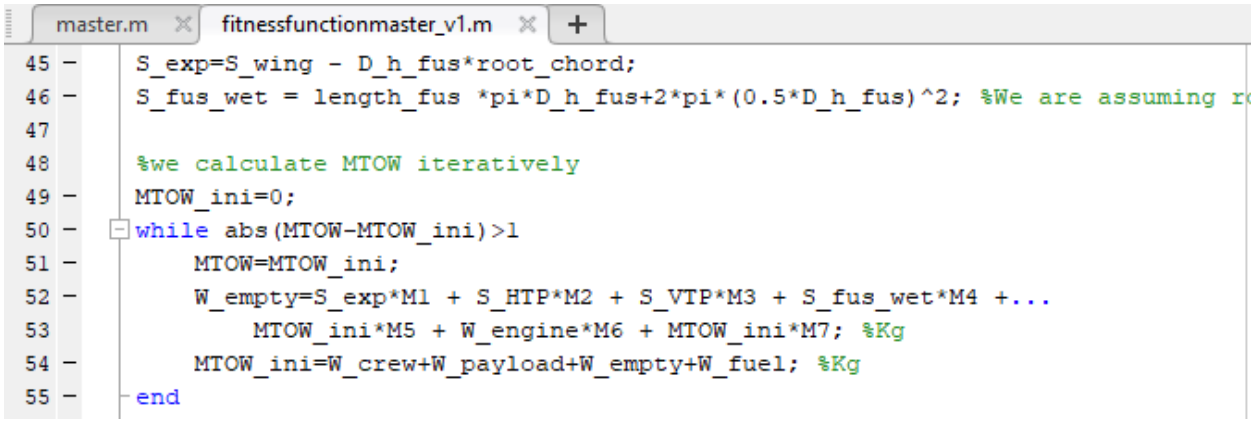


Ilustración 32: fitnessfunctionmaster_v1 imagen 3.

Se calculan los valores de superficie alar expuesta (la superficie alar menos la superficie que está cubierta por el fuselaje) y de área mojada de fuselaje (aproximando a fuselaje cilíndrico).

Con todos estos valores calculados, ya se puede proceder al proceso de cálculo iterativo de MTOW, explicado en su apartado correspondiente. Recordamos que la convergencia de este proceso iterativo está garantizada por el teorema del punto fijo, dado que nuestra función de MTOW = f(MTOW) es contractiva. Además, como su derivada es mucho menor que uno, este proceso es rápido y converge en aproximadamente 3 o 4 iteraciones en las pruebas que hemos llevado a cabo, para una precisión de orden unidad.

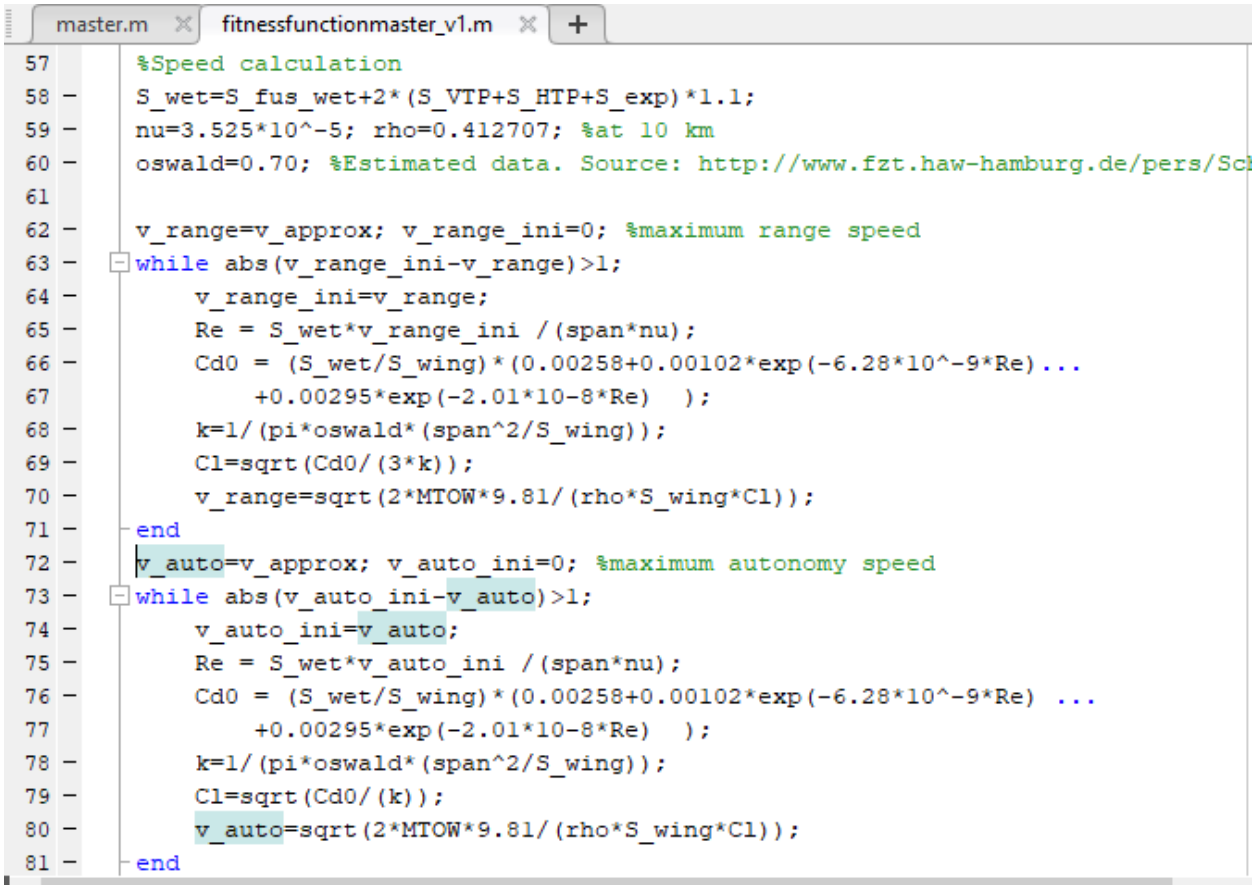


Ilustración 33: fitnessfunctionmaster_v1 imagen 4.

Pasamos al cálculo de la velocidad. En la línea 58 estimamos la superficie mojada como la superficie mojada del fuselaje, más las superficies aerodinámicas (dos veces, extradós e intradós) más un 10% para estimar otras superficies.

Damos valores de las características del aire a una altura de 10 km, y el coeficiente de Oswald típico de aviones de aerolínea [7].

Una vez tenemos esos valores, hacemos dos tipos de cálculo de velocidad, de acuerdo a máxima autonomía y a máximo rango. Son cálculos muy similares, la única diferencia está en el cálculo de $C_{L,}$ que es distinto en ambos casos (líneas 69 y 79).

Los cálculos nuevamente son iterativos de acuerdo con la convergencia garantizada por el teorema del punto fijo, como explicado anteriormente.

master.m | fitnessfunctionmaster_v1.m

```
84      %Cost calculation
85 -    N_test=3; %Approximation of number of test aircraft
86 -    W_empty_pounds= W_empty*2.20462;
87
88 -    engineering_hours=0.023*W_empty_pounds^0.66*v^0.96;
89 -    tooling_hours=0.47*W_empty_pounds^0.64*v^0.5;
90 -    manufacturing_hours=0.35*W_empty_pounds^0.79*v^0.42;
91 -    if flag_cargo==1
92 -        quality_hours=0.12*manufacturing_hours;
93 -    else
94 -        quality_hours=0.085*manufacturing_hours;
95 -    end
96
97 -    engineering_dollars=engineering_hours*20.06*5.8; %5.8 is the value in 2018 c
98 -    tooling_dollars=tooling_hours*18.63*5.8;
99 -    manufacturing_dollars=manufacturing_hours*16.83*5.8;
100 -   quality_dollars=quality_hours*16.83*5.8;
101 -   materials_non_recurring_dollars=0.000024*W_empty_pounds^0.72*v^1.92*5.8;
102 -   materials_recurring_dollars=0.05*W_empty_pounds^0.88*v^0.87*5.8;
103 -   flight_test_dollars=0.13*W_empty_pounds^0.71*v^0.59...
104         *N_test^0.72*flag_cargo^-1.56*5.8;
105
106 -   cost=engineering_dollars+tooling_dollars+manufacturing_dollars+...
107         materials_non_recurring_dollars+materials_recurring_dollars+...
108         flight test dollars+quality dollars;
```

Ilustración 34: fitnessfunctionmaster_v1 imagen 5.

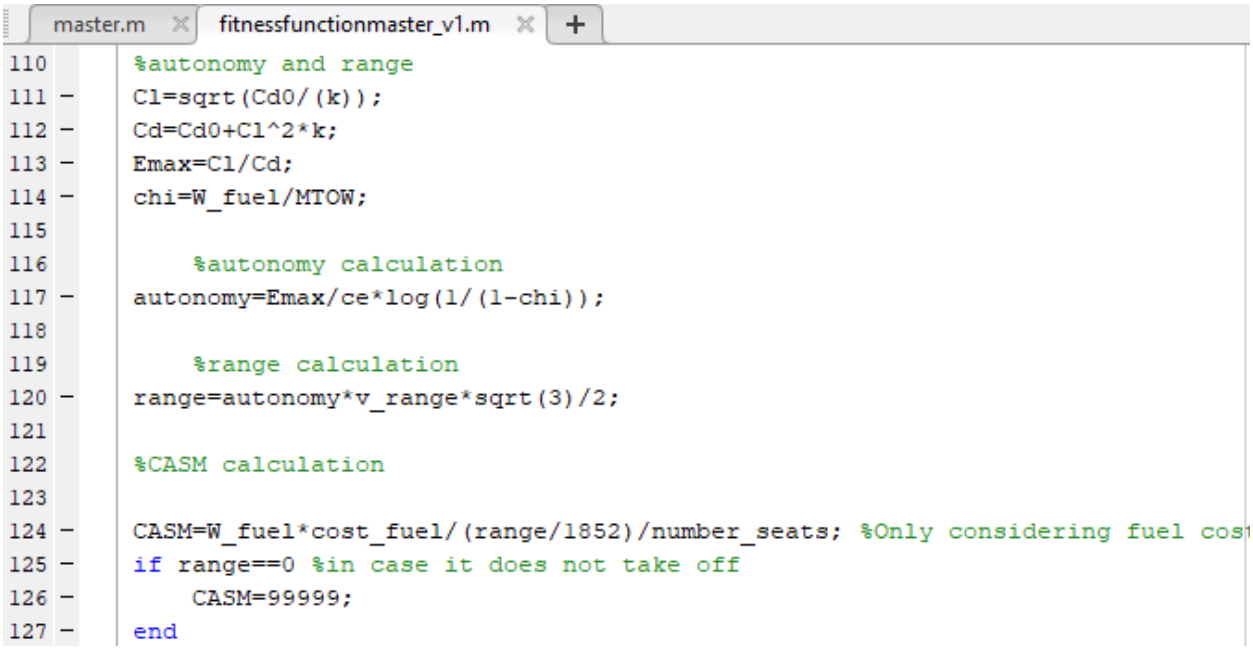
En este apartado se ve el cálculo de coste de acuerdo al modelo RAND, que tiene en cuenta el peso y la velocidad.

En primer lugar, calculamos las horas de trabajo de cada especialidad (líneas 88 a 95).

Entre las líneas 97 y 104 se pasa el cálculo de horas a dólares, teniendo en cuenta la inflación y el coste de horas de cada especialidad.

Por último, se suman todos los costes, más otros costes de test de vuelo y materiales.

Es importante tener en cuenta que este coste es de fabricación de la estructura del avión, no tiene en cuenta elementos como aviónica, motorización, interiores, margen industrial, …



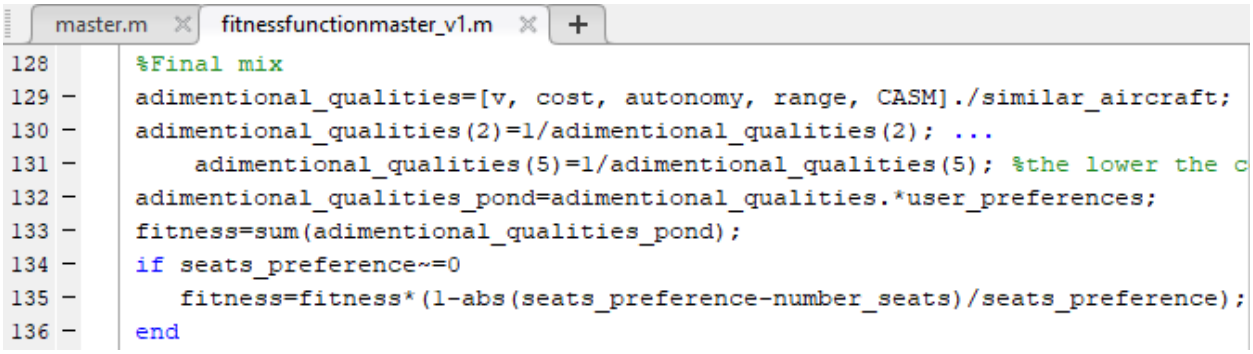
```
110     %autonomy and range
111 -   Cl=sqrt(Cd0/(k));
112 -   Cd=Cd0+Cl^2*k;
113 -   Emax=Cl/Cd;
114 -   chi=W_fuel/MTOW;
115
116         %autonomy calculation
117 -   autonomy=Emax/ce*log(1/(1-chi));
118
119         %range calculation
120 -   range=autonomy*v_range*sqrt(3)/2;
121
122     %CASM calculation
123
124 -   CASM=W_fuel*cost_fuel/(range/1852)/number_seats; %Only considering fuel cost
125 -   if range==0 %in case it does not take off
126 -       CASM=99999;
127 -   end
```

Ilustración 35: fitnessfunctionmaster_v1 imagen 6.

Pasamos al cálculo de autonomía y alcance. En primer lugar, calculamos elementos que serán necesarios para ambos cálculos, como son C_L, C_D, eficiencia aerodinámica máxima, y la relación entre el peso de fuel y MTOW.

Posteriormente se hace el cálculo de acuerdo a las ecuaciones de Breget de alcance y autonomía.

El cálculo de CASM tiene en cuenta el combustible utilizado, su peso, el alcance y el número de asientos. Es importante la comprobación de que, si el alcance es cero, el CASM se convierte en una indeterminación que debe corregirse haciéndolo un valor muy alto (líneas 125 a 127).



```
128     %Final mix
129 -   adimentional_qualities=[v, cost, autonomy, range, CASM]./similar_aircraft;
130 -   adimentional_qualities(2)=1/adimentional_qualities(2); ...
131 -       adimentional_qualities(5)=1/adimentional_qualities(5); %the lower the c
132 -   adimentional_qualities_pond=adimentional_qualities.*user_preferences;
133 -   fitness=sum(adimentional_qualities_pond);
134 -   if seats_preference~=0
135 -      fitness=fitness*(1-abs(seats_preference-number_seats)/seats_preference);
136 -   end
```

Ilustración 36: fitnessfunctionmaster_v1 imagen 7.

Por ultimo, realizamos el cálculo del *fitness* a partir de las prestaciones de la aeronave.

Primero se adimensionalizan respecto a la aeronave similar (línea 129). A continuación, es importante notar que el coste y el CASM se deben invertir, puesto que cuanto menor coste o CASM mayor *fitness* debemos obtener (líneas 130 y 131).

Se multiplican por las preferencias del usuario (línea 132) y se suman para obtener el *fitness*.

Por último, si el usuario ha planteado alguna preferencia de asientos, se penaliza el fitness cuanta más diferencia haya con la cantidad de asientos real (líneas 134 a 136).

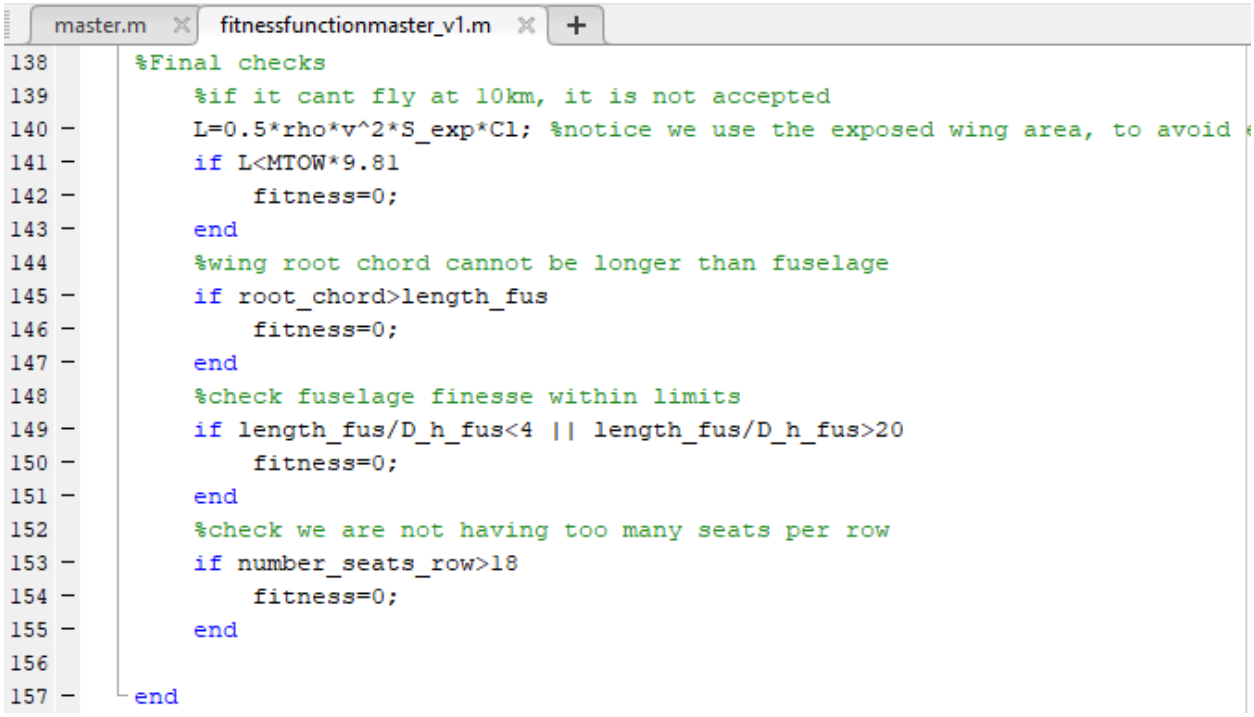


Ilustración 37: fitnessfunctionmaster_v1 imagen 8.

Acabamos la función de *fitness* revisando que la aeronave es aceptable.

La primera comprobación es que el nivel de vuelo de 10 km es aceptable (líneas 140 a 143). Para ello comprobamos que la sustentación sea mayor o igual que el peso.

La comprobación entre las líneas 145 y 147 ha sido mantenida, en la que se comprueba que la cuerda en la raíz es menor que la longitud del fuselaje, pero en la versión actual del código no llega a activarse gracias a la definición de estrechamiento del ala, en contraposición con el código empleado en versiones anteriores.

Se comprueba también que la relación entre longitud y diámetro del fuselaje es aceptable para una aeronave convencional.

Por último, comprobamos que no es una aeronave excesivamente grande, límite que consideramos en 18 asientos por fila. Hacemos esto porque si quisiéramos una aeronave con más asientos, tendría más sentido hacer una aeronave con varios pisos de altura, en lugar de tener todos los asientos en un único piso.

Si cualquiera de las comprobaciones no se cumple, consideramos que la aeronave no es óptima, sin importar el valor de *fitness* que haya obtenido con los cálculos realizados hasta el momento. Para confirmar que esta aeronave se extinguirá, se iguala su *fitness* a 0.

5.3. mixfunctionmaster_v1.m

Esta función es llamada cuando se va a realizar la mezcla de dos padres para dar un elemento de la nueva población.

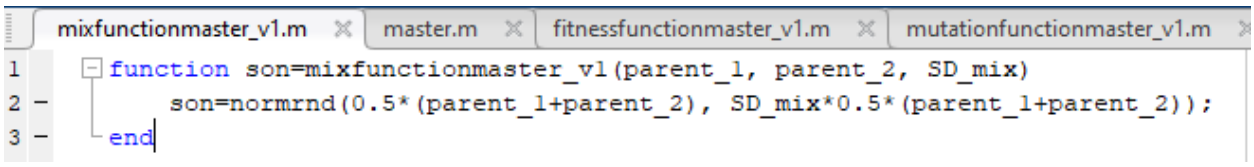


Ilustración 38: mixfunctionmaster_v1.m imagen 1.

Tan solo 1 línea es necesaria para llevar a cabo esta función.

En dicha línea se hace uso de la función *normrnd* para generar un elemento de acuerdo a una probabilidad gaussiana, centrada en la media de los padres y con desviación típica igual a un porcentaje especificado de la media de los padres.

5.4. mutationfunctionmaster_v1.m

Esta es la función empleada para desarrollar la mutación en la nueva población. Es similar a la anterior, aunque ligeramente más compleja.

mutationfunctionmaster_v1.m

```
1     function mutated_element=mutationfunctionmaster_v1...
2         (element, mutation_chance, mutation_SD, genes)
3 -   mutated_element=element;
4 -       for i=1:genes
5 -           if rand(1)<mutation_chance
6 -               mutated_element(i)=normrnd(element(i),mutation_SD*element(i));
7 -           end
8 -       end
9 -   end
```

Ilustración 39: mutationfunctionmaster_v1.m imagen 1.

En primer lugar, se iguala el elemento mutado al elemento en sí, es decir, no se le hace ningún cambio (línea 3).

A continuación, mediante un bucle *for*, para cada uno de los genes se '*echa a suertes'* con un *if* y un número aleatorio (línea 5) si ese gen será o no mutado. En caso de que lo sea, este gen mutado se distribuirá mediante la función *normrnd*, como una gaussiana de media el mismo elemento y desviación típica igual a un cierto porcentaje de dicho elemento.

5.5. stopcriteriamaster_v1.m

Mediante esta función se juzga si se debe o no detener la optimización. Es importante incidir que que es un criterio con parámetros arbitrarios, que deben depender de la potencia de cálculo del ordenador, del tiempo disponible y de la complejidad del problema. Los valores concretos empleados han sido seleccionados mediante experiencia, pero en otros problemas o computadores, deberán variar.

stopcriteriamaster_v1.m | mutationfunctionmaster_v1.m

```
1     function flag_stop=stopcriteriamaster_v1...
2         (generation, fitness_improvement, time)
3 -       if generation>300 && time>180
4 -           if fitness_improvement(end)-fitness_improvement(end-300)==0
5 -               flag_stop=1;
6 -           else
7 -               flag_stop=0;
8 -           end
9 -       else
10 -          flag_stop=0;
11 -      end
12 -  end
```

Ilustración 40: stopcriteriamaster_v1 imagen 1.

Comenzamos decidiendo que mientras haya menos de 300 generaciones o menos de 3 minutos de ejecución, continuamos la optimización (línea 3). El criterio del número de generaciones debe depender del caso concreto, puesto que, incluso el mismo problema, tardará más o menos generaciones en optimizar según los parámetros del código (población, tasa de mutación, ...). El criterio del tiempo dependerá fundamentalmente de la potencia de cálculo del ordenador. Para nuestro equipo (definido en el apartado 3.9) y para el caso en que nos encontramos, la experiencia nos muestra que en menos de 300 generaciones y 3 minutos no se ha llegado al final de la optimización y por lo tanto no debemos detenerla.

Una vez hayamos pasado esa barrera, comprobamos si en las últimas 300 generaciones ha habido mejoras o no. En caso de que no las haya, se establece *flag_stop* igual a 1 para detener la optimización. En caso contrario, continúa siendo cero y la optimización proseguirá. El valor de 300 generaciones es arbitrario, si disponemos de un equipo más potente o más tiempo para hacer los cálculos y queremos un óptimo más ajustado, podemos aumentar dicho número.

6 RESULTADOS Y EJEMPLOS PRÁCTICOS

Vamos a ejecutar el programa para ver cómo se comporta y qué resultados obtenemos.

6.1. Prueba 1 – Aeronave de tamaño medio-alto y largo alcance

Aplicamos primeramente unos parámetros típicos de una aerolínea para vuelos de tamaño medio-alto, en la que daremos importancia a un alcance grande.

En primer lugar, aplicamos como máxima prioridad el CASM, que es la medida de eficiencia con la que generalmente tratan las aerolíneas. También damos una gran importancia al alcance, para poder hacer vuelos de largo alcance. Se da también algo de importancia al coste de la aeronave, una importancia baja a la velocidad y sin importar la autonomía,

Respecto a los asientos, buscamos quedarnos alrededor de los 300.

```
user_preferences=[2, 5, 0, 8, 10];
seats_preference=300;
```

El código se ejecuta durante 672 segundos, llegando a un total de 1656 generaciones. La velocidad media de las generaciones es de 4 décimas de segundo, lo que es bastante alta, por lo que podríamos aumentar el número de elementos de la población si quisiéramos mejorar la exploración del espacio de búsqueda.

Respecto a la aeronave optimizada, presenta los siguientes valores y *fitness*:

best_fitness = 55.9674

best_element = [500.0000 5.1824 3.5651 43.0294 100.0000]

Las alas y envergadura se han ido al máximo permitido. El estabilizador horizontal tiende al mínimo posible -aunque no ha terminado de llegar- mientras que la longitud y anchura del fuselaje sí encuentran un óptimo dentro de los límites permitidos pero alejado de los extremos.

Si ejecutamos este avión en la función *fitness*, podemos ver qué valores alcanzan sus características.

Comenzamos viendo el número de asientos por fila, que es 6. Este es un valor esperado, puesto que los aviones de este tamaño suelen tener la configuración 3+3, con un único pasillo. Con espacio para 50 filas, tenemos un total de, exactamente 300 asientos.

Este número de asientos nos lleva a una estimación de una tripulación de 9 personas. Nuevamente, este valor es razonable, pero depende ampliamente de la aerolínea.

El valor del peso de combustible es el primero que se separa de una aeronave convencional. Casi 140 toneladas para 300 pasajeros es excesivo. Podemos comparar, por ejemplo, con el A350-100, que tiene menos combustible (130 toneladas) aun teniendo una capacidad de pasajeros considerablemente mayor (alrededor de los 400). Este combustible nos dará un alcance más elevado que las aeronaves existentes hoy en día con capacidades similares.

El MTOW se calcula en 270 toneladas, un valor razonable para una aeronave de estas características. Puede ser algo excesivo de acuerdo al número de pasajeros, pero, dada la cantidad de combustible que es capaz de almacenar, tiene un alcance muy alto que justifica este MTOW. El siguiente gráfico desgrana la distribución de dicho MTOW:

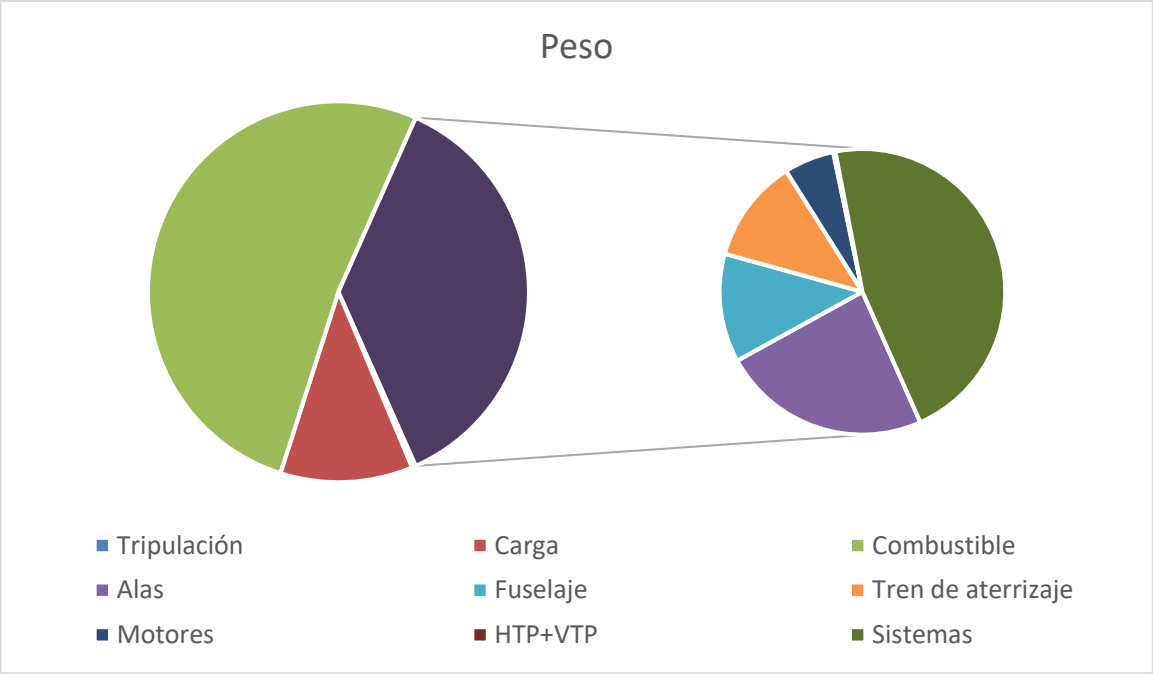


Ilustración 41: Gráfico de la distribución de pesos en la primera prueba

Observamos que el peso de la tripulación es anecdótico (900 kg). La mayor parte del peso viene por el combustible y, en segundo lugar, por la aeronave en sí. Dentro del peso de la aeronave, como era de esperar, una gran parte recae en el fuselaje y las alas. Lo que es menos intuitivo es la gran parte del peso que va a sistemas y al tren de aterrizaje, aunque es la proporción que dicta el método de factores lineales.

Destacamos que el peso de las alas es particularmente grande (tenemos unas alas muy grandes para la aeronave) y que el peso del estabilizador horizontal y vertical es muy pequeño (menos de 240 kg).

Pasando al cálculo de velocidad, vemos que tenemos un C_{D0} bajo pero razonable (0.0096) que nos lleva a una velocidad de 260 m/s, bueno para una aeronave de estas características (aunque algo más rápida de lo normal). La velocidad de máxima autonomía es mucho menor, de 196 m/s.

En cuanto al coste de la estructura, esta asciende a unos 12.5 millones de dólares, distribuido de acuerdo a la siguiente gráfica:

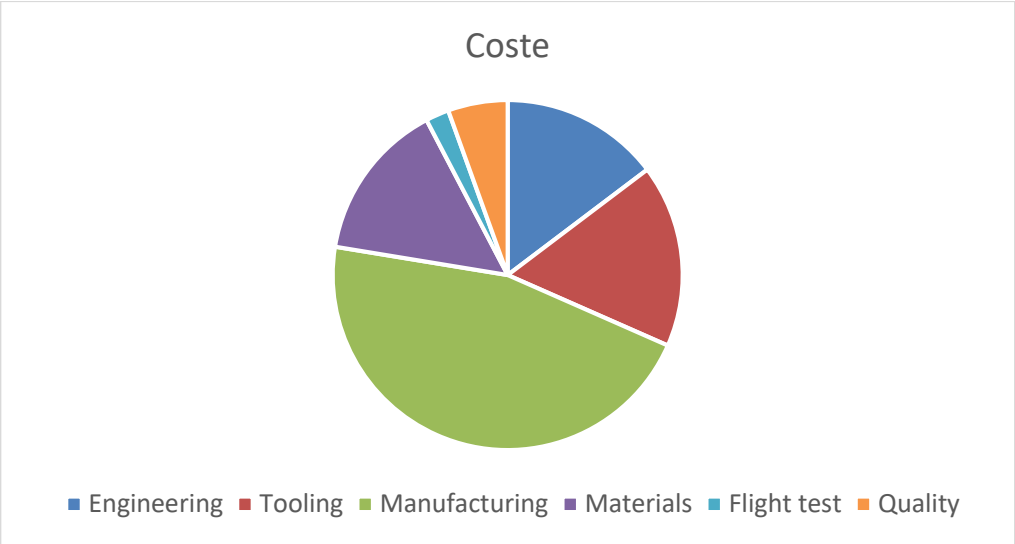


Ilustración 42: Gráfico de distribución de costes en la primera prueba

Comprobar el orden de magnitud de la estructura de costes resulta muy complicado, puesto que no tenemos datos fiables de aeronaves similares. El coste global resulta razonable, mientras que la división de los principales costes en fabricación, en primer lugar, seguido de utillaje, ingeniería y materiales es verosímil.

La autonomía y el rango son muy grandes, gracias a la extensa cantidad de combustible almacenado en las alas. La autonomía son 16 horas mientras que el rango es 13.500 km.

Gracias a este rango tan elevado, el CASM desciende a los 4.35 céntimos de dólar por asiento y milla. Es una eficiencia muy alta, debido a que lo habíamos seleccionado como el criterio más importante en la optimización.

Haciendo cálculos fuera de la función, podemos ver que tendría un crucero óptimo a una altura de aproximadamente 19 km de altitud, bastante por encima de la altura donde convencionalmente vuelan las aeronaves. Esto es debido fundamentalmente al gran tamaño de las alas.

Esta es la aeronave resultado de la primera optimización. Vamos a comprobar, repitiendo múltiples veces la optimización, que siempre se llega al mismo resultado óptimo, es decir, una aeronave con mismas características y mismo *fitness*. Vemos los resultados en la siguiente tabla:

Best_element					Best_fitness
500.0000	5.1824	3.5651	43.0294	100.0000	55.9674
494.9309	5.0894	3.5642	43.0215	100.0000	55.9392
500.0000	5.3734	3.5612	43.0337	100.0000	55.9619
500.0000	5.0000	3.5641	43.1040	100.0000	55.9644
500.0000	5.6276	3.5696	43.0155	100.0000	55.9115
497.9741	5.1961	3.5762	43.0311	100.0000	55.9063
500.0000	5.1266	3.5628	43.0674	100.0000	55.9691
500.0000	5.3741	3.5751	43.0116	100.0000	55.9167
500.0000	5.4080	3.5611	43.0165	100.0000	55.9645
500.0000	5.0000	3.5638	43.0524	100.0000	55.9825

Tabla 11: Resultados de varias optimizaciones en la primera prueba.

Efectivamente, hay muy poca o ninguna diferencia en el resultado final. Esto nos indica que el proceso de optimización ha sido correcto, no quedándose atascado en ningún máximo local ni saliendo de la optimización antes de tiempo.

Ya hemos estudiado la aeronave que ha sido obtenida en la optimización. Ahora podemos también comprobar el camino seguido durante la evolución. En primer lugar, de forma general podemos ver la evolución del *fitness*. En la siguiente gráfica están superpuestas las evoluciones de las 10 pruebas distintas, recortado hasta las 1000 generaciones (las mejoras a partir de dicha generación son demasiado pequeñas para apreciarlas en la gráfica):

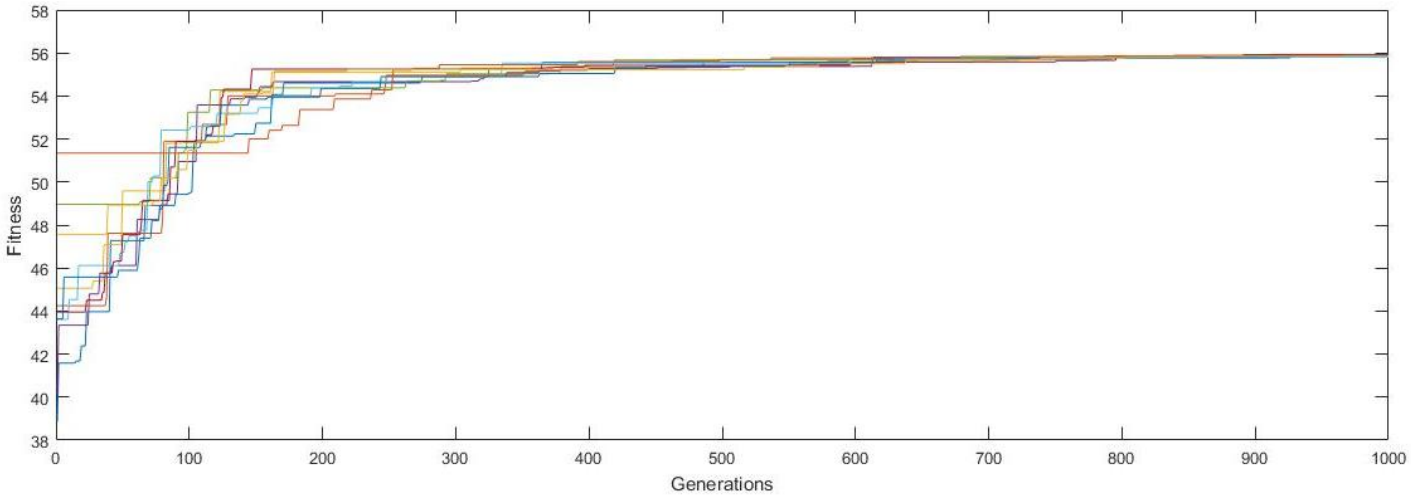
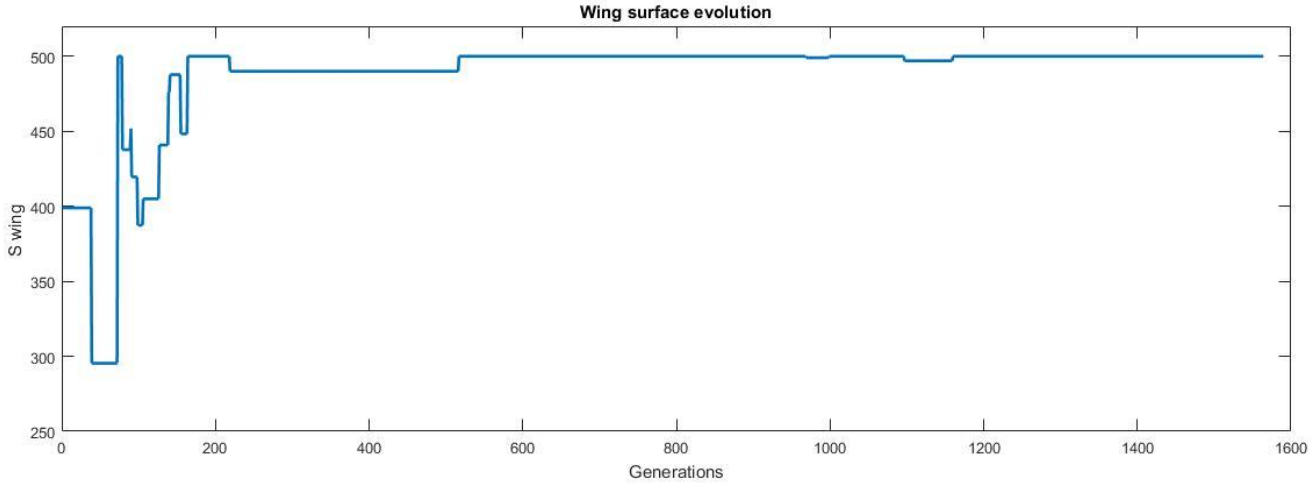
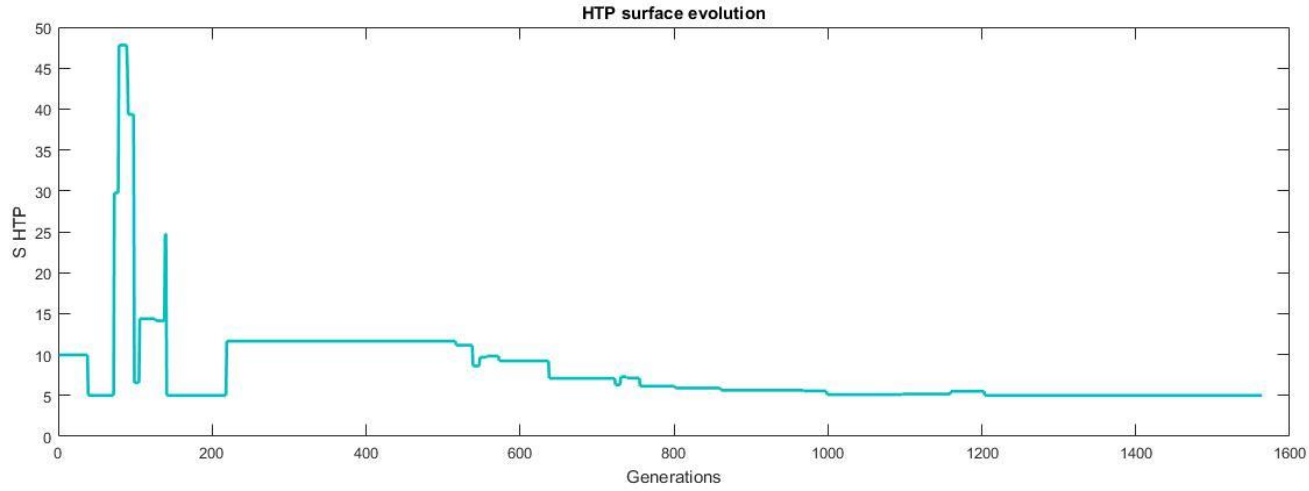


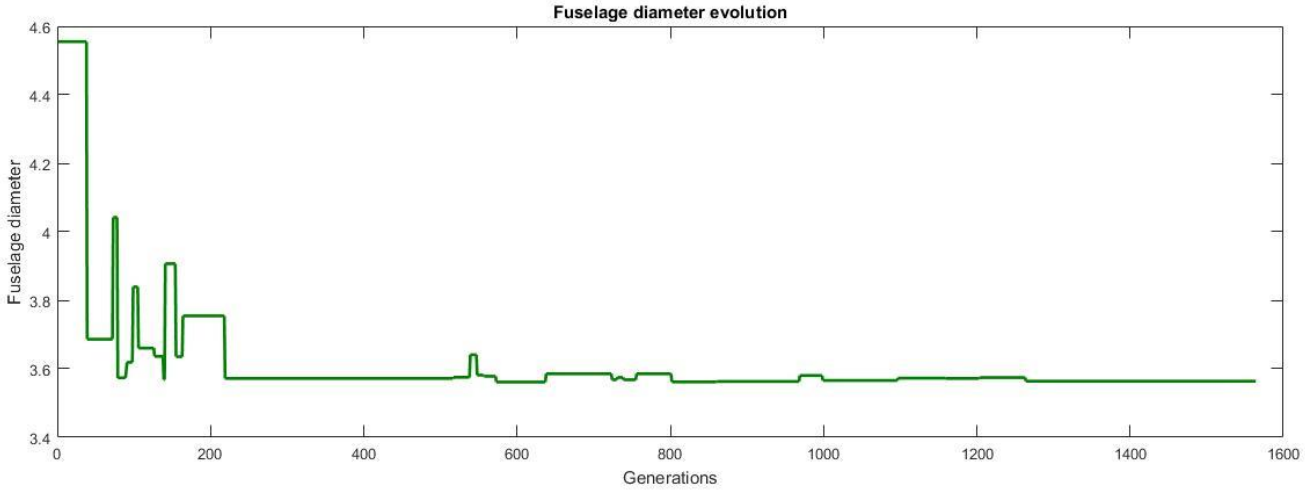
Ilustración 43: Evolución del *fitness* en varias optimizaciones en la primera prueba.

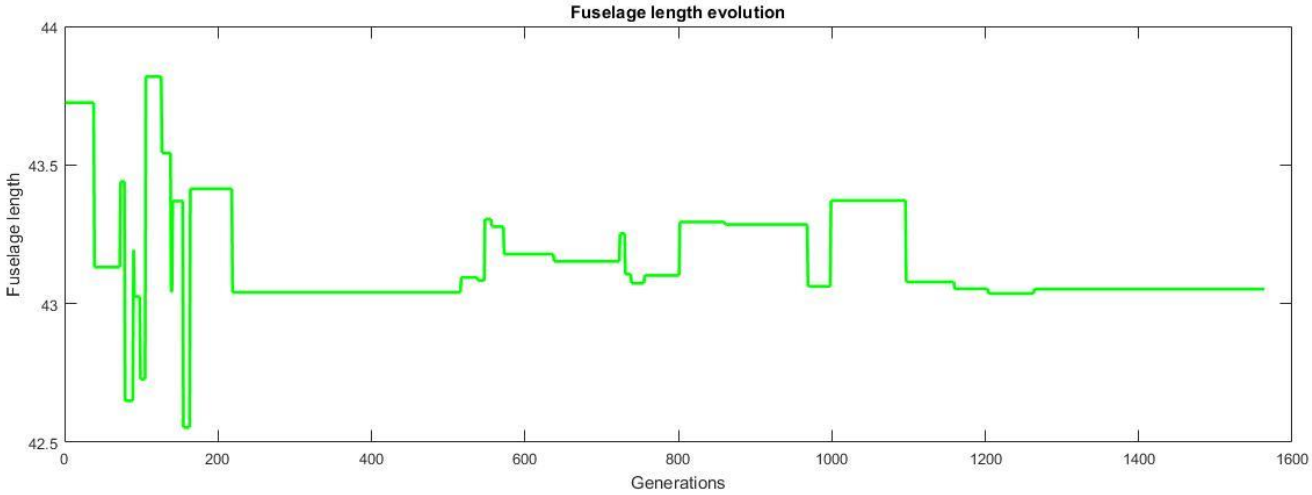
Apreciamos que todas convergen al mismo punto siguiendo una curva similar. Dada la aleatoriedad de la evolución, cada una tiene pequeños escalones en distintos puntos que, de media, siguen un patrón similar. Destacamos que algunas optimizaciones han tenido más suerte que otras respecto al punto de comienzo, dado que en la primera generación tuvieron ya un elemento con un *fitness* elevado. Sin embargo, como en las primeras generaciones se beneficia la exploración respecto a la velocidad, acaban convergiendo con el resto de pruebas.

A continuación, vamos a ver la evolución de las características de la aeronave. En los siguientes gráficos, por simplicidad, mostraremos solamente la evolución de los elementos de una de las optimizaciones. El resto han sido comprobadas y son cualitativamente iguales, aunque los saltos en la evolución sean distintos (de nuevo, por la aleatoriedad de la evolución).




HTP surface evolution
S HTP
Generations
0
5
10
15
20
25
30
35
40
45
50
0
200
400
600
800
1000
1200
1400
1600


Fuselage diameter evolution
Fuselage diameter
Generations
3.4
3.6
3.8
4
4.2
4.4
4.6
0
200
400
600
800
1000
1200
1400
1600


Fuselage length evolution
Fuselage length
Generations
42.5
43
43.5
44
0
200
400
600
800
1000
1200
1400
1600

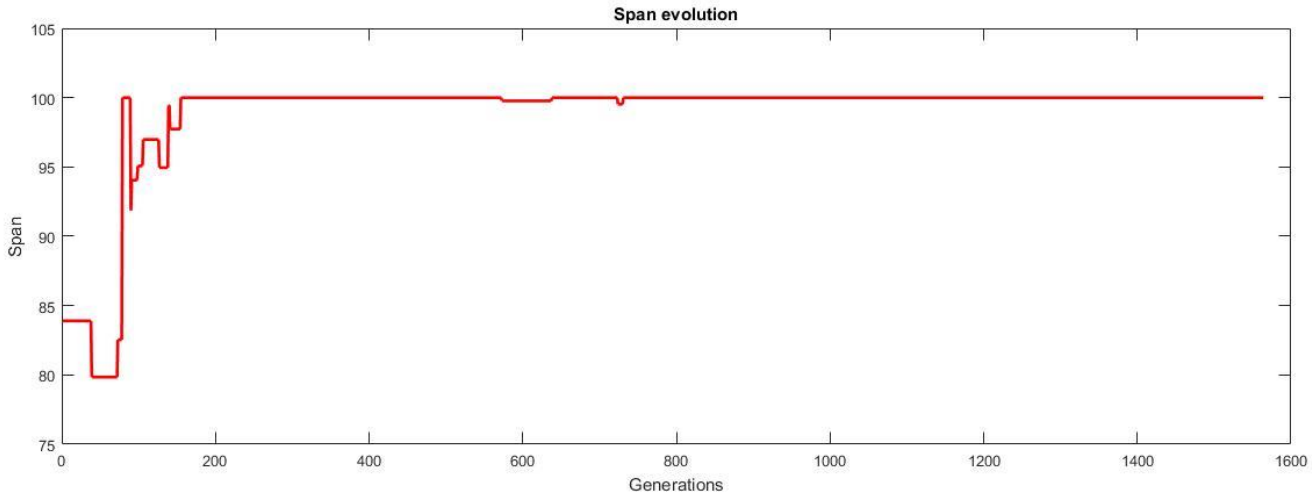


Ilustración 44: Evolución de los parámetros de la aeronave en la primera prueba

Llama la atención la gran variabilidad de estos elementos. Sin embargo, acaban tendiendo a unos valores concretos, que es el óptimo encontrado.

Finalizamos el análisis con unos comentarios sobre los resultados.

En primer lugar, la envergadura no es realista. Su rápida ascensión al máximo permitido será explicada en el apartado de posterior, pero no es aceptable en una aeronave real. Por otro lado, el tamaño de las alas parece excesivo en una aeronave de estas características, aunque si resulta algo más aceptable. El tamaño del estabilizador horizontal tiende al mínimo de una manera más suave, tendencia que también explicaremos.

En general, resulta una aeronave con alas irrealizablemente esbeltas, posiblemente demasiado grandes, y un HTP y VTP demasiado pequeños, capaz de hacer unos cruceros muy largos a bastante velocidad, con una eficiencia por encima de las aeronaves actuales.

6.2. Prueba 2 – Aeronave de tamaño pequeño y corto alcance

A continuación, probamos un enfoque de diseño completamente distinto. Vamos a centrarnos en un transporte regional y esporádico, de pasajeros de negocios, en el que prima ante todo una velocidad alta (para poder ofrecer trayectos muy competitivos en velocidad respecto a otros medios de comunicación en el corto alcance, como pueden ser los trenes de alta velocidad) y un coste de aeronave bajo (dado que es para transportes esporádicos y la aeronave va a estar en tierra mucho tiempo, nos interesa que su coste sea lo más bajo posible). Bajamos al rango de los 20 asientos.

Para el tipo de pasajeros que nos interesa subimos el espacio de pasillo y de ancho de asientos, así como el espacio entre asientos.

```
user_preferences=[8, 10, 0, 0, 2];
seats_preference=20;
```

Aunque hemos dado una importancia de 0 al rango, establecemos un mínimo de 1000 km añadiendo una comprobación en la función de *fitness* que dé valor cero cuando el rango está por debajo del mínimo. De ese modo, conseguimos que tener un rango alto no domine la función *fitness* (no nos interesa un rango alto), pero evolucionamos a aviones que lleguen por encima de los 1000 km.

El código se ejecuta en 1175 generaciones en apenas 265 segundos, dando una velocidad muy alta, de 22 centésimas de segundo por generación. Nuevamente nos indica que el tamaño de población escogido fue algo infravalorado y que podría aumentarse, probablemente mejorando tanto la exploración como la velocidad.

Respecto a la aeronave optimizada, presenta los siguientes valores y *fitness*:

best_fitness = 37.7637

best_element = [55.6063 6.9517 1.9103 12.5193 100.0000]

En esta ocasión, las alas no se van al máximo permitido. Esto es porque habíamos reducido la importancia del rango, por lo tanto, no necesitamos almacenar tanto combustible, mientras que habíamos aumentado la importancia del coste, que hace que nos interese reducir el peso. El estabilizador horizontal vuelve a tender al mínimo posible, mientras que la envergadura va rápidamente al máximo.

Si ejecutamos este avión en la función *fitness*, podemos ver qué valores alcanzan sus características.

Respecto a los asientos, resulta en dos asientos por fila y 10 filas, llegando al número buscado de 20 plazas en total. Para ello, una tripulación pequeña de 4 tripulantes.

Un poco menos de una tonelada de fuel se puede almacenar en las alas. Es una cantidad pequeña, pero suficiente para el corto alcance que buscamos.

La aeronave resulta, como era esperado, pequeña, con un MTOW algo superior a las 17 toneladas. Igual que en el ejemplo anterior, podemos desgranar este resultado en el siguiente gráfico:

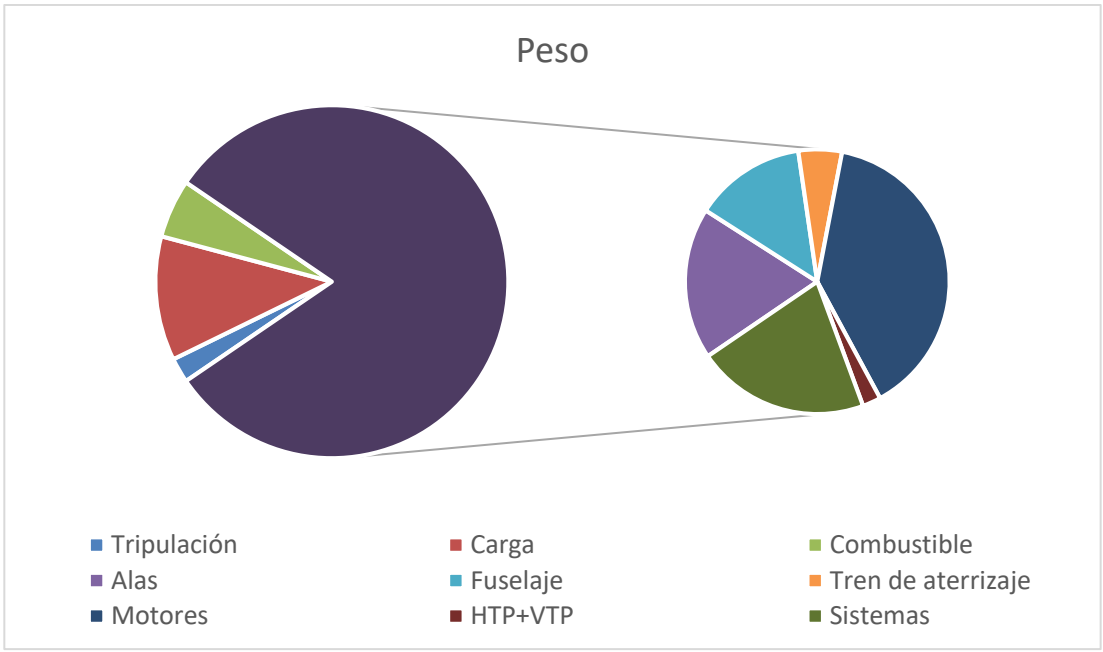


Ilustración 45: Gráfico de distribución del peso de la aeronave en la segunda prueba.

Es una distribución muy distinta a la que habíamos visto previamente. En este caso vemos que el peso de la aeronave domina por completo el MTOW, dejando relegado a un segundo plano la carga y, por detrás, el fuel. Nuevamente, la tripulación es una contribución casi anecdótica.

Respecto a la distribución del peso de la aeronave, está marcado por los motores. Estos motores son excesivamente grandes y potentes para una aeronave tan pequeña. Dado que no estamos evolucionando el tipo de motor, hay que seleccionar de forma manual el peso y consumo específico de los motores adecuados. Los sistemas, el fuselaje y las alas ocupan también posiciones muy relevantes en el peso de la aeronave. Tren de aterrizaje, HTP y VTP tienen pesos bastante bajos.

Llegamos a una velocidad de 105 metros por segundo. Esto es bastante bajo para un avión, y viene debido al bajo peso junto con una altitud no muy elevada. Se podría aumentar la velocidad teniendo un crucero más alto. Sin embargo, por el rango tan bajo, probablemente no sería útil.

El coste sube de 1.7 millones de dólares, distribuido como señala el siguiente gráfico:

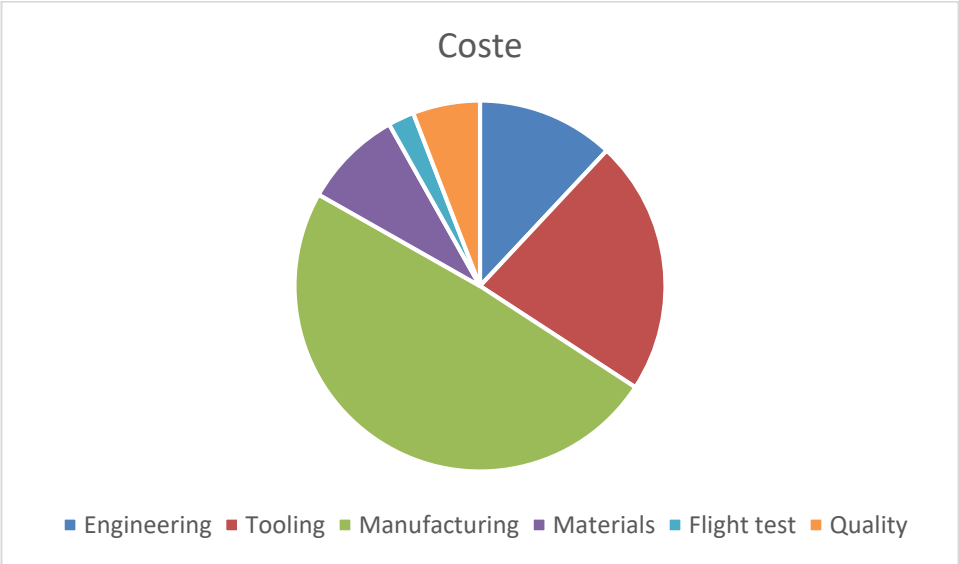


Ilustración 46: Gráfico de distribución de costes en la segunda prueba.

De nuevo, la fabricación domina la distribución de costes. Utillaje e ingeniería también son bastante relevante, dejando a los materiales, test de vuelo y calidad una pequeña parte. Sigue siendo complicado comprobar estos resultados, pero el valor real y los porcentajes de la distribución del coste resultan razonables.

La autonomía es de 3 horas, y el alcance de 1000 km. Ambas son bajas, que era lo que buscábamos para un avión regional. Los 1000 km eran de esperar, puesto que aumentar el rango no mejora el *fitness* y el mínimo impuesto son los 1000 km obtenidos.

El CASM sube a 6 céntimos de dólar por asiento y milla, lo cual es relativamente bajo teniendo en cuenta el bajo número de asientos y de rango, indicando que el consumo de combustible es bajo. Si tuviéramos en cuenta otros costes de operación (tasas aeroportuarias, salarios,) el CASM subiría considerablemente por el bajo número de asientos.

Vamos a hacer funcionar el código para comprobar nuevamente que la evolución no se atasca en máximos relativos y poder ver cómo se produce la evolución.

Best_element					Best_fitness
55.6063	6.9517	1.9103	12.5193	100.0000	37.7637
55.6451	7.6666	1.9198	12.6857	100.0000	37.6003
55.5690	5.7431	1.9173	12.5044	99.9463	37.8741
55.5354	5.0000	1.9227	12.5214	100.0000	37.9426
55.5341	5.5444	1.9021	12.5112	100.0000	37.9484
55.8593	5.0000	1.5341	21.5676	100.0000	36.3205
55.6988	6.2730	1.9069	12.5201	100.0000	37.7944
55.5774	5.1335	1.9228	12.5496	100.0000	37.9016
55.6393	5.0000	1.9128	12.5020	100.0000	37.9291
55.5622	5.0000	1.9156	12.5305	100.0000	37.9458

Tabla 12: Resultados de varias optimizaciones en la segunda prueba

Nuevamente, vemos una homogeneidad en las respuestas con una excepción en la sexta repetición. En dicha optimización se ve que las dimensiones del fuselaje no habían terminado de converger, dando un *fitness* peor que el resto de resultados. Esta discrepancia nos refuerza la importancia de repetir varias veces la misma optimización para comprobar que se llega siempre al mismo óptimo.

A continuación, revisamos la evolución del *fitness* de estas diez optimizaciones:

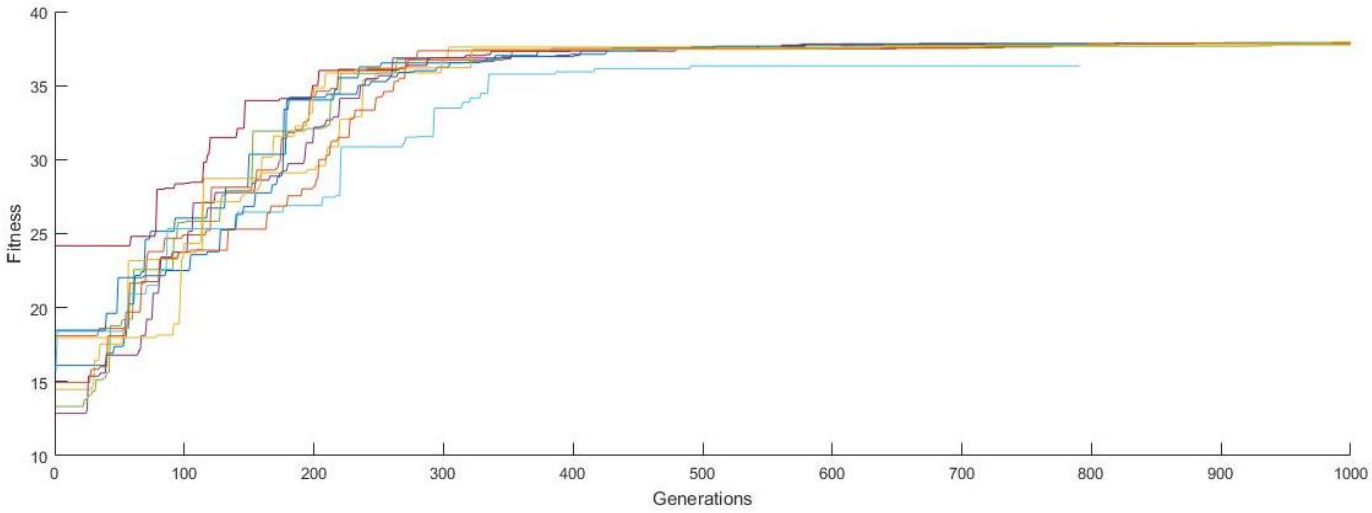
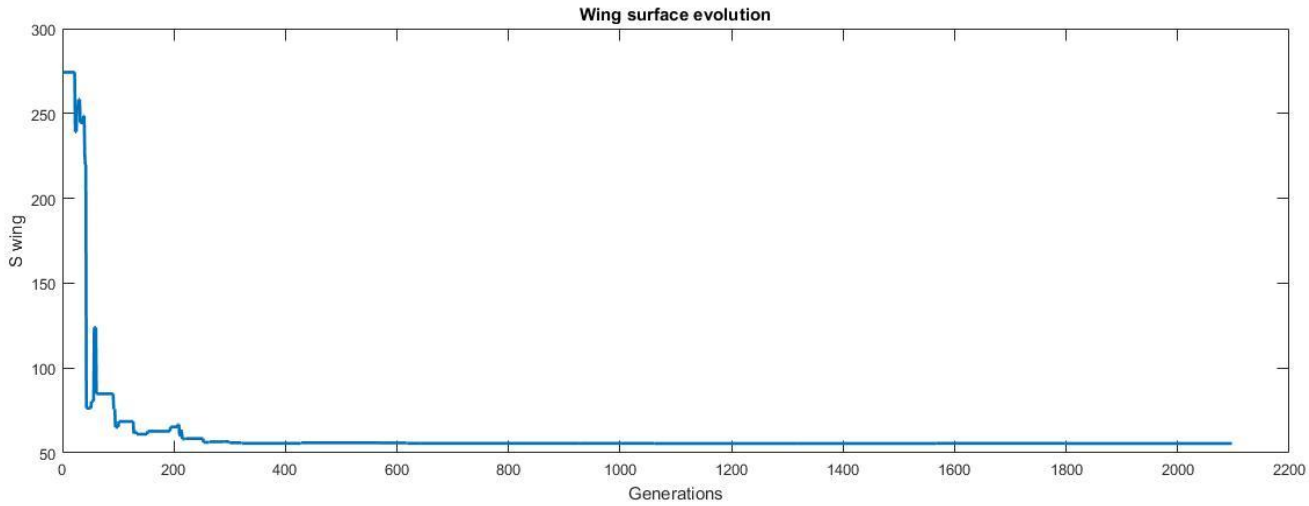
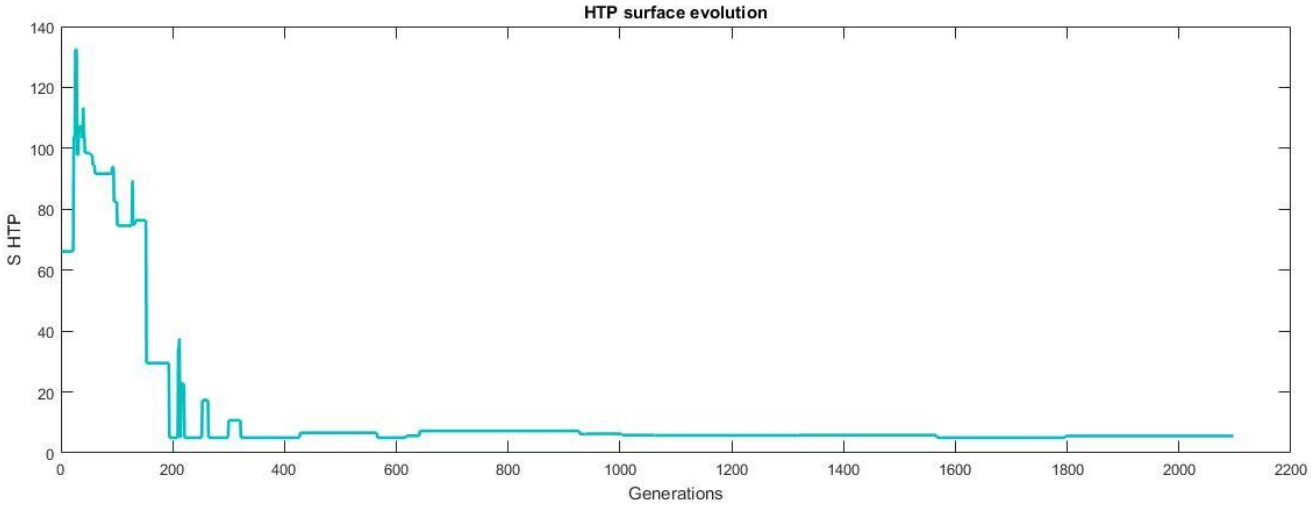


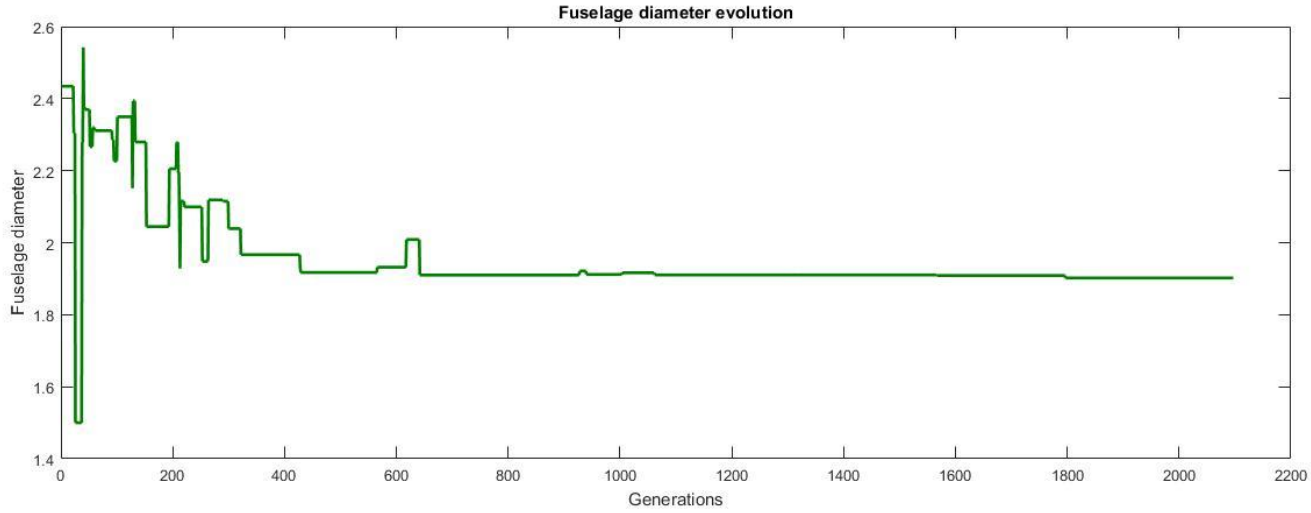
Ilustración 47: Evolución del *fitness* en varias optimizaciones en la segunda prueba.

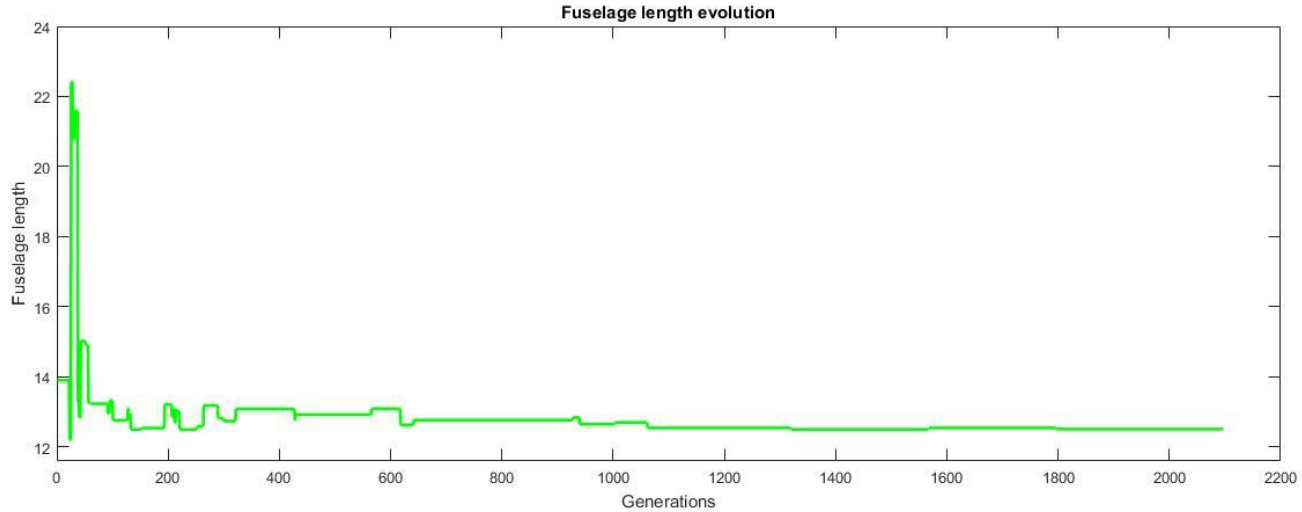
Vemos una curva similar al caso anterior, con saltos aleatorios y con la mayor parte de la variación en las primeras 400 generaciones. Destacamos que la sexta repetición no converge al mismo punto que el resto. Estas anomalías son difíciles de garantizar que no ocurran, y por ello se recomienda que los algoritmos evolutivos se ejecuten varias veces para evitar estos problemas.

Pasamos a ver la evolución de las características de la aeronave:




HTP surface evolution
S HTP
Generations
0
20
40
60
80
100
120
140
0
200
400
600
800
1000
1200
1400
1600
1800
2000
2200


Fuselage diameter evolution
Fuselage diameter
Generations
1.4
1.6
1.8
2
2.2
2.4
2.6
0
200
400
600
800
1000
1200
1400
1600
1800
2000
2200


Fuselage length evolution
Fuselage length
Generations
12
14
16
18
20
22
24
0
200
400
600
800
1000
1200
1400
1600
1800
2000
2200

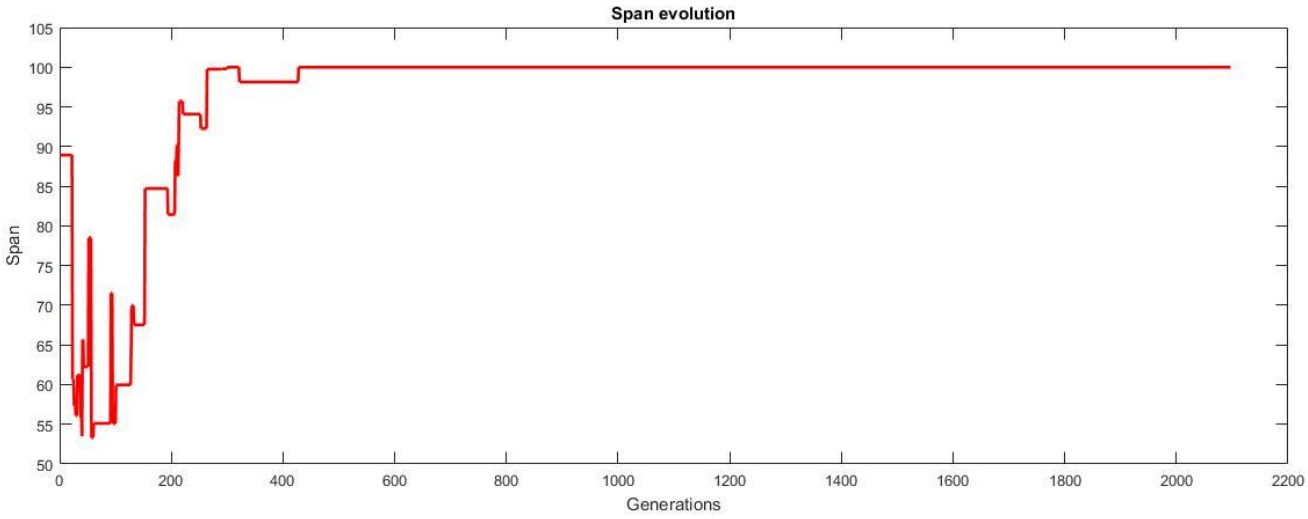


Ilustración 48: Evolución de las características de la aeronave en la segunda prueba.

Vemos unas evoluciones muy similares a las del caso anterior con una alta variabilidad y una tendencia final hacia el óptimo. Finalizamos el apartado con unos comentarios sobre la aeronave diseñada.

En primer lugar, destacamos el bajo precio. Era el principal requisito que se había impuesto y se ha logrado con creces. El segundo era la velocidad. En este resultado encontramos que no logramos tener una velocidad destacable, y la razón para ello es que un aumento de velocidad conlleva un aumento de precio que era el primer requisito. El alcance y capacidad son las especificadas, de 1000 km y 20 pasajeros, respectivamente.

El tamaño del estabilizador horizontal vuelve a tender al mínimo. Igualmente, la envergadura también tiende a un extremo, el máximo. Ambos resultados son poco realistas y serán explicados más adelante.

En general, tenemos una aeronave pequeña, lenta, muy barata, con alas inaceptablemente esbeltas.

Debemos añadir que este tipo de aeronaves suelen tener motores de tipo turbohélice, y volar a unas altitudes más bajas, por lo que sería interesante optimizar evolutivamente esos dos parámetros o, al menos, probar distintas opciones y seleccionar la que dé mejores resultados.

6.3. Comentarios sobre las pruebas

Comenzamos analizando el algoritmo evolutivo en sí. Este algoritmo funciona, dado que encuentra rápidamente el máximo de la función.

Aunque a partir de las 400 generaciones (algo más de un minuto) el *fitness* se encuentra aproximadamente estabilizado y muy cerca de su óptimo, los parámetros particulares de los elementos aún dan variaciones importantes, por lo que hay que ser muy exigente con el criterio de parada.

Hemos comprobado un caso en la segunda prueba donde no se alcanza el óptimo. Es difícil garantizar que esto no vaya a ocurrir nunca, y hay varias formas de tratar con esto. La más simple es repetir la optimización varias veces. Otras pueden ser, en la misma optimización, aumentar de forma importante la mutación y el divisor para incrementar de nuevo la exploración, o incluso resetear parte de la población a valores aleatorios.

Vemos que hay dos valores que no se optimizan, que son la superficie de HTP y la envergadura.

La razón por la que el HTP tiende siempre al mínimo es porque, a mayor superficie de HTP, mayor peso. El aumento de peso es negativo para las prestaciones, y por ello resulta minimizado. En un avión real, el HTP sirve para la estabilización del avión, y por ello siempre requiere un cierto tamaño, pero el criterio de estabilidad no está incluido en la función de *fitness*.

Respecto a la envergadura, comprobamos que siempre tiende al máximo. Esto es porque, a mayor envergadura, mayor esbeltez del ala, y esto influye en reducir la resistencia aerodinámica, lo cual siempre es positivo. Sin embargo, en un avión real un aumento de la esbeltez lleva consigo la necesidad de materiales de mayor calidad (y, por tanto, incrementando el coste) o bien la necesidad de refuerzos (con el consiguiente aumento de peso, lo cual lleva a una pérdida de prestaciones), generalmente una mezcla de ambas. La función *fitness* planteada no incluye consideraciones sobre el aumento de coste o precio con la esbeltez, y es por ello que siempre tiende al máximo.

Tenemos por tanto una función *fitness* incompleta, donde si queremos obtener resultados útiles sobre HTP y envergadura tenemos que añadir las consideraciones explicadas. Esto no resulta inmediato, en particular el estudio sobre los costes en dinero y peso del aumento de esbeltez.

Otro problema encontrado resulta en el tamaño de las alas. Si se le da importancia al rango, y dado que la función de *fitness* considera que todo el combustible se almacena en las alas, acabamos optimizando a unas alas inaceptablemente grandes. Una solución a este problema sería incluir en la optimización la posibilidad de ubicar tanques de combustible en el fuselaje. Estos tanques permitirían reducir el tamaño de las alas (y por tanto el peso), aunque obligarían a aumentar el tamaño del fuselaje o reducir el número de asientos disponibles. El algoritmo evolutivo debería ser capaz de encontrar el óptimo a este problema.

Tambien, observamos que las características del fuselaje se optimizan de manera muy adecuada.

Los resultados en prestaciones son coherentes con los requerimientos establecidos y con las aeronaves optimizadas.

7 CONCLUSIONES

Finalizamos el trabajo revisando las expectativas del inicio con lo finalmente logrado.

Hemos hecho un estudio en profundidad de los algoritmos de optimización por estrategia evolutiva. Se han explicado distintas técnicas, justificando las empleadas y con dos ejemplos muy amplios que validan la programación realizada.

El repaso del proceso de diseño de aeronaves ha sido breve, empleado fundamentalmente para ubicar la utilidad del programa realizado.

Respecto al código creado, la optimización funciona. El *fitness* llega al máximo posible, sin atascarse en máximos relativos ni salir de la optimización antes de tiempo. Sin embargo, nos hemos encontrado con que la función de *fitness* creada no es suficientemente extensa, y le faltan análisis respecto a las superficies estabilizadoras y de envergadura, y ello lleva a que esos elementos no se optimicen correctamente.

Las mejoras de este trabajo deben ir encaminadas a mejorar la función de *fitness*, y las opciones sugeridas se resumen en el siguiente apartado.

7.1. Continuaciones y mejoras

A continuación, establecemos una pequeña lista con las mejoras más inmediatas de las que se podría beneficiar el código realizado:

- Incluir optimización del nivel de vuelo. Esto permitiría a aeronaves con alas más grandes volar a mayor altura, donde la baja densidad reduce la resistencia y, por tanto, consumo de combustible.
- Incluir optimización del tipo de motor y número de motores. Esto permite ganar velocidad a cambio de mayor número de motores o motores más potentes, a cambio de un incremento en el consumo. Por lo tanto, es útil cuando la velocidad prima por encima de la economía. Además, permitiría tener un motor de un tamaño ajustado a la aeronave, de forma que funcione en el rango óptimo de empuje.
- Incluir optimización de tanques de combustible. Cuando interesa una aeronave barata, pero con un rango amplio, reducir alas añadiendo tanques de combustible nos permite conseguir ambos efectos.
- Mejorar la mezcla de parámetros, dado que en ocasiones una sola de las características puede dominar la función de *fitness* como, por ejemplo, ocurrión en el problema coste-velocidad en la segunda prueba realizada.
- Incluir criterios de estabilidad longitudinal para la correcta optimización de la superficie del HTP.
- Incluir un estudio del aumento de peso y coste con el aumento de envergadura para la correcta optimización de la envergadura.
- Incluir criterios de estabilidad lateral para la correcta optimización de la superficie del VTP.

8 BIBLIOGRAFÍA

[1] - Damián Rivas - Tema 4: Actuaciones integrales. - Apuntes mecáncia de vuelo, Universidad de Sevilla, ETSI, departamento de ingeniería aeroespacial y mecánica de fluidos.

[2] - Ajoy Kumar Kundu. - Aircraft design. - Cambridge aerospace Series.

[3] - R.W.Hess, H.P.Romanoff. - Aircraft Airframe Cost Estimating Relationships, Study Approach and Conclusions. - R-3255-AF. - December 1987.

[4] - Gerard W. H. van Es. - Rapid Estimation of the Zero-Lift Drag Coefficient of Transport Aircraft. - Journal of aircraft Vol.39 No.4 - July-August 2002.

[5] - Joseph P. Large, Harry G. Campbell, David Cates. - Parametric Equations for Estimating Aircraft Airframe Costs. - R-1693-1-PA&E. - Febrero 1976.

[6] - Airbus SAS. - All about the A320 Family, Technical appendices. - Mayo 2002.

[7] - M. Nita, D. Scholz. - Estimating the Oswald factor from basic aircraft geometrical parameters, documentID 281424. - Hamburg University of applied sciences Aero, Aircraft Design and Systems Group Berliner Tor 9, 20099 Hamburg, Germany. - 2012.

[8] - Ricardo Sánchez Pastor. - Desarrollo de una herramienta académica para el estudio de la estimación estructural de aeronaves mediante una interfaz gráfica basada en Matlab: AStr.gui. - Proyecto de fin de carrera, Universidad de Sevilla, ETSI, ingeniería aeronáutica. - 2015.

[9] - Sergio Esteban Roncero. - Tema 3: Dimensionado preliminar de aeronaves. - Apuntes cálculo de aeronaves, Universidad de Sevilla, ETSI, Departamento de Ingeniería Aeroespacial Y Mecánica de Fluidos

[10] - Sergio Esteban Roncero. - Tema 13: Estructuras Detalladas Métodos Estimación Pesos. - Apuntes cálculo de aeronaves, Universidad de Sevilla, ETSI, Departamento de Ingeniería Aeroespacial Y Mecánica de Fluidos

[11] - Oliver Kramer. - Genetic algorithms essentials. - Studies in Computational Intelligence Volume 679. – 2017.

[12] - Mitsuo Gen, Runwei Cheng, Lin Lin. – Network Models and Optimization, Multiobjectiv Genetic Algorithm Approach. – Decision engineering Series ISSN 1619-5736 – 2008.

[13] - Peter John Bentley B.Sc. (Hons). - Generic evolutionary design of solid objects using a genetic algorithm. - Thesis submitted to the University of Huddersfield. – November 1996.

[14] - Aerliner price index – Flight International page 183 – 10 august 1976

[15] - Página web oficial de Boeing: https://www.boeing.com/

[16] - Página web official de Airbus: https://www.airbus.com/

9 ANEXOS

9.1. Código de optimización de funciones trigonométricas

9.1.1. Función f(x)=sin(x)

```
clc
clear all
tic
%HIGH LEVEL DESCRIPTION
%Este programa busca el máximo de la funcion f=x*sin(x) en el intervalo
%[0,100]
%El máximo es 95.823793608465


%CODE PARAMETERS
N=10000; %number of elements in the population
genes = 1; %number of genes in an element, esta funcion solo tiene un
parametro, por lo tanto solo un gen con el que evolucionar
mutation_chance=0.05; %chance of a random mutation
mutation_SD=5;
mix_SD=0.00005;
divisor=10;
salida=0.9999999;
mutation_rate_change=0.05;

%OPTIMIZATION ALGORITHM

    %creation of original population
    best_element=zeros(1,genes); %inicializamos para que los loop sean mas
rapidos
    best_fitness=0;%inicializamos para que los loop sean mas rapidos
    population=zeros(N,genes);%inicializamos para que los loop sean mas
rapidos
    for i=1:N
        population(i,:)=rand(1,genes)*100; %Creamos poblacion entre 0 y 100
    end
    generation=1
    while best_fitness<salida*95.823793608465 %este es el maximo, mientras no
lleguemos seguimos con el loop
        %fitness calculation
        fitness=zeros(N,1);
        for i=1:N
           fitness(i,1)=fitnessfunction1(population(i,:)); %la fitness
function es la funcion en si, sin(x)*x
        end
        [max_fitness, position_max]=max(fitness); %guardamos el maximo
        if max_fitness>best_fitness
            best_fitness=max_fitness;
            best_element=population(position_max,:);
```

```
        end
        fitness_improvement(generation)=best_fitness; %util utilizar
plot(fitness_improvement) para ver como mejora el fitness con las
generaciones
        worst_fitness(generation)=min(fitness);
        success=zeros(1,50);
        %creation of mating pool
        for i=1:N
            if abs(fitness(i))<0.001
                fitness(i)=0.001; %no se aceptan ceros
            end
        end
        mating_pool_number=fitness+(abs(min(fitness)))+1;
        mating_pool_number=mating_pool_number.^(1+generation/divisor);
        mating_pool_number=round(mating_pool_number.*100);
        if sum(isinf(mating_pool_number))>=1
            divisor=generation;
            for i=1:length(mating_pool_number)
                if isinf(mating_pool_number(i))==1
                    mating_pool_number(i)=1;
                else
                    mating_pool_number(i)=0;
                end
            end
        end
%
mating_pool_number=abs(fitness./min(abs(fitness))).^(generation/divisor);
%
mating_pool_number=round(mating_pool_number.*100).*(fitness./abs(fitness));
%
mating_pool_number=mating_pool_number+abs(min(mating_pool_number))+1;
        while sum(mating_pool_number)>100*N
            mating_pool_number=round(mating_pool_number./3);
        end
        mating_pool=zeros(sum(mating_pool_number),genes);
        k=1;
        for i=1:N
            for j=1:mating_pool_number(i,1)
                mating_pool(k,:)=population(i,:);
                k=k+1;
            end
        end

        %creation of new population
            %mix
        new_population=zeros(N,genes);
        for i=1:N %la mezcla de padres es la media
            parent_1=mating_pool(ceil(rand(1)*length(mating_pool)),:);
            parent_2=mating_pool(ceil(rand(1)*length(mating_pool)),:);
            new_population(i,:)=mixfunction1(parent_1, parent_2, mix_SD);
            %new_population(i,:)=mixfunction(parent_1, parent_2);
        end
        population=new_population;
            %mutation
                %success measure
                if generation>50
                    for i=1:49
                        success(i)=success(i+1);
                    end
                    if fitness_improvement(end)-fitness_improvement(end-1)>0
                        success(50)=1;
                    else
```

```
                        success(5)=0;
                    end
                    %change mutation
                    if sum(success)==0 && mutation_SD>0.0001 ;
                        mutation_SD=mutation_SD*(1-
mutation_rate_change);%reducir
                    end
                    if sum(success)==0
                        divisor=divisor*0.99;
                    end
                    if sum(success)>1;

mutation_SD=mutation_SD*(1+mutation_rate_change);%aumentar
                    end
                end
        %mutation
        for i=1:N

population(i,:)=mutationfunction1(population(i,:),mutation_chance,
mutation_SD); %la mutacion es distinta al principio (mucha variacion para
explorar el espacio de busqueda) que al final (poca variacion para afinar la
solucion)
            %population(i,:)=mutationfunction(population(i,:),mutation_rate);
        end
        for i=1:N
            if population(i,:)>100 %esto ultimo es para asegurarnos que al
mutar elementos no se nos salen del espacio (0, 100)
                population(i,:)=100;
            end
            if population(i,:)<0
                population(i,:)=0;
            end
        end
        population(end,:)=best_element;
        generation=generation+1;
    end

%FINAL SOLUTION

generation
best_fitness;
best_element
toc
```

```
function fitness=fitnessfunction1(element)
    fitness=element*sin(element);
end
```

```
function son=mixfunction1(parent_1, parent_2, SD)
son=normrnd(0.5*(parent_1+parent_2), SD);
end
```

```
function mutated_element=mutationfunction1(element, mutation_chance,
mutation_SD)
```

```
mutated_element=element;
if rand(1)<mutation_chance
   mutated_element=normrnd(element,mutation_SD);
end
end
```

9.1.2. Función f(x,y)=x·y·cos(x·y)

```
clc
clear all
tic
%HIGH LEVEL DESCRIPTION
%Este programa busca el máximo de la funcion f=x*y*cos(x*y) en el intervalo
%[-5,5]
%El máximo es 24.780070296586839, para -5, -5
%eso lo resuelve muy facil porque el maximo está en el extremo. Probamos
%con -6.
%el maximo es 31.431827278534620 en x=-5.6747 y=-5.541740470000001




%CODE PARAMETERS
N=10000; %number of elements in the population
genes = 2; %number of genes in an element, esta funcion solo tiene un
parametro, por lo tanto solo un gen con el que evolucionar
mutation_chance=0.05; %chance of a random mutation
mutation_SD=5;
mix_SD=0.00005;
divisor=10;
salida=0.9999999;
mutation_rate_change=0.05;

%OPTIMIZATION ALGORITHM

    %creation of original population
    best_element=zeros(1,genes); %inicializamos para que los loop sean mas
rapidos
    best_fitness=0;%inicializamos para que los loop sean mas rapidos
    population=zeros(N,genes);%inicializamos para que los loop sean mas
rapidos
    for i=1:N
        population(i,:)=rand(1,genes)*11-6; %Creamos poblacion entre -6,5
    end
    generation=1;
    while best_fitness<salida*31.431827278534620 %este es el maximo, mientras
no lleguemos seguimos con el loop
        %fitness calculation
        fitness=zeros(N,1);
        for i=1:N
           fitness(i,1)=fitnessfunction2(population(i,:)); %la fitness
function es la funcion en si, sin(x)*x
        end
        [max_fitness, position_max]=max(fitness); %guardamos el maximo
        if max_fitness>best_fitness
            best_fitness=max_fitness;
            best_element=population(position_max,:);
        end
```

```
        fitness_improvement(generation)=best_fitness; %util utilizar
plot(fitness_improvement) para ver como mejora el fitness con las
generaciones
        worst_fitness(generation)=min(fitness);
        success=zeros(1,50);
        %creation of mating pool
%         for i=1:N
%             if abs(fitness(i))<0.001
%                 fitness(i)=0.001; %no se aceptan ceros --> ahora si
%             end
%         end
        mating_pool_number=fitness+(abs(min(fitness)))+1;
        mating_pool_number=mating_pool_number.^(1+generation/divisor);
        mating_pool_number=round(mating_pool_number.*100);
        if sum(isinf(mating_pool_number))>=1
            divisor=generation;
            for i=1:length(mating_pool_number)
                if isinf(mating_pool_number(i))==1
                    mating_pool_number(i)=1;
                else
                    mating_pool_number(i)=0;
                end
            end
        end
%
mating_pool_number=abs(fitness./min(abs(fitness))).^(generation/divisor);
%
mating_pool_number=round(mating_pool_number.*100).*(fitness./abs(fitness));
%
mating_pool_number=mating_pool_number+abs(min(mating_pool_number))+1;
        while sum(mating_pool_number)>100*N
            mating_pool_number=round(mating_pool_number./3);
        end
        mating_pool=zeros(sum(mating_pool_number),genes);
        k=1;
        for i=1:N
            for j=1:mating_pool_number(i,1)
                mating_pool(k,:)=population(i,:);
                k=k+1;
            end
        end

        %creation of new population
            %mix
        new_population=zeros(N,genes);
        for i=1:N %la mezcla de padres es la media
            parent_1=mating_pool(ceil(rand(1)*length(mating_pool)),:);
            parent_2=mating_pool(ceil(rand(1)*length(mating_pool)),:);
            new_population(i,:)=mixfunction2(parent_1, parent_2, mix_SD);
            %new_population(i,:)=mixfunction(parent_1, parent_2);
        end
        population=new_population;
            %mutation
                %success measure
                if generation>50
                    for i=1:49
                        success(i)=success(i+1);
                    end
                    if fitness_improvement(end)-fitness_improvement(end-1)>0
                        success(50)=1;
                    else
                        success(50)=0;
                    end
```

```
                    %change mutation
                    if sum(success)==0 && mutation_SD>0.001 ;
                        mutation_SD=mutation_SD*(1-
mutation_rate_change);%reducir
                    end
%                     if sum(success)==0
%                         divisor=divisor*0.99;
%                     end
                    if sum(success)>1;

mutation_SD=mutation_SD*(1+mutation_rate_change);%aumentar
                    end
                end
        %mutation
        for i=1:N

population(i,:)=mutationfunction2(population(i,:),mutation_chance,
mutation_SD); %la mutacion es distinta al principio (mucha variacion para
explorar el espacio de busqueda) que al final (poca variacion para afinar la
solucion)
            %population(i,:)=mutationfunction(population(i,:),mutation_rate);
        end
        for i=1:N
            for j=1:2
                if population(i,j)>5 %esto ultimo es para asegurarnos que al
mutar elementos no se nos salen del espacio (0, 100)
                    population(i,j)=5;
                end
                if population(i,j)<-6
                    population(i,j)=-6;
                end
            end
        end
        population(end,:)=best_element;
        generation=generation+1;
    end

%FINAL SOLUTION

generation;
best_fitness;
best_element;
toc
```

```
function fitness=fitnessfunction2(element)
    fitness=element(1)*element(2)*cos(element(1)*element(2));
end
```

```
function son=mixfunction2(parent_1, parent_2, SD)
son=normrnd(0.5*(parent_1+parent_2), SD);
end
```

```
function mutated_element=mutationfunctionmaster_v1(element, mutation_chance,
mutation_SD)
mutated_element=element;
```

```
if rand(1)<mutation_chance
   mutated_element=normrnd(element,mutation_SD);
end
end
```

9.2. Código de optimización de diseño preliminar de aeronaves

```
%HIGH LEVEL DESCRIPTION
%This codes optimices the preliminary design of an aircraft using evolution
%strategy. User must change in the USER PARAMETERS section.

%CHANGES
%eliminate c_body by using tap ratio, span and S_wing
%corrected negative fuel weight
%corrected inadequate wing surface

clc
clear all
tic

%USER PARAMETERS

parameter_max=[500, 250, 10, 100, 100]; %maximum value for: S_wing, S_HTP,
D_h_fus, length_fus, span, in m or m^2
parameter_min=[50,5,1,20,10]; %minimum value for: S_wing, S_HTP, D_h_fus,
length_fus, span
flag_cargo =1 ; %1 if aircraft is NOT cargo, 2 if aircraft IS cargo
ce=2*0.744/36000*9.81; %specific fuel consumption in 1/s, that unit is needed
for equations
W_engine=2*2150;  %engines weight kg. Enginges from JT8D-7, typical for
airlines
taper_ratio=0.3; %tip chord divided by root chord. Typical in airlines.
seat_width=0.51308;%decide confort for passengers. Typical economy class
seat_pitch=0.76;
aisle_width=0.4826; %ICAO minimum 0.381m for +20 passengers

    %preferences
user_preferences=[2, 5, 0, 8, 10]; %stablish, from 0 to 10, being 0
absolutely irrelevant and 10 completely trascendental, the importance of
speed, cost, autonomy, range and CASM
seats_preference=300; %stablish how many seats would be preferable. 0 if
irrelevant

    %similar aircraft
similar_aircraft=[292.5343, 5.7922e+06, 1.1180e+04, 2.8322e+06, 0.0597];
%write the order of magnitude of a similar aircraft of the following: speed
(m/s) cost airframe ($), autonomy (s), range(m), CASM ($)


%CODE PARAMETERS

genes=5; %length of the vector of each element. The parameters in the genes
are: S_wing, S_HTP, D_h_fus, length_fus, span, in m and m^2
divisor=50; %the bigger the divisor, the smaller difference between fitness
N=500; %population size
SD_mix=0.05; %percentage of the value that will be standard deviation during
mix
```

```
SD_mutation=0.5; %percentage of the value that will be standard deviation
during mix. Starting point, will self-adapt
chance_mutation=0.02; %chance of mutation occuring
mutation_rate_change=0.01; %for adaptative mutation SD
mating_pool_size=1000; %how many times bigger than the population will be the
mating pool
MTOW_approx = 100000; %approximate MTOW for iterative calculation, kg
v_approx = 200; %approximate speed for iterative calculation, m/s

%INITIALIZE

best_element=zeros(1,genes); %initialize
best_fitness=0; %initialize
population=zeros(N,genes); %initialize
flag_stop=0; %initialize
success=zeros(1, 50); %initialize

%OPTIMIZATION ALGORITHM

    %creation of original population
    for i=1:N %for every member of the population
        population(i,:)=rand(1,genes); %we create random elements
        population(i,:)=(population(i,:)+parameter_min./...
        (parameter_max-parameter_min)).*(parameter_max-parameter_min); %we
size the genes according to max and min
    end
    generation=1 %this random population is the first generation
    while flag_stop==0 %while we have not decided to stop
        %fitness calculation
        fitness=zeros(N,1); %initialize
        for i=1:N %for every member of the population
           fitness(i)=fitnessfunctionmaster_v1(population(i,:), ...
               MTOW_approx, v_approx, flag_cargo, taper_ratio, ce, ...
               W_engine, user_preferences, seats_preference, seat_width,...
               seat_pitch, aisle_width, similar_aircraft); %we calculate the
fitness with a function
        end
        [max_fitness, position_max]=max(fitness); %we save the maximum value
and its location in the matrix
        if max_fitness>best_fitness %if the maximun fitness of this
generation has improved the all-time maximum, we save it
            best_fitness=max_fitness;%this is the new all-time maximum
            best_element=population(position_max,:); %this element must be
saved for the next generation
        end
        fitness_improvement(generation)=best_fitness; %in this vector we
store the best fitness of every generation
        element_improvement(generation, :)=best_element; %in this vector we
store the best element of every generation
        %creation of mating pool
        if min(fitness)<=0
            mating_pool_number=fitness+(abs(min(fitness)))+1;
        else
            mating_pool_number=fitness./min(fitness);
        end
        mating_pool_number=mating_pool_number.^(1+generation/divisor);
        mating_pool_number=round(mating_pool_number.*100);
        if sum(isinf(mating_pool_number))>=1
            divisor=generation;
            for i=1:length(mating_pool_number)
                if isinf(mating_pool_number(i))==1
```

```
                    mating_pool_number(i)=1;
                else
                    mating_pool_number(i)=0;
                end
            end
        end
        while sum(mating_pool_number)>mating_pool_size*N
            mating_pool_number=round(mating_pool_number./3);
        end
        mating_pool=zeros(sum(mating_pool_number),genes);
        k=1;
        for i=1:N
            for j=1:mating_pool_number(i,1)
                mating_pool(k,:)=population(i,:);
                k=k+1;
            end
        end
        %creation of new population
            %mix
        new_population=zeros(N,genes);
        for i=1:N
            parent_1=mating_pool(ceil(rand(1)*length(mating_pool)),:);
%random selection from the mating pool
            parent_2=mating_pool(ceil(rand(1)*length(mating_pool)),:);
            new_population(i,:)=...
                mixfunctionmaster_v1(parent_1, parent_2, SD_mix);
        end
        population=new_population;
            %success measure
        if generation>50
            for i=1:49
                success(i)=success(i+1);
            end
            if fitness_improvement(end)-fitness_improvement(end-1)>0
                success(50)=1;
            else
                success(50)=0;
            end
            %change mutation
            if sum(success)==0 && SD_mutation>0.001 ;
                SD_mutation=SD_mutation*(1-mutation_rate_change);%reduce
            end
            if sum(success)>1;
               SD_mutation=SD_mutation*(1+mutation_rate_change);%increment
            end
       end
        %mutation
        for i=1:N
            population(i,:)=mutationfunctionmaster_v1...
                (population(i,:),chance_mutation, SD_mutation, genes);
        end
        %check everything is within limits
        for i=1:N
            for j=1:genes
                if population(i,j)>parameter_max(j)
                    population(i,j)=parameter_max(j);
                end
                if population(i,j)<parameter_min(j)
                    population(i,j)=parameter_min(j);
                end
            end
        end
```

```
        population(end,:)=best_element; %the best element must be saved for
the new generation
        generation=generation+1

        %check stop criteria
        time=toc;
        flag_stop=...
stopcriteriamaster_v1(generation, fitness_improvement, time); %we check if it
needs to be stopped
    end

%FINAL SOLUTION
generation
time
best_fitness
best_element
```

```
function fitness=fitnessfunctionmaster_v1(population, MTOW_approx, ...
   v_approx, flag_cargo, taper_ratio, ce, W_engine, user_preferences, ...
   seats_preference, seat_width, seat_pitch, aisle_width, similar_aircraft)
%prueba con el b737-100. fitnessfunctionmaster_v1([102,28.99, 3.76,
27.66,28.35], 50000, 200, 1, 0.3, 2*0.744/36000*9.81, 2*2150, [2, 5, 0, 8,
10], 200, 0.51, 0.76, 0.48,  [292.5343, 5.7922e+06, 1.1180e+04, 2.8322e+06,
0.0597])

%weight calculation
M1=49; M2=27; M3=27; M4=24; M5=0.043; M6=1.3; M7=0.17; %values for transport
and bomber. Fighter, general aviation,... have different values
S_wing=population(1);
S_HTP=population(2);
D_h_fus=population(3);
length_fus=population(4);
span=population(5);
MTOW = MTOW_approx;
S_VTP = 0.7*S_HTP;  %We approximate the VTP surface related to the HTP
surface. The value 0.7 comes from typical aircrafts.
D_v_fus = D_h_fus; %we assume round
cost_fuel=0.6881; %cost of kilogram of fuel. Source: IATA 20-july-2018
https://www.iata.org/publications/economics/fuel-monitor/Pages/index.aspx
root_chord=S_wing/(span*0.5*(1+taper_ratio));

%estimation of crew, seats and fuel
number_seats_row=floor(D_h_fus/seat_width);
while D_h_fus-number_seats_row*seat_width<aisle_width
    number_seats_row=number_seats_row-1;
end
if number_seats_row>6
    while D_h_fus-number_seats_row*seat_width<2*aisle_width
    number_seats_row=number_seats_row-1;
    end
end
if number_seats_row>12
    while D_h_fus-number_seats_row*seat_width<3*aisle_width
    number_seats_row=number_seats_row-1;
    end
```

```
end
number_rows=floor((length_fus-5)/seat_pitch); %we consider 5m for unused
fuselage length
number_seats=number_rows*number_seats_row;
number_crew=3+ceil(number_seats/50); %This depends on many factors. Since its
not critical, we will simplify as
pilot+copilot+auxiliar+auxiliarforeach50passenger

W_crew= 100*number_crew;
W_payload=100*number_seats; %we assume 100kg for passenger/crew+luggage
W_fuel=309.16*S_wing-16248; %in kilograms, from statistical values
if W_fuel<0
    W_fuel=0;
end

S_exp=S_wing - D_h_fus*root_chord;
S_fus_wet = length_fus *pi*D_h_fus+2*pi*(0.5*D_h_fus)^2; %We are assuming
round fuselage

%we calculate MTOW iteratively
MTOW_ini=0;
while abs(MTOW-MTOW_ini)>1
    MTOW=MTOW_ini;
    W_empty=S_exp*M1 + S_HTP*M2 + S_VTP*M3 + S_fus_wet*M4 +...
        MTOW_ini*M5 + W_engine*M6 + MTOW_ini*M7; %Kg
    MTOW_ini=W_crew+W_payload+W_empty+W_fuel; %Kg
end

%Speed calculation
S_wet=S_fus_wet+2*(S_VTP+S_HTP+S_exp)*1.1;
nu=3.525*10^-5; rho=0.412707; %at 10 km
oswald=0.70; %Estimated data. Source: http://www.fzt.haw-
hamburg.de/pers/Scholz/OPerA/OPerA_PUB_DLRK_12-09-10.pdf

v_range=v_approx; v_range_ini=0; %maximum range speed
while abs(v_range_ini-v_range)>1;
    v_range_ini=v_range;
    Re = S_wet*v_range_ini /(span*nu);
    Cd0 = (S_wet/S_wing)*(0.00258+0.00102*exp(-6.28*10^-9*Re)...
        +0.00295*exp(-2.01*10-8*Re)  );
    k=1/(pi*oswald*(span^2/S_wing));
    Cl=sqrt(Cd0/(3*k));
    v_range=sqrt(2*MTOW*9.81/(rho*S_wing*Cl));
end
v_auto=v_approx; v_auto_ini=0; %maximum autonomy speed
while abs(v_auto_ini-v_auto)>1;
    v_auto_ini=v_auto;
    Re = S_wet*v_auto_ini /(span*nu);
    Cd0 = (S_wet/S_wing)*(0.00258+0.00102*exp(-6.28*10^-9*Re) ...
        +0.00295*exp(-2.01*10-8*Re)  );
    k=1/(pi*oswald*(span^2/S_wing));
    Cl=sqrt(Cd0/(k));
    v_auto=sqrt(2*MTOW*9.81/(rho*S_wing*Cl));
end
v=v_range; %we consider cruise speed as maximum range speed.

%Cost calculation
N_test=3; %Approximation of number of test aircraft
W_empty_pounds= W_empty*2.20462;

engineering_hours=0.023*W_empty_pounds^0.66*v^0.96;
```

```
tooling_hours=0.47*W_empty_pounds^0.64*v^0.5;
manufacturing_hours=0.35*W_empty_pounds^0.79*v^0.42;
if flag_cargo==1
    quality_hours=0.12*manufacturing_hours;
else
    quality_hours=0.085*manufacturing_hours;
end

engineering_dollars=engineering_hours*20.06*5.8; %5.8 is the value in 2018 of
one 1973 dollar
tooling_dollars=tooling_hours*18.63*5.8;
manufacturing_dollars=manufacturing_hours*16.83*5.8;
quality_dollars=quality_hours*16.83*5.8;
materials_non_recurring_dollars=0.000024*W_empty_pounds^0.72*v^1.92*5.8;
materials_recurring_dollars=0.05*W_empty_pounds^0.88*v^0.87*5.8;
flight_test_dollars=0.13*W_empty_pounds^0.71*v^0.59...
    *N_test^0.72*flag_cargo^-1.56*5.8;

cost=engineering_dollars+tooling_dollars+manufacturing_dollars+...
    materials_non_recurring_dollars+materials_recurring_dollars+...
    flight_test_dollars+quality_dollars;

%autonomy and range
Cl=sqrt(Cd0/(k));
Cd=Cd0+Cl^2*k;
Emax=Cl/Cd;
chi=W_fuel/MTOW;

    %autonomy calculation
autonomy=Emax/ce*log(1/(1-chi));

    %range calculation
range=autonomy*v_range*sqrt(3)/2;

%CASM calculation

CASM=W_fuel*cost_fuel/(range/1852)/number_seats; %Only considering fuel
costs, not salaries or other
if range==0 %in case it does not take off
    CASM=99999;
end
%Final mix
adimentional_qualities=[v, cost, autonomy, range, CASM]./similar_aircraft;
adimentional_qualities(2)=1/adimentional_qualities(2); ...
    adimentional_qualities(5)=1/adimentional_qualities(5); %the lower the
cost and CASM, the better
adimentional_qualities_pond=adimentional_qualities.*user_preferences;
fitness=sum(adimentional_qualities_pond);
if seats_preference~=0
   fitness=fitness*(1-abs(seats_preference-number_seats)/seats_preference);
end

%Final checks
    %if it cant fly at 10km, it is not accepted
    L=0.5*rho*v^2*S_exp*Cl; %notice we use the exposed wing area, to avoid
extremely short wings
    if L<MTOW*9.81
        fitness=0;
    end
    %wing root chord cannot be longer than fuselage
    if root_chord>length_fus
```

```
        fitness=0;
    end
    %check fuselage finesse within limits
    if length_fus/D_h_fus<4 || length_fus/D_h_fus>20
        fitness=0;
    end
    %check we are not having too many seats per row
    if number_seats_row>18
        fitness=0;
    end

end
```

```
function son=mixfunctionmaster_v1(parent_1, parent_2, SD_mix)
    son=normrnd(0.5*(parent_1+parent_2), SD_mix*0.5*(parent_1+parent_2));
end
```

```
function mutated_element=mutationfunctionmaster_v1...
    (element, mutation_chance, mutation_SD, genes)
mutated_element=element;
    for i=1:genes
        if rand(1)<mutation_chance
            mutated_element(i)=normrnd(element(i),mutation_SD*element(i));
        end
    end
end
```

```
function flag_stop=stopcriteriamaster_v1...
    (generation, fitness_improvement, time)
    if generation>300 && time>180
        if fitness_improvement(end)-fitness_improvement(end-300)==0
            flag_stop=1;
        else
            flag_stop=0;
        end
    else
        flag_stop=0;
    end
end
```
