Universidad Zaragoza
1542

Trabajo Fin de Grado

Optimización Bayesiana aplicada a la simulación de fluidos

Autor

Javier García Barcos

Director

Rubén Martínez Cantín

Ponente

Luis Montesano del Campo

Grado en Ingeniería Informática
Escuela de Ingeniería y Arquitectura
2015

Optimización Bayesiana aplicada a la simulación de fluidos

RESUMEN

El uso de simuladores nos permite explorar diseños alternativos sin necesidad de producir caros prototipos, pero diseñar optimizaciones alrededor de estos sistemas conlleva un elevado tiempo de ejecución debido a la larga duración de cada simulación. Además, la función subyacente es desconocida (las denominadas funciones caja-negra o *black-box*), por lo que es imposible conocer la derivada de la función a optimizar, necesaria en la mayoría de métodos de optimización.

El presente trabajo trata de resolver dichas limitaciones mediante la aplicación de aprendizaje automático en métodos de optimización global. Concretamente se abordará la optimización Bayesiana con el objetivo de minimizar el número de muestras necesarias para encontrar el óptimo, reduciendo así el tiempo total necesario de optimización.

La optimización Bayesiana se ha realizado a partir de la librería BayesOpt y, como objetivo de la simulación, se ha utilizado el software simulaciones de fluidos XFlow, producto de NextLimit Technologies SL. Por tanto, ha sido necesario desarrollar una interfaz entre ambos programas, mediante la cual se han diseñado experimentos a optimizar y cuyos resultados han sido utilizados para argumentar la viabilidad de optimización Bayesiana en este tipo de problemas.

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD

(Este documento debe acompañar al Trabajo Fin de Grado (TFG)/Trabajo Fin de Máster (TFM) cuando sea depositado para su evaluación).

TRABAJOS DE FIN DE GRADO / FIN DE MÁSTER

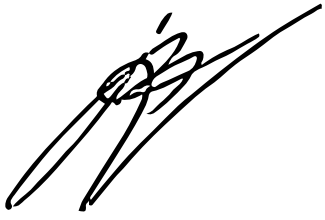
D./Dª. Javier García Barcos,

con nº de DNI 72808890K en aplicación de lo dispuesto en el art. 14 (Derechos de autor) del Acuerdo de 11 de septiembre de 2014, del Consejo de Gobierno, por el que se aprueba el Reglamento de los TFG y TFM de la Universidad de Zaragoza,

Declaro que el presente Trabajo de Fin de (Grado/Máster) Grado, (Título del Trabajo) Optimización Bayesiana aplicada a la simulación de fluidos,

es de mi autoría y es original, no habiéndose utilizado fuente sin ser citada debidamente.

Zaragoza, 25 de septiembre de 2015



Fdo: Javier García Barcos

Agradecimientos:

Quisiera dedicar está sección a agradecer a todos aquellos que me han apoyado durante el camino y han hecho posible que pueda llegar hasta aquí:

A mi director de proyecto, **Rubén Martínez Cantín**. Te agradezco todo lo que me has enseñado y, sobre todo, el empeño que has puesto en ayudarme y guiarme.

A mis compañeros **Jorge y Gabriel**, por haberme aguantado durante toda la Ingeniería. Gracias a vuestra ayuda hasta los retos más difíciles han sido amenos.

A mis amigos más cercanos. Sobre todo a **Francho**, con quien siempre he podido contar tanto en los buenos ratos como en los malos.

A **mis padres**, por el apoyo constante recibido durante todos estos años.

A **mi hermano Luis**, un pilar fundamental en mi día a día. Nunca te podría agradecer lo suficiente todo lo que haces por mí.

Tabla de contenidos

1. Introducción

En el diseño y desarrollo de productos de ingeniería, es habitual la aplicación de simuladores para la validación y experimentación dentro de procesos iterativos, lo que permite ir refinando el producto en cada iteración del proceso. Para reducir la interacción necesaria con el ingeniero, se aplican técnicas de optimización alrededor de estos sistemas, de tal forma que operen de manera semi-supervisada por el ingeniero.

El principal problema es que las exigencias en tiempo de cada simulación son elevadas. En base a las pruebas que se han realizado, una simulación de apenas 2 segundos de un túnel de viento con 3 aerogeneradores pueda tardar hasta 26 horas en una *workstation* de última generación. Por tanto, es fundamental que la optimización utilice el menor número necesario de simulaciones para dar con la configuración óptima de los parámetros. Además, la función subyacente a optimizar es desconocida (las denominadas funciones caja-negra o *black*-box), lo que nos impide conocer su derivada, necesaria en la mayoría de métodos de optimización, y nos limita las suposiciones que podemos realizar sobre la función, como la convexidad de la función, lo que nos obliga a suponer la existencia de mínimos locales y, por tanto, requerimos de un método de optimización global.

Estas dificultades nos restringen al uso de métodos de optimización global, que puedan lidiar con funciones *black-box* y, además, requieran el menor número posible de muestras para reducir el tiempo total de ejecución. Uno de los métodos que cumplen estas limitaciones es la optimización Bayesiana.

La optimización Bayesiana mezcla técnicas de optimización matemática con aprendizaje automático. A grandes rasgos, trata de aprender la forma de la función a partir de las muestras previas y usar esta información para guiar la búsqueda del óptimo de manera más eficiente. Esto permite reducir el número total de muestras y trabajar con funciones tipo *black-box.* Por tanto, es factible la aplicación de optimización Bayesiana a optimización de simulaciones, lo que ha motivado a llevarlo a la práctica en este trabajo.

Debido a que la optimización se ha llevado a cabo en un software de simulación de fluidos, este trabajo ha tenido un elevado componente multidisciplinar, debido a que mezcla temas como aprendizaje automático y optimización matemática con campos como la mecánica de fluidos y la aeronáutica.

Este Trabajo de Fin de Grado (TFG) se ha realizado en el Centro Universitario de la Defensa como parte de una colaboración entre el Learning Laboratory del grupo de Robótica, Percepción y Tiempo Real (RoPeRT) del I3A y la empresa NextLimit Technologies SL.

1.1. Objetivo y alcance

El objetivo de este trabajo es el ajuste automático de parámetros de diseño y configuración para simulaciones de fluidos. Para ello se ha revisado el estado del arte y se ha decidido utilizar métodos de optimización Bayesiana por su eficiencia a la hoar de reducir el número de simulaciones requeridas para optimizar dichos parámetros. Para la simulación de fluidos ha sido necesario conocer y estudiar las herramientas y metodologías disponibles y, en

concreto, el software XFlow, proporcionado por la empresa NextLimit. Para ello ha sido necesario:

- El estudio de artículos y teoría de optimización Bayesiana que permitan identificar la metodología de aplicación a este tipo de problemas.
- La implementación de un conjunto de mejoras y nuevas funcionalidades sobre la librería de optimización bayesiana BayesOpt, lo que ha requerido un estudio detallado tanto del problema de optimización Bayesiana como del código fuente del núcleo C++ de BayesOpt.
- El aprendizaje del programa XFlow con el fin de diseñar nuevas simulaciones con las que experimentar la metodología propuesta.
- Diseño de simulaciones en XFlow tanto para evaluar las herramientas desarrolladas e integradas como para evaluar el potencial en aplicaciones reales como el diseño de un ala.
- Diseño e implementación de una interfaz Python que permita la interacción no supervisada entre BayesOpt y XFlow de manera que permita la optimización mediante BayesOpt de simulaciones de XFlow.
- La experimentación mediante el sistema desarrollado y el uso de simulaciones que permita verificar el correcto funcionamiento de la interfaz y ayude a evaluar la aplicabilidad de la metodología propuesta a problemas reales.
- El desarrollo de software en Python para el procesamiento y analisis de los resultados finales.

1.2. Metodología y Herramientas

Como software de optimización, se ha desarrollado sobre la librería de optimización Bayesiana: BayesOpt. Está implementada en C++ y dispone de interfaz en C, Python, Matlab y Octave. Ha sido necesario incluir nuevas características en BayesOpt, por lo que se ha sido necesario implementar en C++. Para la compilación del proyecto se ha utilizado CMake.

Como software de simulación se ha utilizado un CFD (Computational Fluid Dynamics) o un software de simulación de fluidos. Contamos con la colaboración de NextLimit Technologies, quien nos ha proporcionado una licencia de XFlow CFD. El programa dispone de un modo GUI (*Graphical User Interface*) donde se diseñaran y se especificarán las condiciones a simular. También ofrece un modo CLI (*Command Line Interface*) que será usado por la interfaz para ejecutar las distintas simulaciones requeridas en la optimización.

Para integración entre el software de optimización y el software de simulación, se ha utilizado Python para desarrollar una interfaz que comunique ambos componentes y permita la creación de nuevas simulaciones a optimizar. Los principales motivos de utilizar Python han sido: reaprovechar la interfaz Python ya existente en BayesOpt y reutilizar el script en Python que nos ha proporcionado el equipo de XFlow. Dicho script permite recuperar los resultados de la simulación a partir de los ficheros binarios que genera la simulación XFlow.

También se ha utilizado Python para la generación de gráficas que nos ayuden a argumentar los resultados obtenidos de las optimizaciones mediante la librería matplotlib (3).

Ha sido necesario el uso de software 3D para la visualización de los modelos geométricos generados en una de las optimizaciones, mediante Blender (4) o FreeCAD (5).

Para el control de versiones del código fuente desarrollado se ha utilizado Mercurial y BitBucket, todo el código desarrollado (BayesOpt, interfaz y simulaciones diseñadas) se encuentra en el siguiente repositorio:

https://bitbucket.org/rmcantin/bayesoptnl

El desarrollo sobre la librería BayesOpt se ha realizado de manera incremental y mediante evaluación continua. Se ha procurado introducir el uso de test unitarios y evaluación continua sin ayuda de un framework de test unitarios, con el objetivo de evitar introducir nuevas dependencias al proyecto.

Las funcionalidades de la interfaz se han ido implementando a medida que se iban diseñando las simulaciones, de tal forma que al identificarse una necesidad, se procedía a implementarla en la interfaz.

Por último, para facilitar el desarrollo, BayesOpt, XFlow y la interfaz han sido desplegados en el mismo equipo, lo que facilita la interacción entre los componentes y permite centrarnos en la obtención de resultados.

1.3. Contenidos de la Memoria

Se comenzará por introducir la optimización Bayesiana en el capítulo 2. Aquí se cubrirá a grandes rasgos en qué consiste la optimización Bayesiana desde un punto de vista teórico y se indicarán las técnicas concretas utilizadas. A continuación, en el capítulo 3, se enumerarán el conjunto de modificaciones y mejoras introducidas en la librería BayesOpt. Esto ha requerido conocer tanto la optimización Bayesiana como la propia librería BayesOpt. Al ser un trabajo multidisciplinar, ha sido necesario aprender a utilizar XFlow para el diseño de simulaciones. Por tanto, ha sido indispensable comprender la terminología utilizada en campos como la mecánica de fluidos o la aeronáutica para poder analizar los resultados obtenidos. En el capítulo 4 se detallarán algunos de los términos utilizados y las particularidades de XFlow.

Una vez explicados ambos componentes, XFlow y BayesOpt, se procederá en el capítulo 5 a explicar la integración entre ambos cuyo resultado es una interfaz que permite modificar la interacción entre ambos para la generación de nuevas optimizaciones.

En el capítulo 6 se cubrirá la experimentación realizada y los resultados obtenidos. En los apartados de este capítulo se detallarán los dos experimentos desarrollados y sus resultados: optimización del ángulo de un cilindro minimizando la fuerza de arrastre y optimización de la forma de un ala para obtener la mínima fuerza de arrastre restringido a un valor mínimo de fuerza de elevación. Las conclusiones derivadas de la experimentación aparecerán en el capítulo 7. Aquí también se discuten posibles trabajos derivados del presente trabajo, como mejoras en la optimización o la inclusión del sistema a un entorno profesional.

Por último, en el Anexo I se presenta el manual de usuario. Ahí se realiza una guía con el fin de explicar la interfaz a un usuario final. También se incluyo el listado de funciones disponibles para facilitar el uso de la interfaz.

2. Optimización Bayesiana

En este apartado se expondrá un resumen sobre optimización Bayesiana y se concretarán algunas de las técnicas utilizadas. El contenido de este resumen está basado en un artículo tutorial sobre optimización Bayesiana (6). En caso de necesitar profundizar más, se recomienda consultar (7), un informe más reciente sobre optimización Bayesiana en el que se profundiza más en sus partes y en el que se detallan técnicas más recientes.

Partimos de una función desconocida, que denominaremos función coste *f,* la cual queremos minimizar o maximizar. Al ser *f* desconocida, tenemos que asumir que puede no ser convexa o que pueden existir varios mínimos locales, por lo que se necesitará una optimización de tipo global, siendo necesario delimitar el espacio de la función coste a optimizar (*bounds)*. La única forma de obtener información de *f* es mediante la evaluación de puntos *x* en la función de coste donde, si existe ausencia de ruido, se devolverá el valor real de la función coste en dicho punto: *f(x)*. Por tanto para recuperar la función coste *f*, sería necesario un muestreo exhaustivamente la función, algo intratable si el coste de cada evaluación es elevado. Aquí es donde introducimos la optimización Bayesiana con el objetivo de minimizar el número de muestras necesarios al guiar la búsqueda de manera eficiente hacia el óptimo.

La optimización Bayesiana toma su nombre debido a que usa el *Teorema de Bayes* (8), el cual expone que la probabilidad *a posteriori* de un modelo M dada cierta evidencia E es proporcional a la probabilidad de E dado M multiplicado por la probabilidad *a priori* de M:

$$P(M|E) \propto P(E|M)P(M)$$

Con respecto a la ecuación original del *Teorema de Bayes*, en lugar de dividir por P(E), se simplifica y se asume que P(E) actúa como constante normalizada. Por tanto, retirando la constante obtenemos la ecuación proporcional (indicada mediante el uso del símbolo $\propto$).

Otro motivo por el que se denomina Bayesiana es porque la interpretación probabilista es la Bayesiana o subjetiva. En contraste con otras interpretaciones como la frecuentista, la probabilidad Bayesiana es la cantidad que se asigna con el propósito de representar un estado de conocimiento. La probabilidad es asignada a una hipótesis, la cual se va actualizando conforme se adquiere nueva información.

En optimización Bayesiana, la probabilidad *a priori* del modelo P(M) representa las suposiciones sobre las posibles funciones coste válidas. Aunque la función coste es desconocida, es razonable asumir que existe algún tipo de conocimiento *a priori* acerca de la función, como la suavidad de la función o el ruido de las muestras. Esto permite determinar, de todas las funciones posibles, cuales son más factibles.

Por otra parte, el modelo *M* será la función coste f que queremos optimizar y la evidencia E es el conjunto de observaciones o muestras obtenidas hasta el momento sobre la función coste. Por tanto, si se define x_i como la *i-ésima* muestra y $f(x_i)$ como la evaluación en dicho punto, conforme acumulamos observaciones $D_{1:t} = \{x_{1:t}, f(x_{1:t})\}$, la distribución *a priori P(f)* se combina con la función de verosimilitud $P(D_{1:t} \mid f)$. Dicho de otro modo, si la función es suave y libre de ruido, nuevos puntos con mucha varianza u oscilación son menos probables

que puntos que se desvíen poco de la media. Combinando esto obtenemos la distribución *a posteriori*:

$$P(f|D_{1:t}) \propto P(D_{1:t}|f)P(f)$$

La distribución *a posteriori* captura la información actualizada sobre la función coste desconocida, lo que nos ayuda a discernir que funciones son más probables que correspondan a la función coste dadas las observaciones previas. Otra interpretación es la estimación de la función coste mediante un modelo que la sustituya (*surrogate model).*

En la figura 2.a. aparecen 2 distribuciones de funciones, una en la que únicamente contamos con la información a priori (a) y otra en la que contamos con información extra proporcionada por 2 muestras.

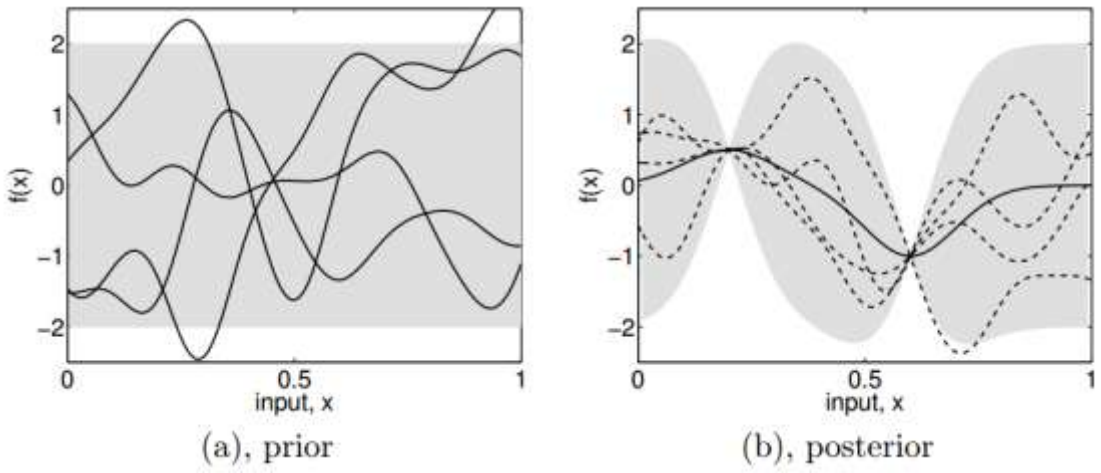


Figura 2.a. Distribución a priori (a) y a posteriori (b) mediante procesos Gaussianos. En procesos Gaussianos cada punto del espacio se le asocia una variable aleatoria con distribución normal. La unión de todas las variables aleatorias dan lugar a una distribución sobre funciones como las que aparecen en la figura. En (a) solo se dispone de la información a priori, media 0 y cierta desviación típica (área sombreada), por lo que las funciones muestreadas de la distribución son muy variadas (líneas continuas). En (b) disponemos de 2 muestras, se dispone de más información de la media (línea continua) y las posibles funciones muestreadas son más limitadas (líneas discontinuas). La fuente original pertenece a la introducción del libro *Gaussian Processes for Machine Learning* (9).

En este trabajo se ha utilizado procesos Gaussianos como *surrogate model*. La idea de los procesos Gaussianos es modelar cada punto del espacio como una variable aleatoria con distribución normal. Al unir todas esas variables aleatorias del espacio da lugar a una distribución sobre funciones, es decir, si tomamos una muestra de la distribución estamos obteniendo una función. Un proceso Gaussiano se especificada mediante una función de la media $m(x)$ y un *kernel* $k(x, x')$:

$$GP(m(x), k(x, x'))$$

Donde habitualmente se selecciona una función cero para la media $m(x)$:

$$GP(0, k(x, x'))$$

No obstante, en este trabajo se ha utilizado un proceso Gaussiano más elaborado añadiendo distribuciones a priori alternativas. Se ha introducido un modelo de regresión lineal $\Phi(x)$ en la media (en lugar de la media cero) y se han incluido la distribución a priorí w sobre la media y la distribución a priorí σ^2 sobre la varianza, junto al *kernel* k.

$$GP(w^T\Phi(x)\,, \sigma^2 k(x, x'))$$

Una función de covarianza o *kernel*, es una función *k* que acepta un par de valores y devuelve un valor numérico real. Dicho valor es la correlación entre el par de valores introducidos. Concretamente, en este trabajo se utilizará el *kernel* Matern ARD de 5º orden:

$$k_{v=5/2}(r) = \left(1 + \frac{\sqrt{5}r}{l} + \frac{5r^2}{3l^2}\right)\exp\left(-\frac{\sqrt{5}r}{l}\right), \quad donde: r = ||\, x_1 - x_2\, ||_2$$

En el *kernel* Matern ARD, l es un híper-parámetro que permite parametrizar las propiedades de suavidad del modelo, ver en figura 2.b. Habitualmente se utiliza el símbolo Θ para señalar al híper-parámetro del *kernel.*

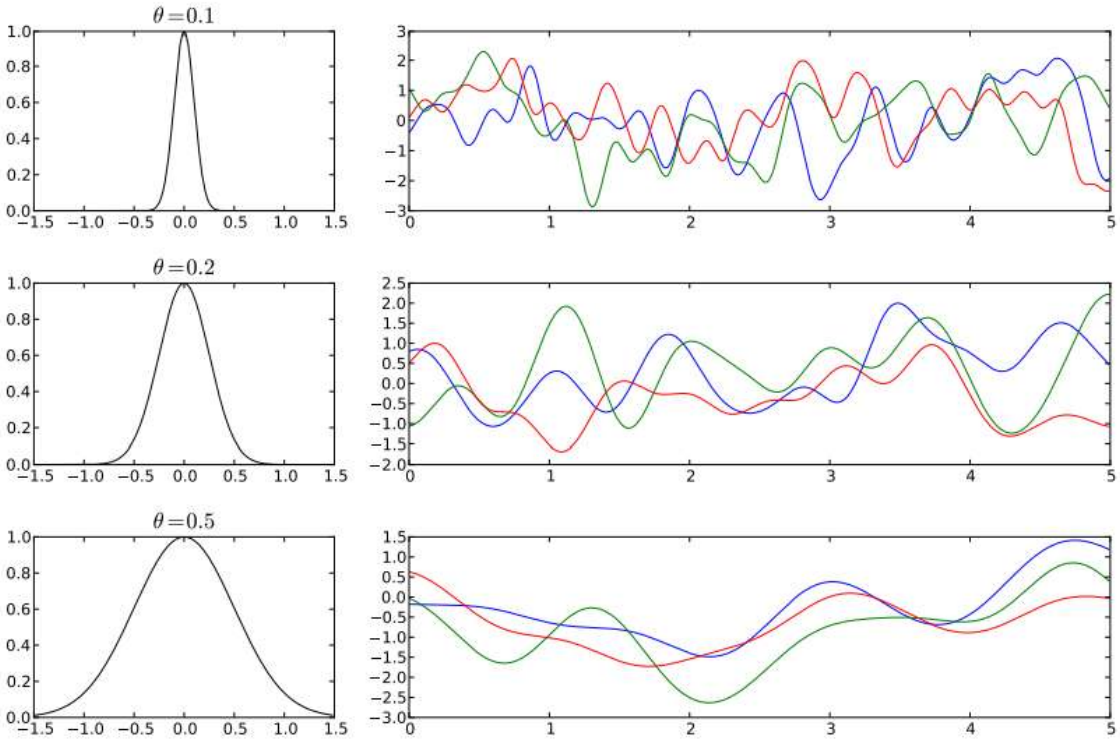


Figura 2.b. El *kernel* define la correlación entre pares de puntos. La figura muestra la influencia de cambiar el híper-parámetro Θ del kernel (izquierda) en la correlación de los puntos. A mayor valor mayor correlación y, por tanto, las posibles funciones muestreadas de una distribución (derecha). La fuente original de la imagen pertenece al tutorial de optimización Bayesiana (6)**.**

Los híper-parámetros tienen que ser aprendidos en base a la evidencia utilizando técnicas como máxima verosimilitud (ML) (10) o Markov Chain Monte Carlo (MCMC) (11). Como en optimización Bayesiana con cada iteración se obtiene una nueva evidencia, es necesario reaprender los híper-parámetros para ajustarse lo mejor posible a la información actualizada. En comparativa entre ambas técnicas: MCMC es más lento porque requiere

repetir todos los cálculos para cada una de las muestras que ejecuta pero es más robusto porque trabaja con varios de los posibles valores de híper-parámetros. ML utiliza solo el más probable para la estimación de los híper-parámetros.

Para poder discutir sobre la siguiente muestra a evaluar, a partir de la distribución a posteriori de la iteración actual (t), necesitamos obtener la distribución predictiva que nos permita discutir sobre el valor de cualquier x en t+1. Utilizando procesos Gaussianos con función media cero $GP\big(0, k(x, x')\big)$ y aplicando la formula de Sherman-Morrison-Woodbury podemos obtener la siguiente distribución predictiva (más detallado en (6)):

$$P(f_{t+1}|D_{1:t}, x_{t+1}) = N(\mu_t(x_{t+1}), \sigma_t^2(x_{t+1}))$$

donde:

$$\mu_t(x_{t+1}) = k^T K^{-1} f_{1:t}$$

$$\sigma_t^2(x_{t+1}) = k(x_{t+1}, x_{t+1}) - \ k^T K^{-1} k$$

A partir de $\mu_t(\cdot)$ y $\sigma_t^2(\cdot)$ podemos obtener la media y la varianza de la distribución de predicción en cada punto del espacio. Aquí es donde entran en juego las funciones de adquisición C, las cuales generan una interpretación de $\mu_t(\cdot)$ y $\sigma_t^2(\cdot)$ de tal manera que la muestra x que maximiza la función de adquisición C corresponde a la mejor muestra a evaluar para la siguiente iteración x_{t+1}:

$$x_{t+1} = \arg_{\mathrm{x}} \max C_t(x)$$

Este proceso de optimización secundario de maximizar la función de adquisición es mucho más sencillo de optimizar debido a que la función está diseñada de tal forma que sea rápida de evaluar y, por tanto, de maximizar.

La decisión obtenida mediante el uso de la función de adquisición debe dar lugar a un equilibrio entre la explotación y la exploración para poder guiar la optimización de manera eficiente hacia el óptimo:

● La explotación es la acción de tomar puntos en una zona limitada del espacio cercana al mejor valor encontrado hasta el momento.

● La exploración es la acción de tomar puntos en una zona extensa del espacio en la que no se tiene información, con el objetivo de obtener puntos más prometedores que el mejor valor encontrado hasta el momento.

La función adquisición utilizada en este trabajo es el *expected improvement (EI)*, la cual mide la magnitud de posible mejora con respecto al mejor obtenido hasta el momento o cero en el caso que no exista mejora. Siendo *E[...]* el valor esperado y siendo un problema de maximización, la formula quedaría:

$$\mathrm{EI(x)} \ = \ \mathrm{E[max(0, f(x) - \ f(x_{best})]}$$

En caso de ser de minimización, se cambia el signo:

$$EI(x) = E[\max(0, f(x_{best}) - f(x)]$$

Se ha elegido *expected improvement* como función de adquisición frente a otras alternativas debido a que no requiere ajustar un parámetro propio para lidiar con el problema de exploración y explotación y porque es conocida por dar lugar a una optimización eficiente en el número de muestras necesario al minimizar, aunque otras funciones de adquisición pueden ser igual de válidas en función de la situación (12).

Por último, es importante destacar que la optimización necesita datos con los que empezar para construir el primer modelo y así evitar sesgos. Por ello, las muestras iniciales se obtienen mediante un muestreo adecuado del espacio. En concreto, se ha utilizado el *Latin Hipercube Sampling* (13)*,*

Resumiendo, la optimización Bayesiana puede dividirse en 2 etapas:

● Etapa de aprendizaje: Aprender el modelo y los híper-parámetros a partir de la información disponible: muestras evaluadas e información a priori, lo que ayuda a generar un modelo predictivo.

●Etapa de decisión: Decidir la siguiente muestra a partir del modelo predictivo mediante la maximización de la función de adquisición.

Ambas etapas son ejecutadas en bucle durante un número determinado de iteraciones. Al finalizar el bucle, el mejor valor encontrado será devuelto como el óptimo. En la figura 2.c. puede verse un ejemplo de 3 iteraciones de una optimización Bayesiana de una función.

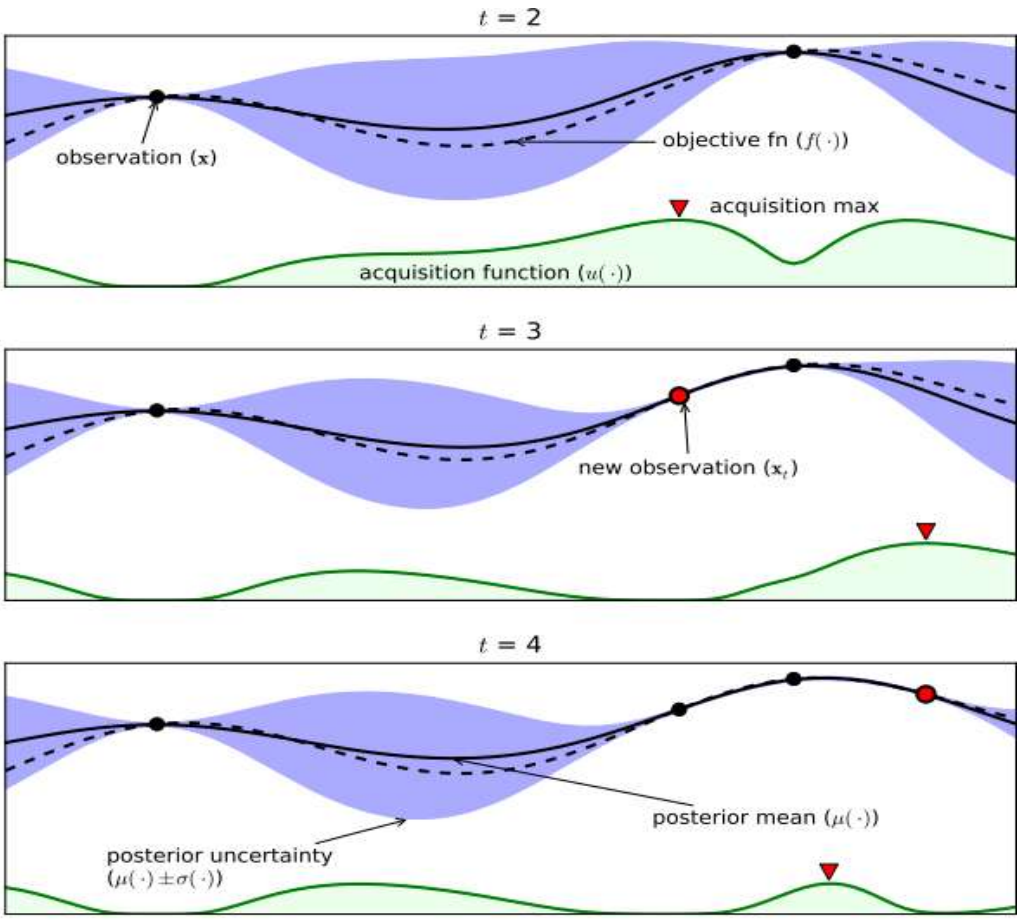


Figura 2.c. En esta figura se muestran 3 iteraciones de la Optimización Bayesiana. La fuente original pertenece al tutorial de optimización Bayesiana (6). En cada una de las iteraciones se muestra el modelo aprendido, con cierta media (línea continua) y cierta incertidumbre (área sombreada) con el fin de maximizar una función coste (línea discontinua) que es desconocida. A partir del modelo aprendido se hace uso de una función de adquisición (línea continua inferior) cuyo máximo (triangulo) nos índica donde lanzar la siguiente muestra. Se puede observar como en la primera imagen, debido a la alta incertidumbre en la parte central hay un máximo en la función de adquisición. En cuanto se obtiene la observación y la incertidumbre disminuye, esa zona pierde interés y pasa a ser cero en la función de adquisición. Nótese que la función es de maximización, habría que cambiar el signo en la función de coste para realizar la minimización y también cambiar el signo en función de adquisición utilizada.

3. Desarrollo sobre BayesOpt

Aunque uno de los principales motivos de usar Optimización Bayesiana es el de reducir el número necesario de muestras para encontrar el óptimo con el fin de reducir el tiempo total necesario, no hay que olvidar que las simulaciones utilizadas pueden durar desde segundos hasta horas o días, en función de su complejidad y la capacidad de computo. Si existiera cualquier percance, por ejemplo un corte de electricidad, perderíamos todo el progreso de la optimización acumulado hasta el momento.

Por ello, es necesario dotar de un mecanismo para poder guardar y restaurar el progreso en la optimización. A pesar de que el código fuente de BayesOpt contemplaba la posibilidad de esta funcionalidad en sus parámetros, no se encontraba implementada. Así que

uno de los primeros objetivos abordados fue la implementación de dicho sistema en la librería BayesOpt..

Para desarrollar el guardado y restauración del estado de la optimización, fue necesario identificar y guardar todos los datos necesarios en una clase que represente el estado de la optimización y serializar la información de dicho estado para poder escribirla y leerla de fichero.

En la identificación de los datos que alberga la clase estado hay que tener en consideración parámetros que puedan ser modificados en la ejecución que restaura el estado. Por ejemplo: el parámetro número total de iteraciones, en lugar de restaurarlo puede interesar poder modificarlo en la nueva ejecución, pudiendo extender el número de iteraciones sin necesidad de reiniciar la optimización.

Por otra parte, para evitar nuevas dependencias, se ha decidido desarrollar un serializador sencillo en lugar de buscar una librería que ofrezca un serializador. Esto nos permite personalizar el serializador y decidir el formato de salida de los datos. Se ha decidido un formato sencillo tipo "clave=valor" para que sea legible por humanos y sea sencillo de analizar sintácticamente. Esto es útil a la hora de analizar los resultados a partir del fichero, ya que puede realizarse en cualquier entorno de programación implementando un sencillo *parser*. El serializador también ha sido reutilizado para poder introducir los parámetros de BayesOpt mediante un fichero en los ejemplos de programas C++. Al ejecutar el programa, se indica en el argumento el nombre del fichero que contiene parámetros.

Otra modificación realizada a BayesOpt ha sido la de mejorar la clase que almacena los parámetros de la optimización y es usada como API por las interfaces de BayesOpt. Los parámetros se encontraban en una estructura de C mientras que el resto de BayesOpt está implementado mediante un esquema orientado a objetos en C++. El principal problema es que la modificación de los parámetros de C a C++ ha repercutido en todo el código de la librería, por lo que para modificarlo ha sido necesario un conocimiento exhaustivo de la librería BayesOpt. Además, la versión C sigue siendo necesaria para la API, por lo que ambas versiones deben coexistir y deben existir mecanismos para traducir entre ambas versiones. Las ventajas de la modificación son evidentes: homogenización del código fuente de la librería y acceso a las librerías estándar de C++, las cuales ayudan a simplificar el manejo de la memoria (añadiendo robustez frente a *memory leaks)* y permiten el uso de estructuras dinámicas (como *std::vector*) que facilitan el uso de ciertos parámetros en forma vectorial como los límites del espacio de entrada o las distribuciones a priori de los híper-parámetros. Por otro lado, como sigue existiendo la versión en C, no se pierde ninguna funcionalidad fundamental, como el API, que permite el uso de los *wrappers* a Python, Matlab y Octave.

Por último, para poder asegurar el correcto funcionamiento de las mejoras introducidas, se han acompañado con test unitarios conforme se iban desarrollando las mejoras. Esto ha ayudado a asegurar que todo sigue funcionando tras cada cambio introducido a BayesOpt.

4. XFlow

Con motivo de la colaboración con NextLimit Technologies S.L., contamos en este trabajo con el software XFlow CFD (2). XFlow es un software de simulación de fluidos de altas prestaciones. La principal característica es que no requiere la creación de mallas ya que realiza una aproximación cinética basada en partículas. Esto evita la creación de la tradicional malla no uniforme que limitaría la complejidad de las superficies de los objetos geométricos.

Concretamente utiliza una tecnología propietaria basada en métodos Lattice Boltzmann (LBM). El uso de LBM permite la ejecución eficiente en arquitecturas masivamente paralelizadas, como los clúster mediante el uso de MPI, pero también es capaz de aprovechar la paralelización en computadores *multi-core*. XFlow permite simulaciones de diversos tipos: aerodinámicas, hidrodinámicas, acústicas y térmicas, entre otras. En este trabajo se ha decidido centrarnos en simulaciones de un único fluido, denominadas *single phase*, y se ha utilizando una configuración tipo túnel de viento.

La forma en la que XFlow realiza la discretización libera al usuario de configurar parámetros concretos de los algoritmos, facilitando el aprendizaje del simulador. Únicamente es necesario configurar los parámetros que controlan la disposición y tamaños de las celdas de la discretización. Es habitual configurar la discretización estructurada en varios niveles, es decir, a parte de configurar el tamaño global de las celdas, permite adaptar el tamaño a resoluciones menores cerca de paredes o cuando detecta gradientes elevados (útil para modelar correctamente los vórtices).

Ofrece herramientas de pre-procesado y post-procesado, las cuales han sido utilizadas para la generación de algunas de las imágenes que aparecen en este trabajo.

Como referencia, las simulaciones pueden durar desde minutos hasta horas o incluso días, en función de la complejidad de la simulación y de la potencia de cálculo disponible. Por ello, aquí cobra sentido la eficiencia en cuanto a las muestras necesarias para realizar una optimización.

Debida a la naturaleza multidisciplinar de esta trabajo, ha sido necesario aprender a utilizar XFlow y entender la terminología que se utiliza en campos como la mecánica de fluidos o la aeronáutica. Se han seguido los tutoriales y la documentación disponibles en XFlow (2), las cuales han servido como punto de partida de las simulaciones diseñadas. Como ejemplo de ello, la primera de las simulaciones diseñadas (ver apartado 6.1.) corresponde a una de las primeras simulaciones de los tutoriales consultados.

Además de ajustar la discretización de las celdas, otro parámetro a ajustar es el *time step* de la simulación, es decir, cuánto tiempo es simulado en cada instante. Para comprobar que la discretización y el *time step* configurados son los adecuados, durante la simulación, XFlow calcula la estabilidad de la simulación. Sin entrar en detalles, la estabilidad es un valor que depende del tiempo, el tamaño de celda y la velocidad y que permite comprobar la convergencia de la simulación. Está basada en la condición de Courant-Friedrichs-Lewy (14).

Si la estabilidad se encuentra cercana a 0, significa que la simulación converge lentamente, por lo que podríamos aumentar el *time step* o el tamaño de las celdas para

acelerar la simulación. Por otro lado, cuando el valor se aproxima o supera 1, la simulación puede no converger, por lo que lo recomendable es reducir el *time step* o el tamaño de las celdas.

Como la estabilidad depende del tiempo, el tamaño de celda y la velocidad del fluido, suele ser más cómodo controlar la estabilidad modificando el número de Courant, el cual especifica cuantas celdas puede avanzar una partícula por *time step.* Por defecto el número de Courant se encuentra a 1.

También se realizaron pruebas mediante optimización Bayesiana para encontrar el número de Courant adecuado para alcanzar cierto parámetros de estabilidad, ver en figura 4.a. No obstante, no se ha incluido en la experimentación debida a la necesidad de validarlo mediante un conjunto de simulaciones suficiente variadas.

Por último, destacar que la elección de estos parámetros en el trabajo ha tenido que realizarse cuidadosamente debido a que al optimizar tenemos que asegurar que cualquier simulación posible en la optimización tenga una convergencia adecuada. Además, los resultados de simulaciones tienen que ser consistentes entre sí, por lo que la configuración tiene que mantenerse intacta durante todas las optimizaciones realizadas, para que el análisis de los resultados de las optimizaciones en conjunto sea válido.

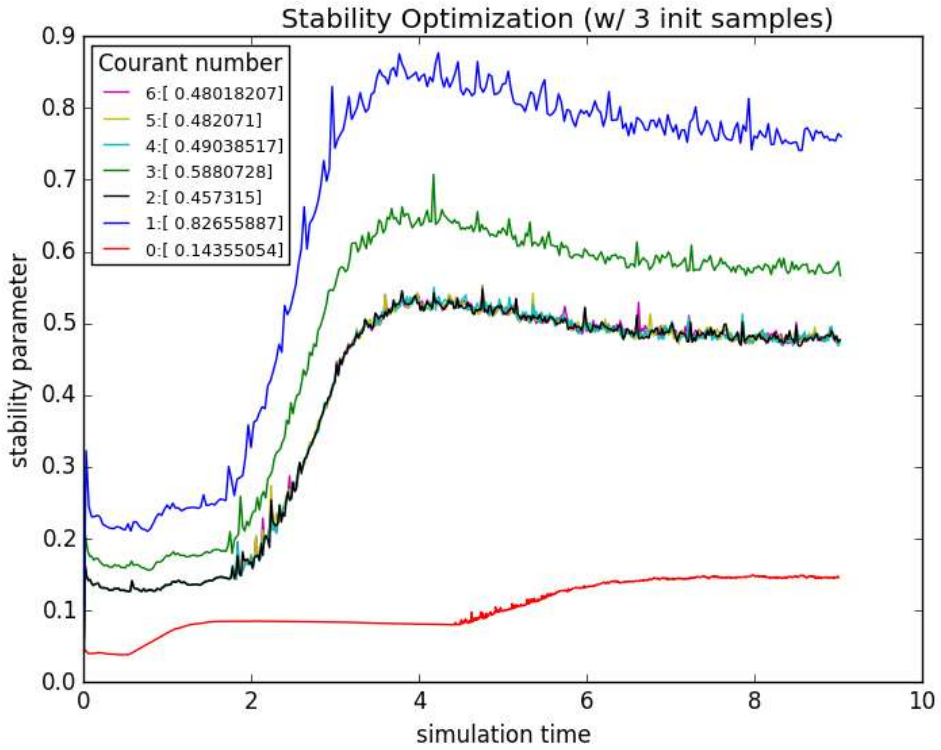


Figura 4.a. Optimización del número de Courant para encontrar un valor de parámetro estabilidad del 0.5. Para ello se utilizo mínimos cuadrados con respecto a la línea horizontal en 0.5. Está optimización en concreto se realizo para obtener de manera automática el número de Courant adecuado para el experimento de los aerogeneradores que, debido a su alta duración, finalmente fue desestimado (en el apartado 6 se comenta dicho experimento).

5. Integración

Nos referiremos a la integración entre BayesOpt y xFlow como el proceso de desarrollar una interfaz que gestione la comunicación entre ambos programas, que además permita modificar el comportamiento de la interacción en función de las particularidades de la simulación que se quiera optimizar. La interacción entre los componentes puede verse en el diagrama de secuencia de la figura 5.a.

La integración se ha realizado en Python, el principal motivo es reutilizar el script en Python proporcionado por el equipo de XFlow, el cual permite extraer los valores numéricos de distintas variables en cada instante de la simulación. El script extrae la información que se almacenan en ficheros binarios durante la simulación.

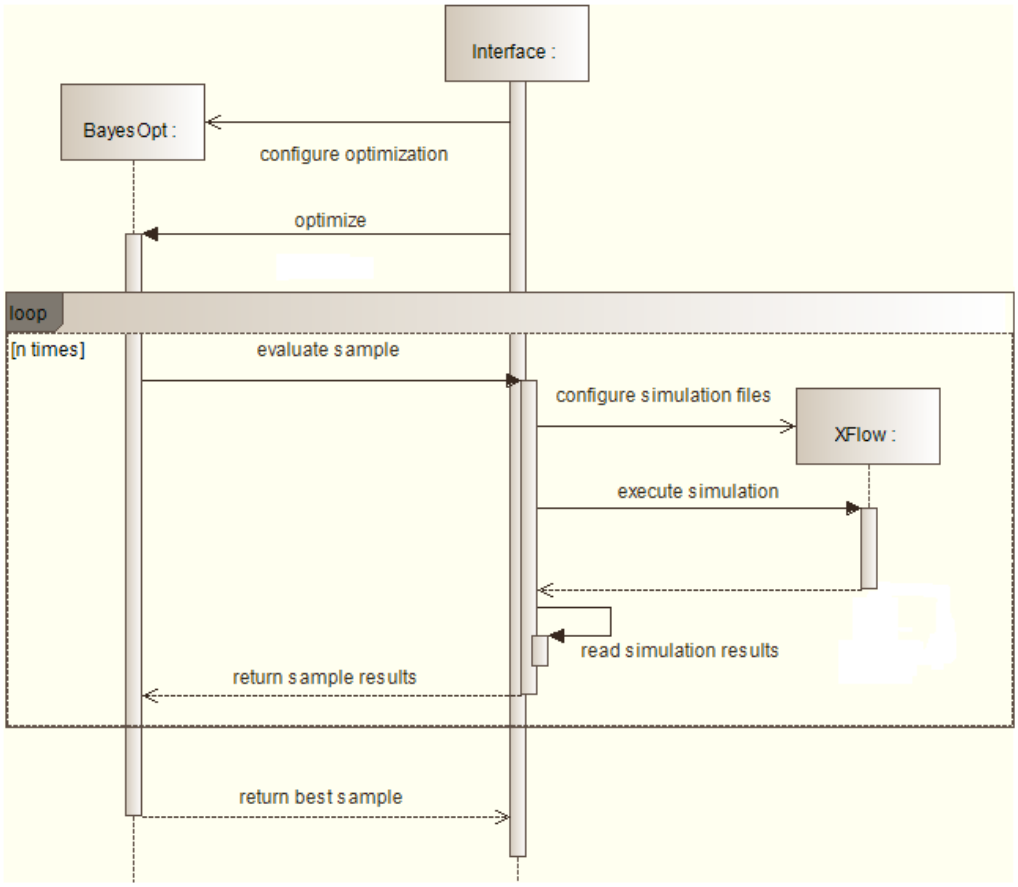


Figura 5.a. Diagrama de secuencia que resume la interacción entre los componentes. Empezamos en la Interfaz la cual configura los parámetros de la optimización y ejecuta BayesOpt (*optimize)*. Ahora BayesOpt controla la ejecución. El bucle que aparece en el diagrama corresponde al bucle de la optimización Bayesiana. En cada iteración BayesOpt evalúa la muestra (*evaluate sample)* llamando a la función programada en la interfaz. Esta función debe ocuparse de configurar la simulación (*configure simulation files)* y proceder a ejecutarla (*execute simulation*). Una vez terminada, es necesario recuperar los resultados a partir de los ficheros generados por la simulación. Se devuelve entonces el resultado de la muestra evaluada a BayesOpt (return sample results). El bucle se realiza un número determinado de iteraciones, una vez finalizadas se devuelve el resultado de la optimización (*return best sample)*.

Partiremos de la interfaz en Python disponible en BayesOpt. Esta interfaz requiere ejecutar la función *optimize* del modulo *bayesopt*:

optimize(func, nDim, lb, ub, params)

Es necesario indicarle una función (*func*) que actuará como función de coste. *nDim* corresponde al número de dimensiones, *lb* y *ub* corresponden a la cota inferior y cota superior de cada dimensión del espacio de puntos y *params* es el conjunto de parámetros de una optimización BayesOpt.

Cada vez que BayesOpt quiera evaluar una muestra llamará a la función indicada con el punto a evaluar (el conjunto de parámetros de entrada) y la función deberá devolver un único valor numérico como resultado. Como el objetivo es poder optimizar simulaciones de XFlow, esa función tiene que programarse para: aplicar los parámetros a la simulación XFlow, ejecutar un proceso que lleve a cabo la simulación XFlow y recuperar los resultados y sintetizarlos en un valor numérico como resultado. En los siguientes párrafos se cubrirá parte de la funcionalidad que ayuda a programar la función con los pasos mencionados

Modificar el comportamiento de una simulación en XFlow consiste, principalmente, en modificar el fichero XML que almacena la configuración del proyecto de la simulación. Así pues, si se quisiese cambiar la posición de un objeto particular de la simulación, habría que identificar en el XML el campo determinado que controla la posición del objeto e insertar el valor necesario. Como el objetivo es modificar automática la configuración en función de la muestra a evaluar, tenemos 2 alternativas para la modificación automática de los valores: programar la modificación del XML mediante un parser de Python o crear una plantilla etiquetada.

La plantilla etiquetada consiste en crear copia del fichero XML pero sustituyendo los campos a modificar por etiquetas (por ejemplo: "XXX_0" para el primer parámetro). Dada la plantilla y el conjunto de parámetros se puede generar con facilidad un nuevo fichero XML para que XFlow utilizará para simular mediante la función *create_from_template()*. Para un posible usuario final, esta alternativa es mucho más cómoda que requerir la programación explicita de los campos a modificar del XML. Podemos ver un fragmento de un fichero etiquetado y un fragmento del fichero generado en la figura 5.b.

```
<Behaviour type="Fixed">
  <Scale>1</Scale>
  <Position>(0,0,0)</Position>
  <Orientation>(0,XXXX_0,0)</Orientation>
</Behaviour>
```

```
<Behaviour type="Fixed">
  <Scale>1</Scale>
  <Position>(0,0,0)</Position>
  <Orientation>(0,47.3295,0)</Orientation>
</Behaviour>
```

Figura 5.b. Ejemplo de uso del etiquetado, a la izquierda parte del XML etiquetado, en la segunda componente de la orientación hemos incluido la etiqueta "XXXX_0", indicando que se introduzca el valor del primer parámetro de entrada. A la derecha el valor sustituido.

El siguiente paso, una vez modificada la configuración de XFlow de acuerdo a los parámetros de entrada, es ejecutar la simulación. Esta se tiene que realizar mediante una serie de invocaciones de procesos a ejecutables de XFlow, los cuales se han encapsulado en la función *execute()* para facilitar la ejecución.

Una vez finalizada la ejecución de la optimización, hay que recuperar los resultados generados por la simulación, mediante la función *loadnumdata()* permitirá acceder a cada variable en cada instante de la simulación (variables como: fuerzas, momentos, velocidades...). Dichos resultados se han de sintetizar en un único valor, por lo que es común utilizar la media de os valores en los instantes finales de la simulación, puesto que las variables se han estabilizado.

Una vez devuelto el resultado de la muestra evaluada, BayesOpt tomará el resultado y lo utiliza para aprender el nuevo modelo y seleccionará la siguiente muestra a evaluar.

Además, para mejorar la facilidad de uso, se han implementado 2 características en la interfaz. La primera de ellas y siguiendo un esquema orientado a objeto (OOP), se ha facilitado una clase que en el momento de instanciarla se le incluye gran parte de la información necesaria (directorio de instalación de XFlow, directorio del proyecto, ficheros específicos...). Esto permite evitar la repetición de argumentos en las distintas llamadas a funciones, puesto que dicha información se almacena en la clase. Las funciones siguen siendo accesibles directamente sin utilizar la clase, pero es necesario incluir toda la información que la función necesita.

La segunda de las características es el uso de decoradores o *decorators* de Python. Básicamente permite el uso de etiquetas sobre la declaración de funciones, algo sencillo de comprender y utilizar por un usuario (ver ejemplo en figura 5.c). Al desarrollador le permite añadir funcionalidad sobre la función etiquetada por el usuario, como por ejemplo: almacenar la función para llamarla en otro momento, conocer cuando se llama a una función y cuando la función termina, entre otros.

```
@xflow.input_modifier()
def modify_input(Xin):
  return [-30 + x * 60 for x in Xin]
```

Figura 5.c *Decorator* aplicado a la función modify_input(Xin). Se indica mediante "@" antes de la función sobre la que debe actuar. En este caso xflow.input_modifier() permite suscribir la función modify_input(Xin) para que sea utilizada automáticamente para mapear el espacio de valores.

Se ofrecen 2 *decorator* en la interfaz. El primero *configure_log()*, se utiliza etiquetando la función de coste que está siendo optimizada. Permite estructurar mejor los *logs* generados (se incluyen cabeceras entre cada simulación en el fichero de *log*) y automatizar la copia de los ficheros binarios de resultados, ya que son sobrescritos en cada simulación realizada.. El segundo *decorator input_modificer()*, permite etiquetar una función para que sea dicha función la que modifique los parámetros de entrada que llegan desde BayesOpt. El principal motivo de utilizar *decorators* de Python es debido a que permiten añadir funcionalidad alrededor de código definido por el usuario sin necesidad de complicar el código ya existente de la interfaz.

En este apartado se ha explicado cómo funciona la interacción y la programación de la interfaz para una simulación que solo requiera modificar lo fundamental. Existen más funcionalidad de las que no se ha hablado, por lo que se recomienda consultar en el Anexo I para más información.

6. Resultados

Las optimizaciones realizadas en este trabajo se han diseñado con el propósito de verificar que la interfaz desarrollada funcione correctamente y se han diseñado problemas fáciles de analizar pero con cierto realismo para poder validar futuras aplicaciones.

A pesar de que el motivo de usar optimización Bayesiana es el de poder optimizar simulaciones que requieran un elevado tiempo de ejecución, el tiempo de ejecución necesario debe ser razonable para poder repetir múltiples veces durante el desarrollo y permitir realizar suficientes optimizaciones como para asegurar que las optimizaciones realizadas convergen hacía el mínimo global, independientemente de las muestras iniciales y la aleatoriedad de métodos como el MCMC (Markov Chain Monte Carlo) (11).

Se han diseñado y analizado 2 simulaciones: optimización del ángulo de un cilindro minimizando las fuerzas de arrastre y optimización de la forma de un ala, de tal manera que genere un mínimo de fuerzas de sustentación y minimice las fuerzas de arrastre. Ambas optimizaciones se cubrirán en los apartados 6.1. y 6.2.

Cabe destacar que también se diseñó una optimización que, debido al elevado tiempo de computación de 28 horas por simulación, se desestimó (ver explicación en figura 6.a.). El fichero de la interfaz de esta optimización es reutilizado en el Anexo I como ejemplo para explicar la interfaz)

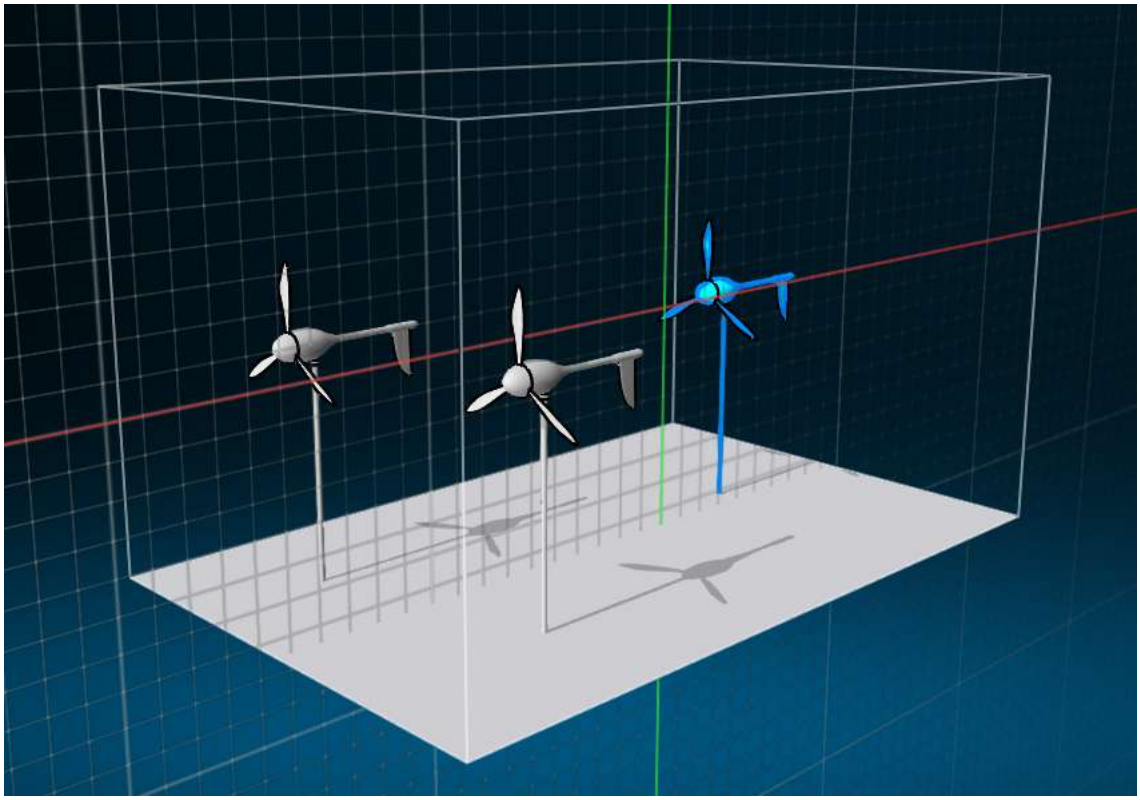


Figura 6.a. Muestra la optimización descartada de los aerogeneradores. Partiendo de un túnel de viento (caja rectangular), se dispone un aerogenerador (marcado) que se encuentra parcialmente obstruido por otros 2 aerogeneradores (no marcados). El objetivo de la optimización es el de encontrar la posición (x,z) del aerogenerador que maximice la velocidad de las aspas.

Con el objetivo de mantener un tiempo de ejecución total razonable, las simulaciones de las optimizaciones se han configurado para ser ejecutadas en 2D, es decir, XFlow utiliza la información contenida únicamente en la intersección de un plano, permitiendo disminuir el tiempo necesario de cada simulación. El plano se puede ver en la figura 6.b.

También comentar que BayesOpt normaliza el espacio de entrada al intervalo (0,1) con el objetivo de mejorar la estabilidad de la optimización. Es por ello que en la interfaz se debe indicar el espacio correcto al que mapear los valores proporcionados por BayesOpt, ya sea directamente en los parámetros de configuración de BayesOpt (*lb* y *ub*) y que sea BayesOpt quien los mapee o mapearlos en la interfaz mediante una función que se ocupe de modificarlos. Se recomienda la primera alternativa, los parámetros de configuración de BayesOpt, pero en los experimentos desarrollados se ha preferido utilizar la segunda alternativa para mantener una consistencia con las herramientas de visualización, ya que reutilizan la función que se ocupa de mapear. En el Anexo I aparecen indicadas ambas aproximaciones.

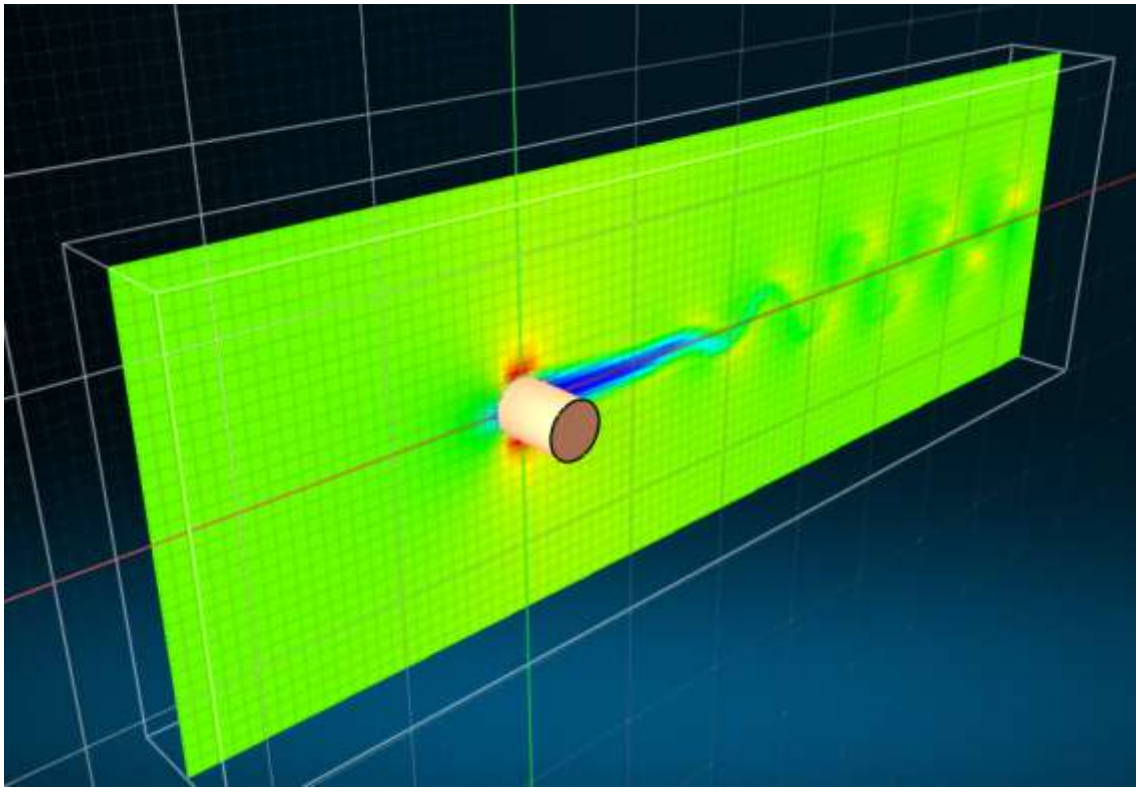


Figura 6.b. Los resultados de una simulación en 2D. El plano (predominantemente verde) intersecta con un cilindro (en la figura solo se aprecia la mitad del cilindro). Mediante el plano y la información de intersecciones se realiza la simulación en 2D, simplificando la simulación y, por tanto, el tiempo de simulación necesario.

Para completar la información en este apartado, indicar que los experimentos han sido ejecutados en la W*orkstation* Lenovo ThinkStation D30. Cuenta con procesador Intel Xeon E3-2620 v2 a 2.10 GHz de 6 núcleos físicos (2 núcleos lógicos por núcleo físico) y una memoria RAM de 64 GB DDR3.

6.1. Optimización del Ángulo de un Cilindro

6.1.1. Detalles de la Optimización

Partimos del ejemplo propuesto en los tutoriales de la documentación de XFlow, donde se ha configurado un túnel de viento en el cual se coloca un cilindro. El objetivo de la optimización será el de rotar el cilindro sobre el eje Y para encontrar el ángulo que minimice las fuerzas en el eje X sobre el cilindro.

Como el problema es simulado en 2D, por lo que a 0 grados se proyectará un circulo sobre el plano XZ. Conforme se aumenta el ángulo, el cilindro proyectará una elipse sobre el plano XZ. A medida que se acerca a 90 grados, la cara del cilindro intersecta con el plano XZ, perdiendo la forma aerodinámica de la elipse y provocando repuntes en las fuerzas en X debido a las esquinas de la cara. En la figura 6.1.1.b. se pueden apreciar 4 casos distintos en función del ángulo.

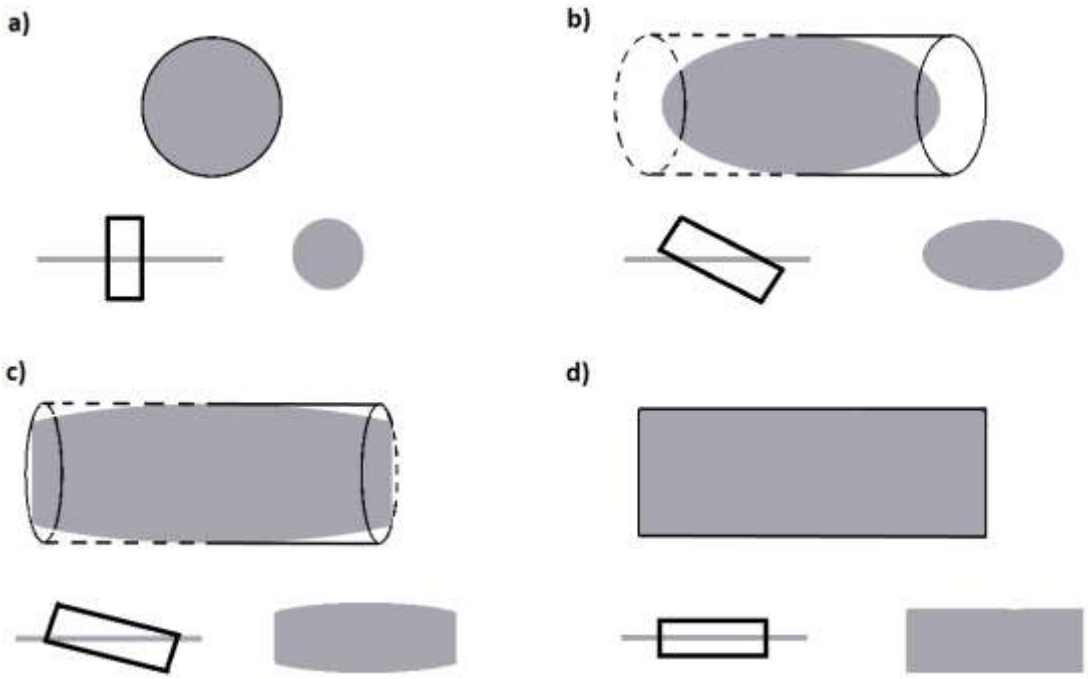


Figura 6.1.1.a. Cuatro imágenes, cada una representa la intersección de un cilindro con cierto ángulo. Los ángulos de cada imagen son: a) 0º, b) 60º, c) 75º, d) 90º. Para facilitar la compresión de la imagen, supongamos que el plano es la hoja de papel blanca de este trabajo, en la cual queremos atravesar un cilindro (líneas negras) con cierto ángulo. El área gris en la imagen es la intersección entre la hoja y el cilindro, es decir, la parte que sería necesaria recortar del papel para poder atravesar el cilindro. Para facilitar la visualización, bajo cada imagen aparece: la vista cenital de la intersección plano-cilindro (izquierda) y una miniatura con solo el área de intersección (derecha). Como la simulación es 2D, XFlow realizará la simulación únicamente con la forma de la intersección.

Por tanto, la optimización es únicamente de un parámetro, el ángulo entre 0 y 90. Este intervalo de ángulo se normaliza al intervalo 0 y 1 dentro de BayesOpt.

La función coste dependerá de las fuerzas en X sobre el cilindro. Como al inicio de la simulación la corriente de aire no ha interactuado con el cilindro, es necesario dar un margen de tiempo para que el aire interactúe con el cilindro y que las fuerzas en X se estabilicen. Por tanto, tomamos únicamente las fuerzas de la segunda mitad de la simulación, ya que representan mejor el valor correcto, ya que son valores más estables. De ese conjunto de muestras tomamos la media, que es devuelta como resultado de la muestra evaluada.

Utilizaremos el sistema de etiquetas de la interfaz para generar ficheros XML que XFlow utiliza para la configuración de la simulación. En esta optimización, etiquetaremos la rotación sobre el eje Y en la plantilla para que pueda ser manipulada según el ángulo que sea necesario aplicar.

También se ha de mapear los valores que nos proporciona BayesOpt entre 0 y 1 a los grados del ángulo. Bastará con multiplicar por 90, quedando el intervalo 0 y 90 grados que usará XFlow como rotación sobre el eje Y del cilindro.

6.1.2. Resultados Obtenidos

Puesto que se cuenta con un único parámetro a optimizar, es viable muestrear toda la función de coste mediante un muestreo en rejilla entre 0 y 90 grados. Se han repartido 100 muestras de manera uniforme entre los ángulos 0 y 90 grados. Este muestreo permitirá hacernos a la idea de la forma que presenta función coste que minimizaremos mediante optimización (ver figura 6.1.2.a).

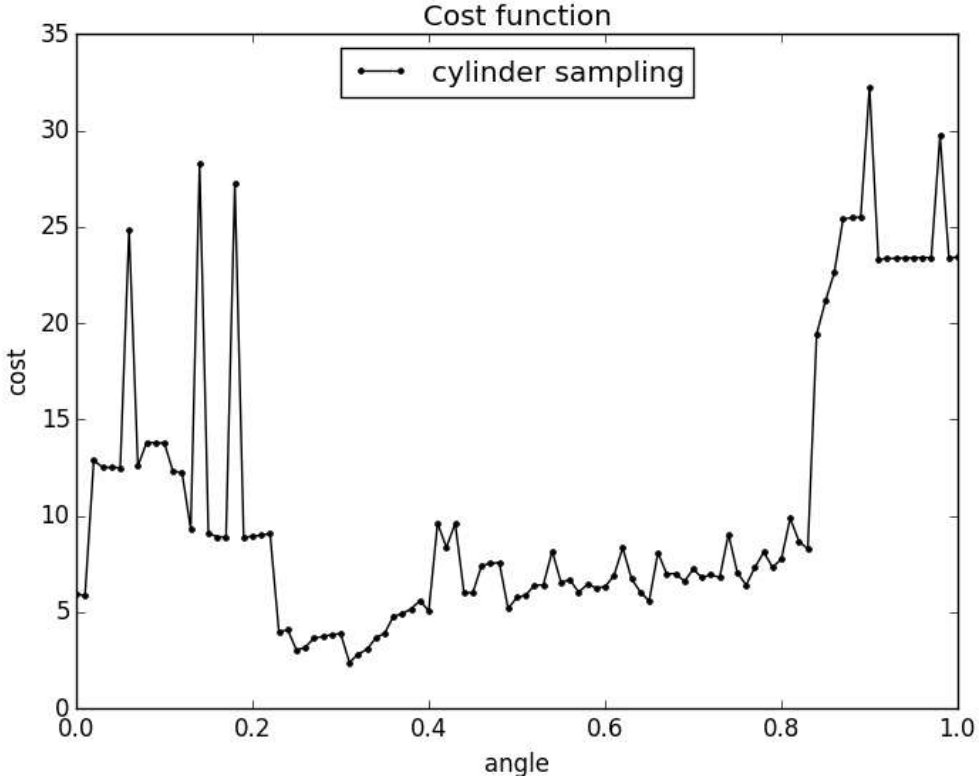


Figura 6.1.2.a. Muestreo de las fuerzas Fx sobre el cilindro para distintos ángulos. Se han utilizado 100 muestras repartidas de manera uniforme entre los ángulos 0 y 90 grados (normalizados en la gráfica entre 0 y 1). Esta gráfica permite hacernos a la idea de la forma que tiene la función que optimizaremos: existen múltiples mínimos locales y el mínimo global se encuentra aproximadamente a 1/3 del ángulo máximo.

La mejor muestra obtenida con el muestreo en rejilla (ver en figura 6.1.2.a.) se usará como cota de referencia con respecto a las optimizaciones realizadas. Utilizaremos graficas que nos muestren la mejor muestra obtenida hasta el momento en cada una de las iteraciones, es una métrica habitual que ayuda a comprobar la forma en la que progresan las optimizaciones. A lo largo del trabajo, nos referiremos a este tipo de gráficas como: gráficas coste-por-iteración. Podemos ver una gráfica de este tipo en la figura 6.1.2.b, donde aparece la cota de referencia y la mejor muestra en cada iteración de cada optimización realizada. En cada optimización cambia la semilla de aleatoriedad utilizada lo que repercute en el conjunto de muestras iniciales y los resultados mediante MCMC, lo que provoca que cada optimización sea diferente.

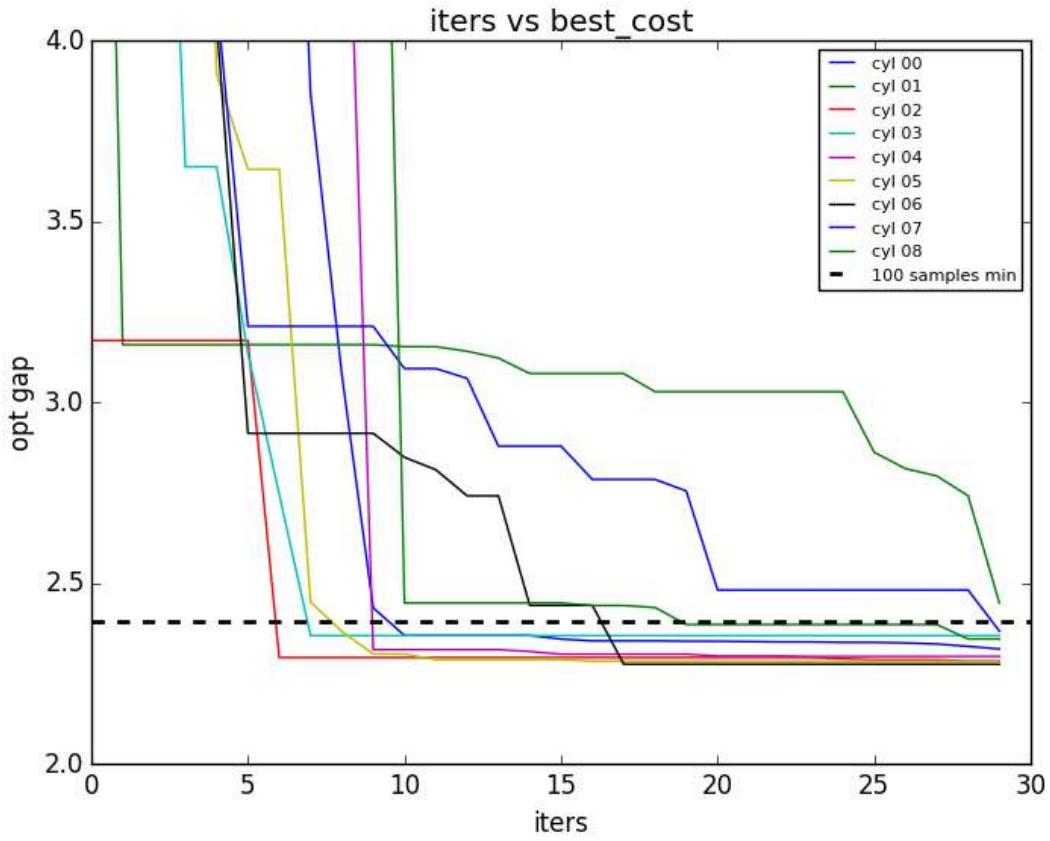


Figura 6.1.2.b. Gráfica coste-por-iteración. Este tipo de gráficas muestra la mejor muestra obtenida hasta cada una de las iteraciones. La línea negra puntuada es el coste de la mejor muestra obtenida de las 100 realizadas mediante el muestreo en rejilla. Las líneas coloreadas son cada una de las optimizaciones realizadas.

De la figura 6.1.2.b puede extraerse que, exceptuando una de las optimizaciones, el resto consigue superar la cota de referencia en menos de 30 iteraciones. Teniendo en cuenta que las 5 primeras iteraciones corresponden al muestreo inicial mediante *Latin Hypercube Sampling* (13), tenemos que solo con 25 iteraciones de optimización la mayoría alcanza y supera la cota de referencia obtenida mediante 100 muestras.

Para sintetizar mejor la información y poder analizar de forma general todas las optimizaciones, interpretaremos las funciones de la gráfica coste-por-iteración como una distribución t de Student (15) de todas las optimizaciones, de tal manera que obtenemos una

distribución de la media real, la cual tiene cierta media, cierta varianza y tantos grados de libertad como número de muestras (en este caso número de optimizaciones).

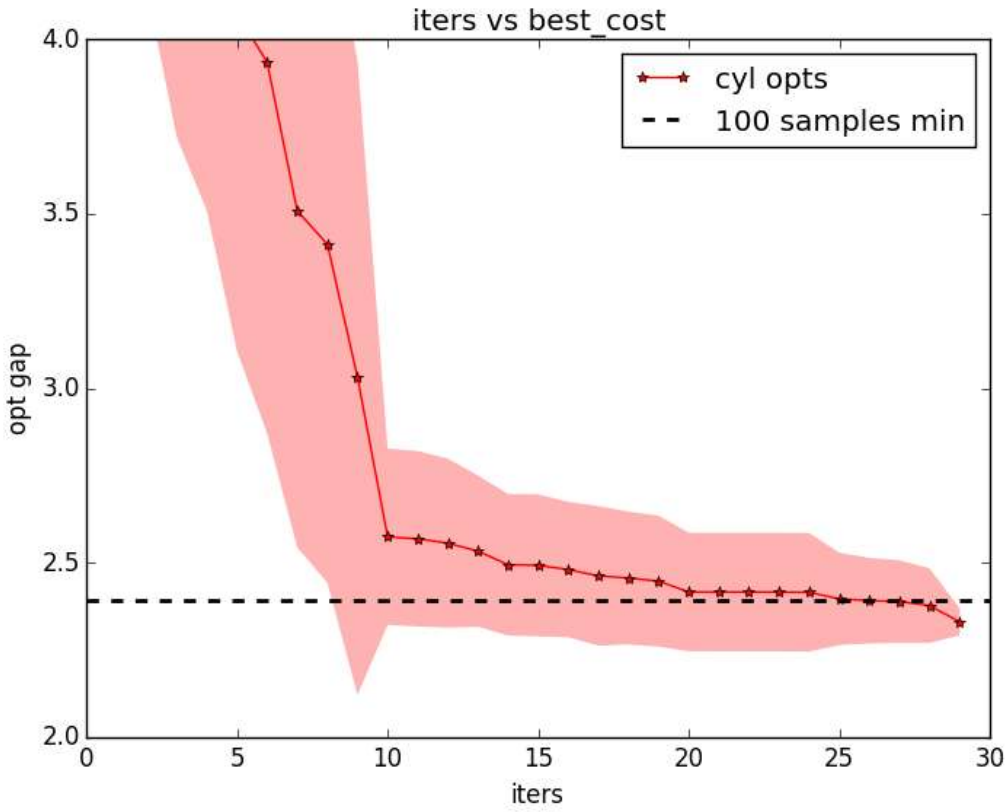


Figura 6.1.2.c. Gráfica coste-por-iteración de la media de todas las optimizaciones modelada mediante una distribución t de Student con un intervalo de confianza del 0.95. La media de la distribución ayuda a discernir como la media real de las optimizaciones progresa minimizando el coste y los intervalos de incertidumbre de la distribución permiten comprobar cómo se reducen la incertidumbre, es decir, la media real de las optimizaciones tiende a converger hacia un valor.

A partir de la figura 6.1.2.c. podemos observar que a medida que progresa la optimización, los intervalos de incertidumbre sobre el valor real de la media se reducen, signo claro de que las optimizaciones están convergiendo hacia el mismo valor, el mínimo. En la iteración 30 se puede observar que la incertidumbre deja de contener a la cota de referencia, indicando que existe una elevada probabilidad de que la media real supera la cota de referencia en 30 iteraciones.

En las figuras 6.1.2.d. se muestran los post-procesados realizados con XFlow de cilindros en los ángulos extremos: 90º y 0º. A continuación en la figura 6.1.2.e. se muestra el ala a 45º en comparación con el ala óptima, aproximadamente en 27º.

Por último concluir que no solo obtenemos una mejora con respecto a la cota de referencia, sino que también el número de muestras necesarias se reduce a menos de la tercera parte con respecto al muestreo en rejilla.

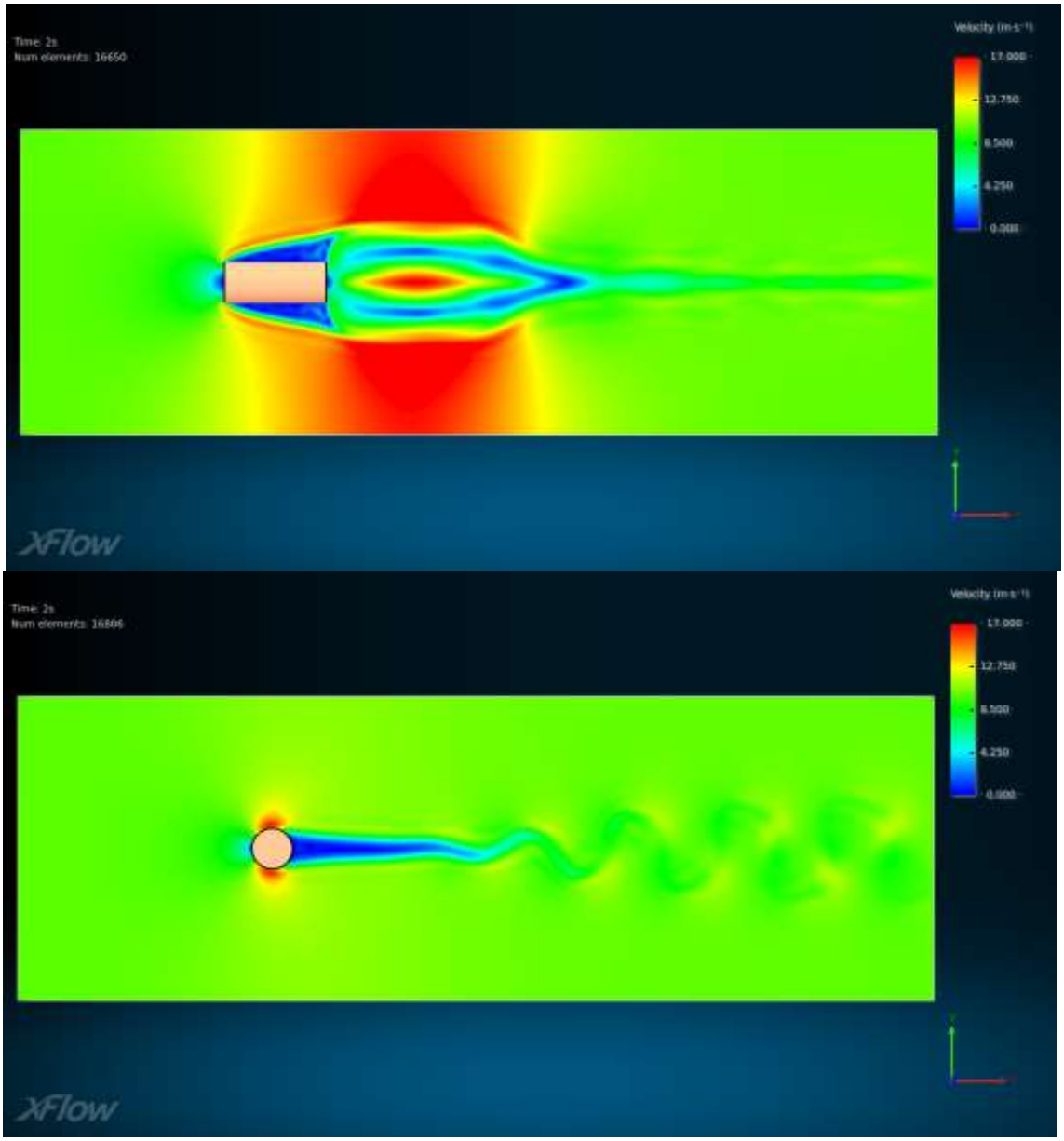


Figura 6.1.2.d. Muestra los ángulos extremos en la optimización. Se puede apreciar como ninguna de estas es la óptima en cuanto a la fuerza de arrastre que generan. Arriba, el cilindro a 90º, claramente se aprecia la resistencia. Abajo, el cilindro a 0º, se puede observar una menor fuerza de arrastre, pero sigue siendo elevada si la comparamos con otros ángulos, como los ángulos de la figura 6.1.2.e.

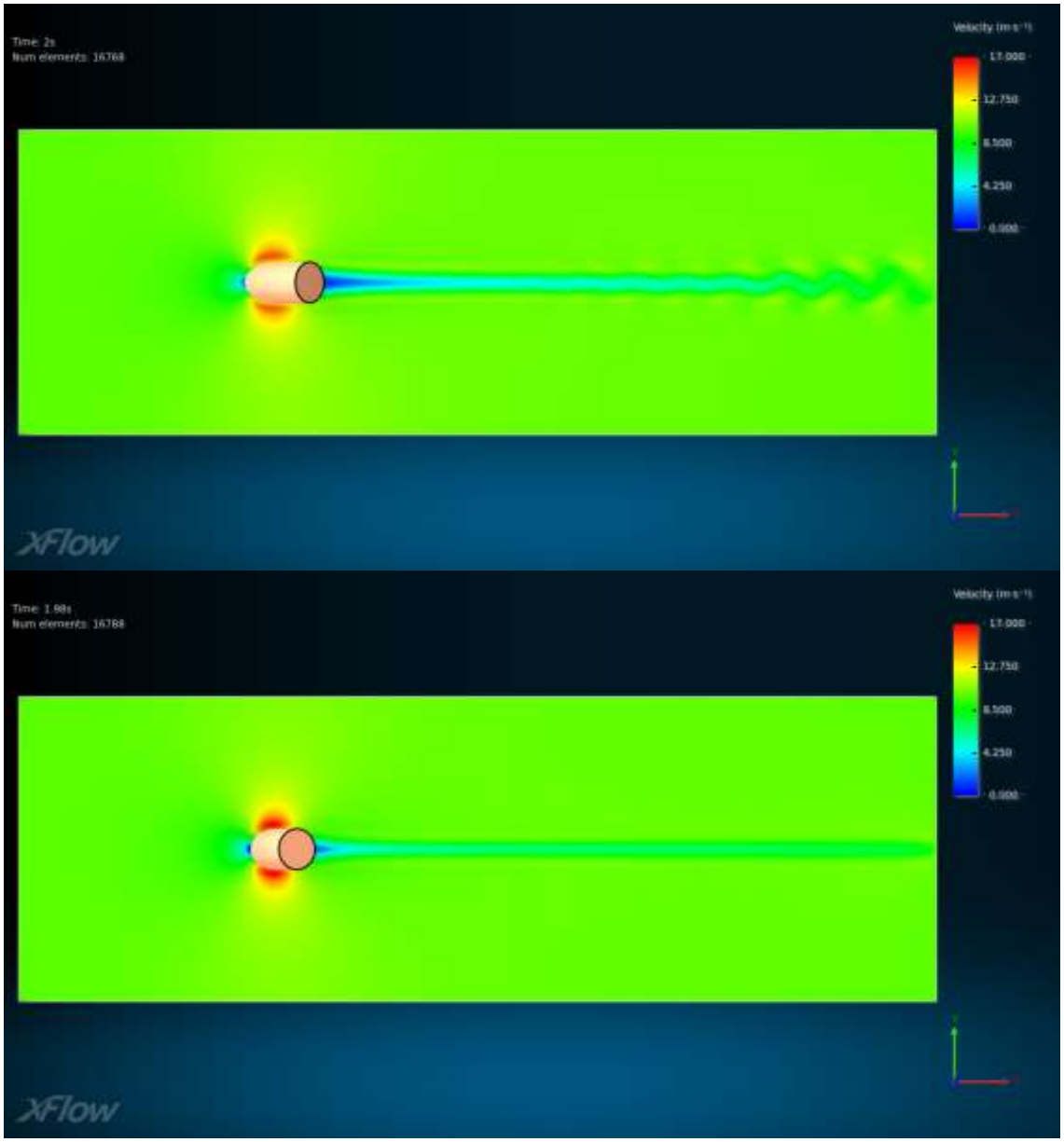


Figura 6.1.2.e. Muestra el ángulo de 45º (arriba) con respecto al ángulo óptimo (abajo) encontrado en la optimización. Aquí las diferencias son más sutiles, pero podemos apreciar como la estela que genera el ángulo de 45º es más ancha ya que está frenando más aire y por tanto las fuerzas de arrastre son mayores que las del ángulo óptimo. Además, al tener menor fuerza de arrastre, la estela generada en el ángulo óptimo es mucho más estable que el resto.

6.2. Optimización de la Forma de un Ala

6.2.1 Detalles de la Optimización

Esta optimización trata de diseñar la forma de un ala de avión con el objetivo de minimizar la resistencia al viento o *drag* restringiendo a que las fuerzas de alzamiento o *lift* tengan un cierto valor mínimo, ya que se espera que el ala sea capaz de elevar el resto del avión.

Hemos partido de los datos (16) de un avión ligero, el Cessna 172 Skyhawk (ver en figura 6.2.1.a), cuyo peso ronda entre 635 Kg (el avión vacio) y 1043 Kg (peso máximo para poder despegar). Tomando una envergadura de 11 metros y quitando aproximadamente 1 metro de la cabina de la envergadura, tenemos un ancho de 5 metros por ala, es decir 10 metros ambas alas.

Figura 6.2.1.a Imagen de un Cessna F172G de 1965 en pleno vuelo, es una de las variantes del modelo Cessna 172 Skyhawk.

Como la simulación es en 2D no existiría anchura, puesto que se simula sobre el plano XY, por tanto asumimos que los resultados están sobre la unidad Z=1, es decir anchura de la simulación 1 metro. A partir de esto y los datos mencionados en el párrafo anterior, tenemos:

$$1043\ kg * 9.8 \frac{N}{Kg} * \frac{1}{10} m^{-1} \sim 1000\ \frac{N}{m}$$

Por tanto, se ha elegido un valor mínimo de *lift* de 1000 Newton. Si el ala generase menos valor de lift, procedemos a penalizar el resultado. La función a minimizar es la siguiente:

$$y = Fx, \qquad si\ Fy \geq 1000$$

$$y = Fx * 100 + 1000, \quad si\ Fy < 1000$$

La penalización es 100 veces el resultado original, al cual se suma 1000 para que no interfiera con valores no penalizados con fuerzas Fx por debajo de 1000.

Se ha configurado un túnel de viento, en la que la densidad del material es igual a la del aire, y se ha dispuesto un objeto geométrico con forma de ala (ver figura 6.2.1.b.). En este

caso la velocidad del túnel de viento será mucho mayor, se aproximará a la velocidad que alcanzan las aeronaves ligeras. Se ha tomado una velocidad de 40 m/s. El valor se encuentra entre la velocidad mínima (22,8 m/s) y la velocidad máxima (77.8 m/s) del avión de referencia (16).

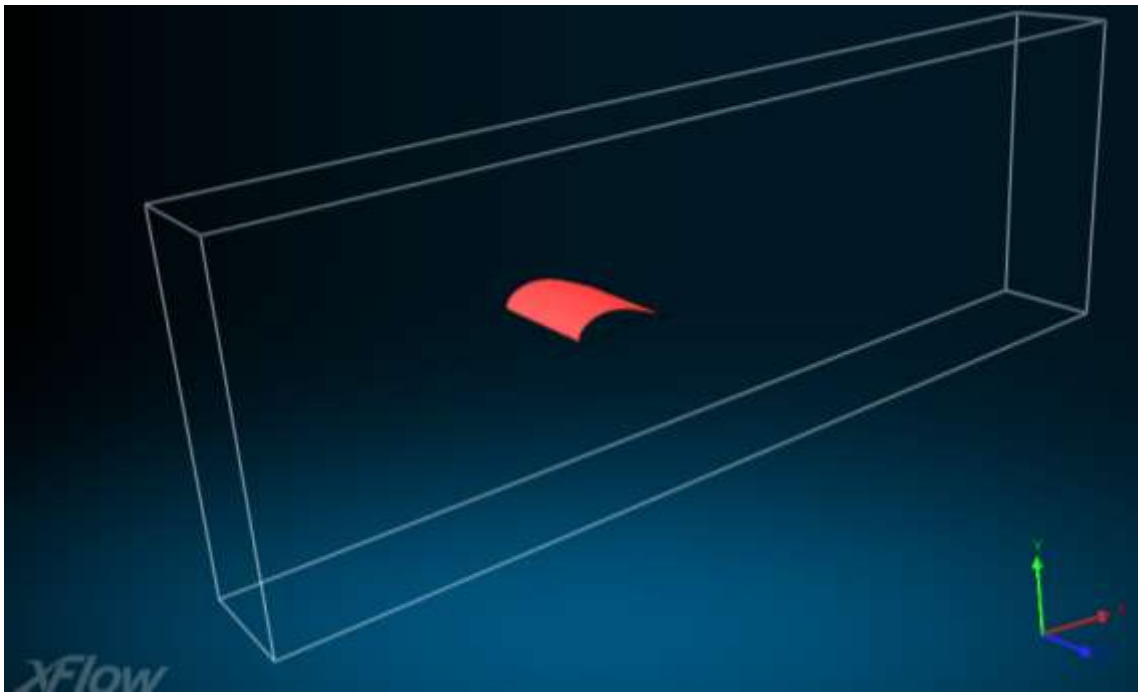


Figura 6.2.1.b Configuración de la simulación en XFlow de un ala en un túnel de viento. La caja determina el volumen del túnel de viento. Se introduce un modelo en 3D de un ala dentro del túnel de viento, la cual, en cada iteración de la optimización deberá ser sustituida por una nueva ala generada de acuerdo a los parámetros de entrada. Al ser una simulación 2D, solo se simulara en un plano en los ejes XY, dividendo la caja por la mitad y tomando únicamente la intersección entre el plano y el ala.

Como todos los parámetros afectan a la forma del ala, la interfaz debe ocuparse de proporcionar a XFlow un fichero que contenga el objeto geométrico con la forma descrita por los parámetros. La forma en la que se crea y se proporciona a XFlow se discute en el apartado 6.2.2.

6.2.2. Modelado Paramétrico

En este apartado se discute como se consigue, a partir de los parámetros, generar e introducir el objeto geométrico con forma de ala dentro de la simulación.

La forma de un ala es geométricamente compleja de expresar, es por ello que se trata de representar mediante un conjunto de parámetros que definan su forma (ver figura 6.2.2.a.).

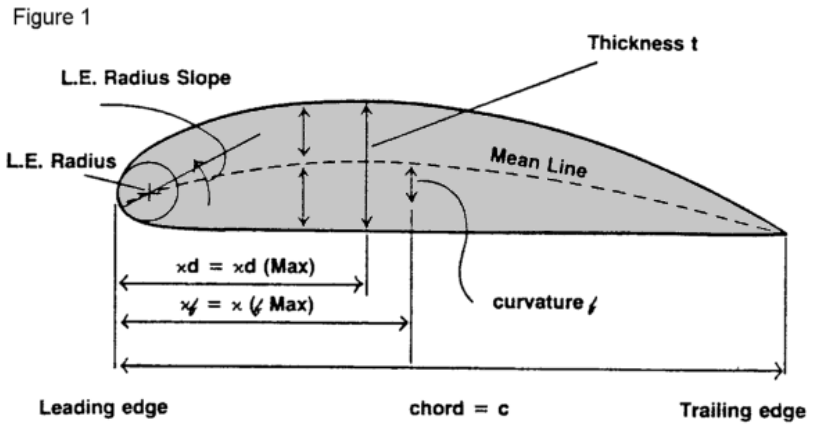


Figura 6.2.2.a. Descripción de la forma de un ala a través de diferentes características geométricas.

Aunque existen formulados diferentes modelos paramétricos, para este trabajo se ha decidido simplificar el ala como 2 líneas curvas que representan la parte superior e inferior del ala. Esta simplificación nos permite utilizar curvas paramétricas para definir cada una de las curvas. Concretamente se decidió utilizar curvas parametrizadas mediante puntos de control que influyen sobre la forma de la curva como son: NURBS, B-Splines o curvas de Bézier. Este tipo de curvas son comúnmente utilizadas en programas de diseño asistido por computadora o CAD.

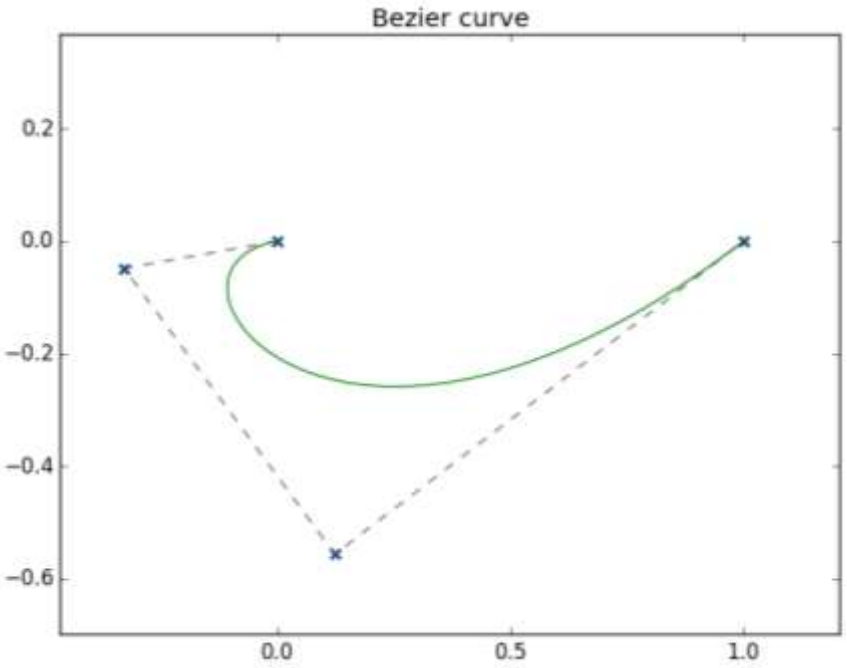


Figura 6.2.2.b. Curva de Bézier a partir de 4 puntos de control. Ofrecen suficiente libertad para generar curvas complejas modificando únicamente los puntos de control de manera interactiva.

Finalmente se decidió formular cada curva mediante una curva de Bézier (17) mediante 4 puntos de control (ver figura 6.2.2.b.), ya que: 4 puntos de control ofrecen suficiente libertad como para describir la forma de un ala con facilidad, son fáciles de manipular de manera interactiva y existen implementaciones disponibles para poder generarlas.

Con la manera de aproximarnos al problema decidida, se comenzó por desarrollar un prototipo para verificar la viabilidad de realizarlo. En la figura 6.2.2.c. muestra una captura del prototipo desarrollado. Este prototipo permite visualizar y manipular curvas de Bézier, por lo que con 2 curvas de Bézier se puede generar ambos perfiles de un ala en 2D

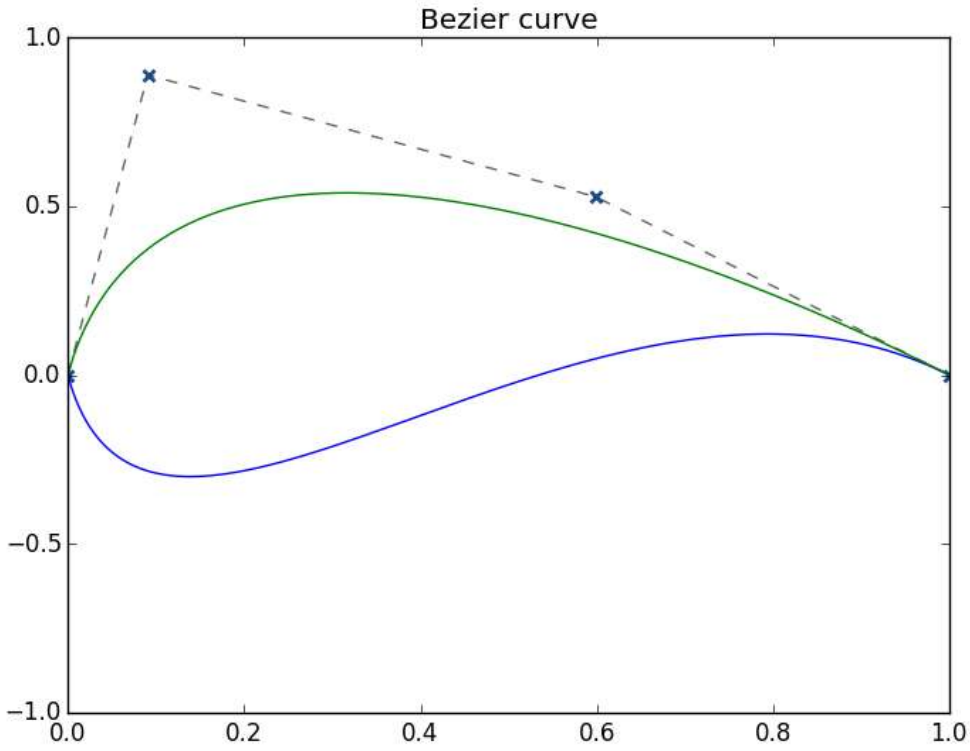


Figura 6.2.2.c. Captura del prototipo desarrollado en Python. Se muestra una aproximación de un ala generada de manera interactiva utilizando únicamente 2 curvas de Bézier de 4 puntos de control cada una. Este prototipo ha sido clave para decidir la configuración de los puntos de control, es decir, delimitar que posiciones pueden tomar los puntos de control de tal forma que los objetos geométricos siempre sean válidos (el perfil superior no intersecte con el perfil inferior).

Probando diferentes configuraciones de los puntos de control en la aplicación interactiva, dimos con una configuración que se asemeja a las alas de un avión. Partiendo de las 2 curvas de Bézier, fijamos los extremos frontal y trasero entre ambas de manera que coincidan entre sí, tal y como aparece en la figura 6.2.2.c.. El resto de puntos de control, aquellos que no son los extremos de la curva, los hacemos dependientes de 2 en 2 de la siguiente manera:

$$X_1^{superior} = X_1^{inferior}$$

$$Y_1^{superior} = -Y_1^{inferior}$$

$$X_2^{superior} = X_2^{inferior}$$

$$Y_2^{superior} = Y_2^{inferior}$$

Esta dependencia permite obtener el perfil inferior a partir del perfil superior, esto nos permite reducir a 4 parámetros: $X_1^{superior}$, $Y_1^{superior}$, $X_2^{superior}$, $Y_2^{superior}$. Los perfiles del ala mostrada en la figura 6.2.2.c. cumplen dicha dependencia.

Por último, se limitan los valores de los 4 parámetros para que los perfiles superior e inferior para generar alas con curvas suaves y evitar generar alas suficientemente finas.

El principal problema de definir el ala como 2 curvas de Bézier reside en la parte frontal, donde se unen ambos perfiles, el ala pierde la derivabilidad, en la figura 6.2.2.d. se aprecia el efecto.

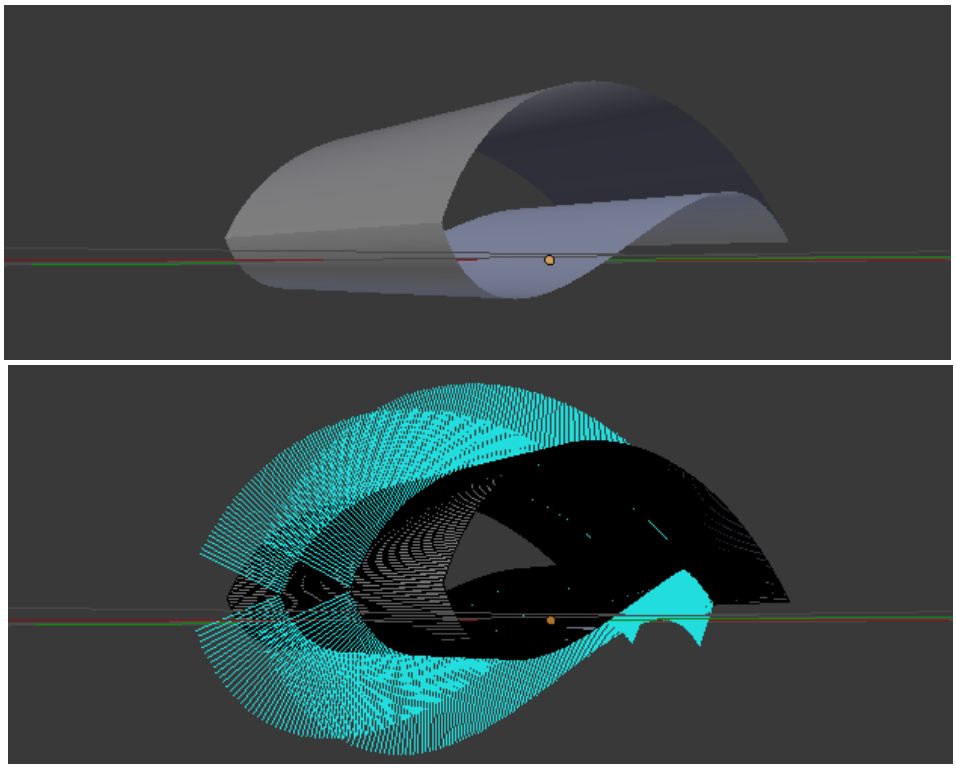


Figura 6.2.2.d. Visualización del ala, ya generada en 3D, en el programa Blender (4)**. Se puede apreciar en la imagen superior el punto donde se unen ambas curvas. Si activamos la visualización de las normales (imagen inferior) podemos apreciar una clara discontinuidad en la orientación de las normales, por tanto, deja de ser derivable en dicho punto.**

Para solucionarlo, en lugar de utilizar 2 curvas de Bézier de 4 puntos de control, utilizamos una única curva de Bézier de 7 puntos de control que empieza y termina en el final del ala. Con esto nos aseguramos la continuidad de la curva en la parte frontal, pero añade un nuevo punto de control que debe ser controlado, ver figura 6.2.2.e. Se decidió fijar Y=0 en ese punto y únicamente añadir la posición X como nuevo parámetro, lo que influye directamente en la longitud del ala generada.

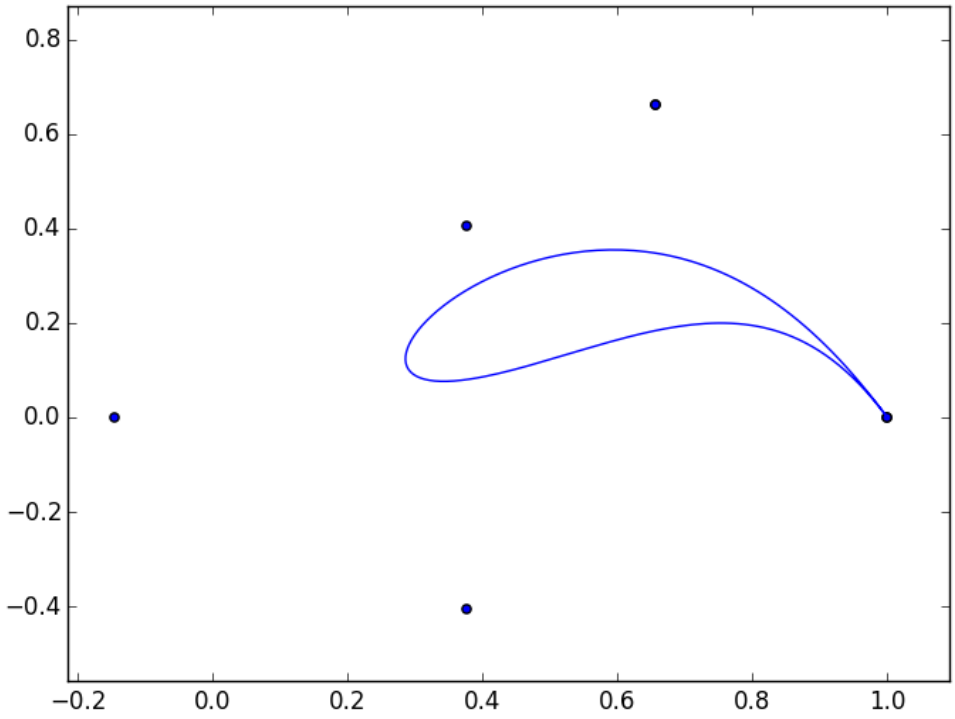


Figura 6.2.2.e. Muestra el ala generada mediante una curva de Bézier de 7 puntos, que ayuda a mantener la continuidad con respecto a la versión con 2 curvas de Bézier. El punto de control añadido es el que se encuentra a la izquierda, el cual es fijado a Y=0 y X es lo que se parametriza. Esta figura contiene los 7 puntos de control, pero hay 2 pares obstruidos entre sí (los 2 puntos a la derecha en realidad son 4).

El siguiente paso es el de generar un objeto geométrico en 3D a partir de los parámetros proporcionados. La manera de proceder es sencilla: con la curva de Bézier obtenida disponemos de una figura 2D a la cual le aplicamos una operación de extrudir para generar un objeto 3D (ver explicación en figura 6.2.2.f.).

El último paso es el generar un fichero que represente el objeto geométrica 3D planteado cuyo formato sea admitido por XFlow. Se implemento un generador de ficheros binarios de formato STL (18), el cual es un formato que describe la geometría únicamente mediante una malla de triángulos con sus respectivas normales. Aunque XFlow permite utilizar STL, durante la importación se genera un nuevo fichero en formato propio de XFlow. Por ello es imposible sustituir el fichero con el nuevo objeto geométrico en cada simulación a ejecutar mediante la interfaz, ya que exige importarlo desde la GUI del programa.

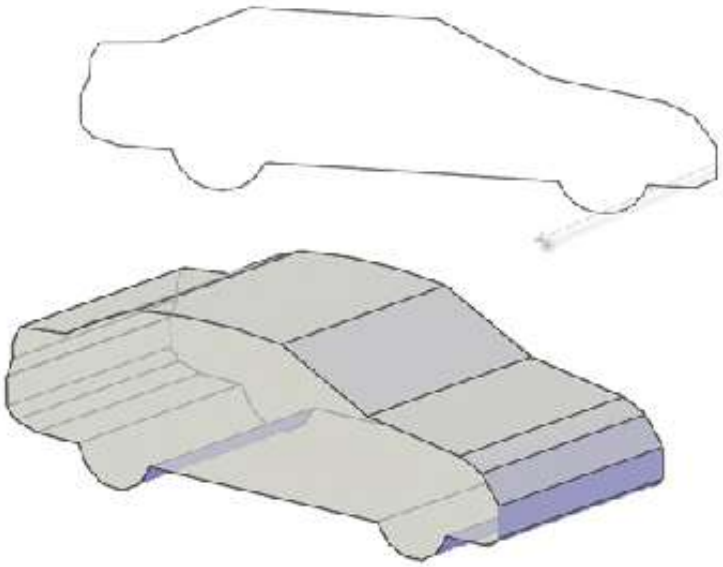


Figura 6.2.2.f. Operación de extrudir (*extrude*) aplicada a la sección de un coche. Los vértices y aristas originales se duplican y se trasladan. El espacio entre las aristas originales y las aristas duplicadas se rellena mediante caras, dando lugar a una representación 3D. La fuente original de la imagen pertenece a la documentación de AutoCAD (19)**.**

Como solución, el equipo de XFlow propuso utilizar el formato STEP (20), los cuales si son usados directamente por XFlow, ya que no necesita generar otro fichero para poder utilizarlos. Los ficheros STEP son utilizados por programas tipo CAD para representar objetos e intercambiar información. Puesto que la semántica del formato es amplia y compleja (21), en lugar de implementar completamente un generador de ficheros STEP, se ha utilizado un programa CAD para generar un fichero STEP que defina una superficie de Bézier mediante 7x2 puntos control. Concretamente se ha utilizado el programa FreeCAD (5).

A partir del fichero STEP generado de forma manual, la interfaz solo deberá modificar las partes de dicho fichero donde se especifican las posiciones de los puntos de control de la superficie de Bézier y sustituirlos por los indicados por los parámetros. En cada simulación, la interfaz deberá sobrescribir el fichero STEP existente por el fichero generado (ver figura 6.2.2.g.)

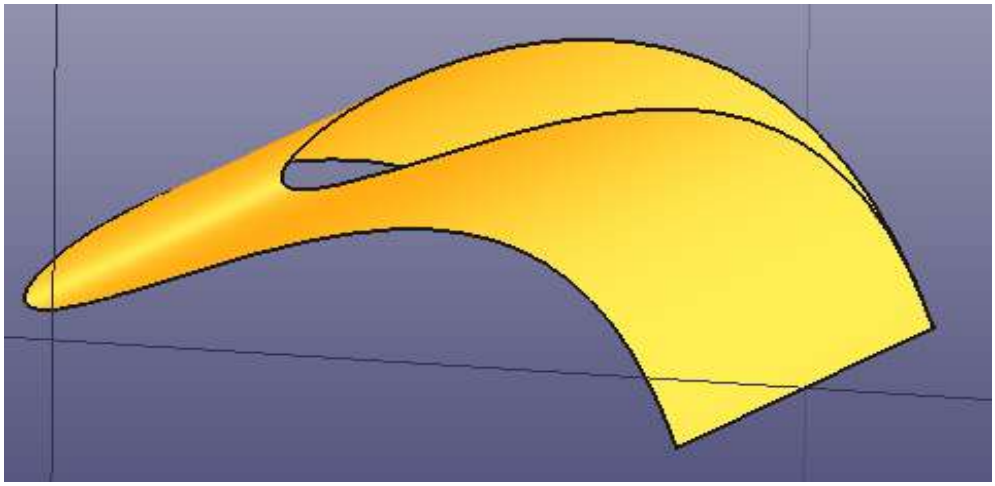


Figura 6.2.2.g. Esta figura muestra un ala generada mediante una superficie de Bézier de 7x2 puntos de control en el programa FreeCAD (5)**. Guardando el ala en un fichero STEP podemos utilizarlo como base para generar otras alas. Únicamente será necesario modificar los puntos de control de la superficie de Bézier del fichero de acuerdo a los parámetros y sobrescribir el fichero STEP que use la optimización por el recién creado.**

6.2.3. Resultados Obtenidos

Uno de los problemas más críticos de esta simulación es la penalización aplicada. Idealmente, habría que realizar una optimización con restricciones (22), pero ese aspecto de optimización Bayesiana todavía está siendo investigado. Por ello se ha planteado una solución intermedia mediante la penalización de valores (explicada en el aparatado 6.2.1). Es indispensable aplicar la penalización para poder dar con alas que realmente vuelen, es decir, que generen suficiente *lift*. El problema es que se está introduciendo una discontinuidad en la función de coste. Generalmente las muestras con menor *drag* son las que se encuentran en el límite de ser penalizadas. Por tanto, si existen muestras penalizadas al otro lado de limite, pueden perjudicar a la selección de la muestra.

Esto es debido a que los *kernel* utilizados no son capaces de interpretar la penalización, por lo que al penalizar (dar un elevado coste a una muestra), se está introduciendo información contraproducente: los puntos que rodean al punto penalizados también se considerarán penalizados (se considera que su coste es elevado).

Una limitación de los procesos Gaussianos es que no son capaces de generar modelos adecuados ante la existencia de gran cantidad de espurios (no confundir espurios con muestras con ruido). En este caso las muestras penalizadas actúan como espurios, ya que no representan correctamente a la función subyacente, esto limita la convergencia de las optimizaciones mediante procesos Gaussianos(ver figura 6.2.3.a.).

Para tratar de solucionarlo, hemos analizado los resultados con otras 2 *surrogate model* distintas a la procesos Gaussianos: Procesos de t de Student donde la distribución a priori de los hiperparámetros de media y varianza del proceso usan como distribución conjunta a priori una tipo Jeffrey (JEF) o una normal-gamma inversa (NIG).

La principal desventaja de ambos modelos frente a procesos Gaussianos es que tardan más en converger debido a que tienen que aprender las distribuciones sobre la media y la

varianza, pero eso a su vez es su ventaja, ya que al aprender dichas distribuciones puede dejar fuera de la regresión los datos espurios.

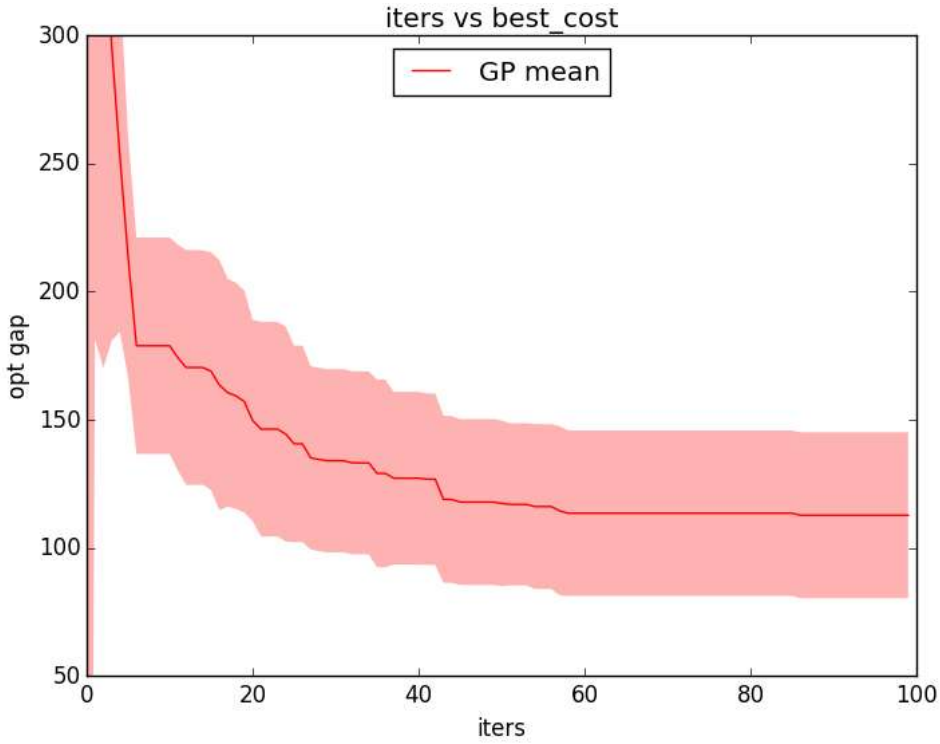


Figura 6.2.3.a Muestra los resultados de la optimización Bayesiana mediante procesos Gaussianos (GP). La figura muestra la media de coste-por-iteración modelada mediante una t-Student al 0.95 de intervalo de confianza. Se puede observar cómo, a pesar de que la optimización avanza, a mitad del total la media de la distribución apenas mejora. La incertidumbre tampoco se reduce a partir de ese momento.

Tanto JEF como NIG parten de la misma premisa que procesos Gaussianos, modelar la distribución a posteriori como una distribución de funciones con cierta media y varianza. La diferencia reside en la media y varianza utilizada, en lugar de ser un valor determinado como en procesos Gaussianos, se incorporan distribuciones sobre la media y la varianza de tal forma que la distribución resultante es conocida: una distribución t-Student. Dado que aprenden la distribución de la media y la varianza, son capaces de dejar fuera de la regresión los espurios.

En cuanto a la diferencia entre JEF y NIG, la primera es una distribución a *priori* no informativa, es decir, expresa de manera vaga la información acerca de una variable (23). NIG, por su parte, requiere indicarle los parámetros de la normal que son la media y la desviación típica de la función paramétrica

Se han comparado los resultados mediante JEF (figura 6.2.3.b) y NIG (figura 6.2.3.c) con respecto a procesos Gaussianos. No se han analizado todos en la misma gráfica debida a la confusión que provocan la mezcla de colores. Se puede observar que JEF y NIG son más consistentes en cuanto a que van mejorando conforme avanzan y su incertidumbre es menor.

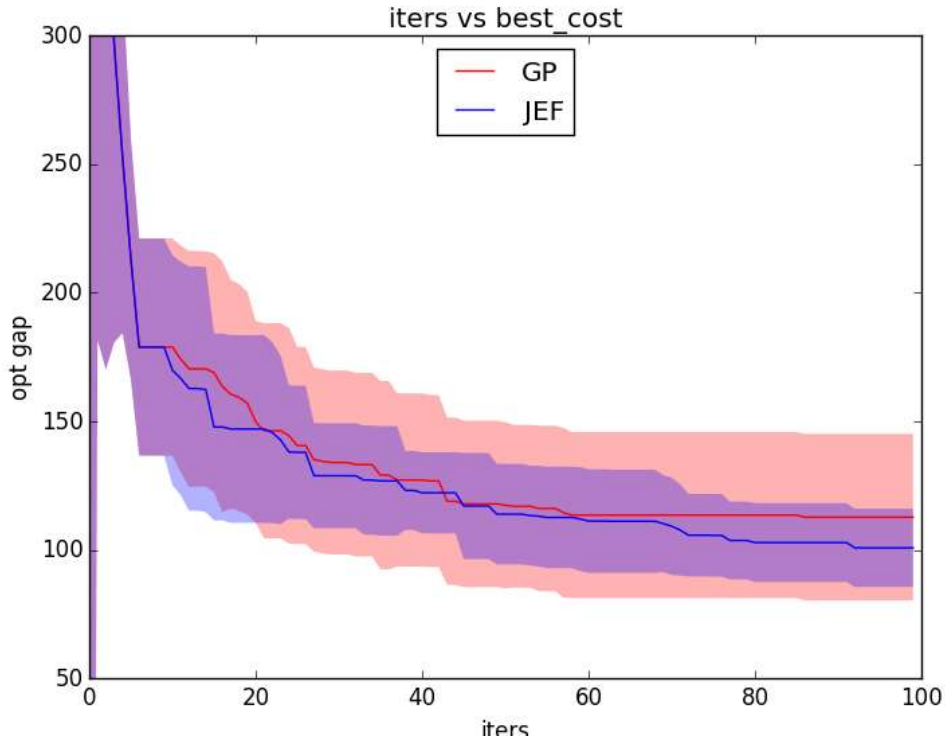


Figura 6.2.3.b Comparación de procesos Gaussianos (GP) con respecto a *Jeffrey's prior* (JEF). La principal mejora se produce en la incertidumbre, donde en JEF se reduce mucho más rapido que GP. Además JEF es más robusto que GP y, por tanto, la media de la distribución resultante es mejor en JEF que en GP.

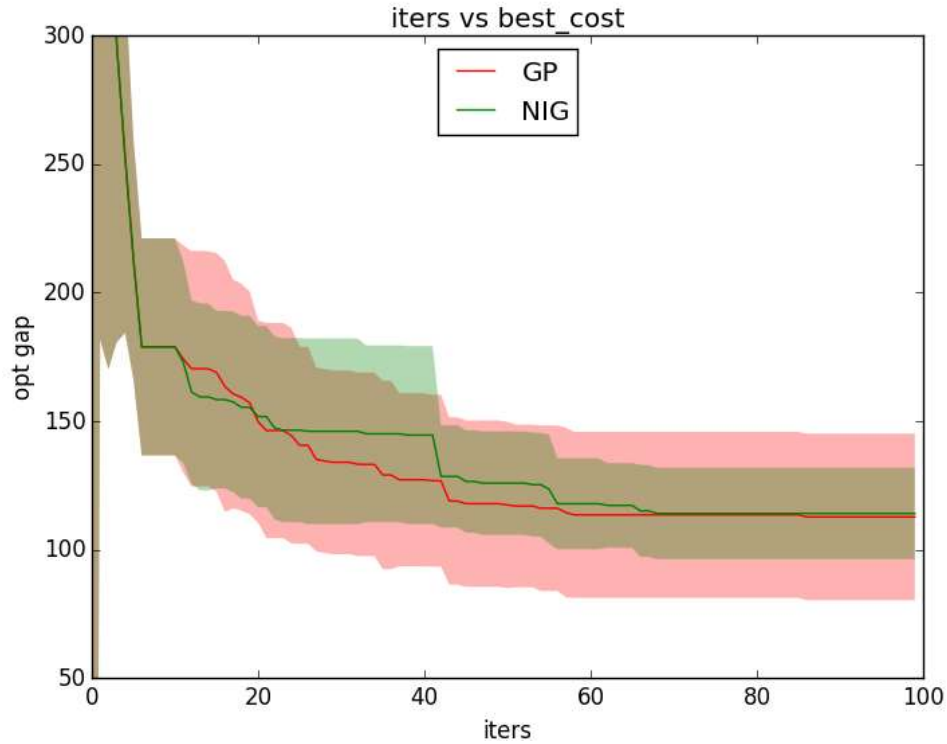


Figura 6.2.3.c Comparación de procesos Gaussianos (GP) con respecto normal-inverse gamma (NIG). A pesar que la media de la distribución es similar en GP y NIG, hay que tener en cuenta que la incertidumbre sobre la media real en NIG es notablemente más reducida que en GP.

En GP (procesos Gaussianos), como ya se ha comentado anteriormente, las datos espurios pueden deteriorar completamente el modelo, en este caso generando optimizaciones que dejan de mejorar a mitad del total de iteraciones y cuya incertidumbre sobre la media real es elevada.

En JEF (*Jeffrey's prior*), a pesar de que al tener que aprender más que GP debería avanzar más lento, debida a la robusted que proporciona frente a espurios alcanza y supera el rendimiento de GP, tanto en la media como en la incertidumbre de la distribución sobre la media real.

En NIG (normal-inverse gamma), al igual que JEF, tiene que aprender más y debería avanzar más lento, pero alcanza unos resultados similares a GP pero con una incertidumbre sobre la media real mucho menor. La media de los resultados obtenidos es peor respecto a JEF, lo que puede ser debido a que los parámetros utilizados $\alpha=1$ y $\beta=1$ de la distribución inversa de gamma no son los adecuados.

En la figura 6.2.3.e. se muestra un ejemplo de un ala penalizada en la optimización. En la figura 6.2.3.f. se muestra la mejor ala de una de las optimizaciones.

Concluir que, aunque todavía no se disponen de técnicas de optimización con restricciones en optimización Bayesiana, es capaz de generar alas con formas variadas pero que siempre son válidas (no penalizadas) y que convergen en mayor o menor medida hacia los mismos valores de *drag* (valor que está siendo optimizado).

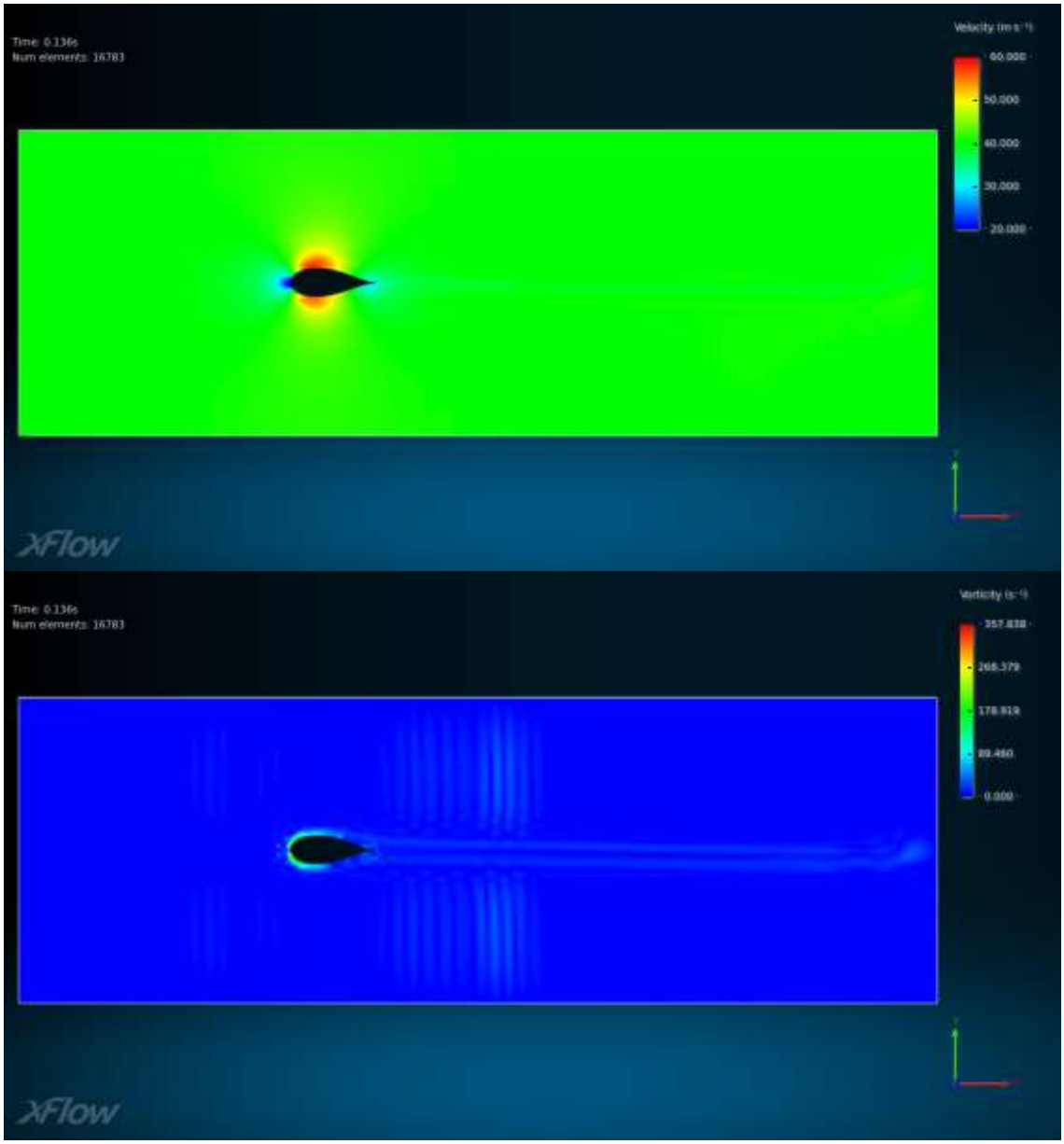


Figura 6.2.3.e. Muestra el post-procesado de un ala penalizada. En la imagen superior se muestra la velocidad y en la imagen inferior la vorticidad. A pesar de ser un ala con muy poco *drag*, se aprecia que la forma del ala generada es simétrica, por lo que nunca generará lift ya que el flujo de aire que pasa por encima y el que pasa por debajo del ala circulan a la misma velocidad (se puede ver en la vorticidad que la estela generada esta al mismo nivel que el ala y las franjas verticales están alineadas, signo claro de que no existe *lift*).

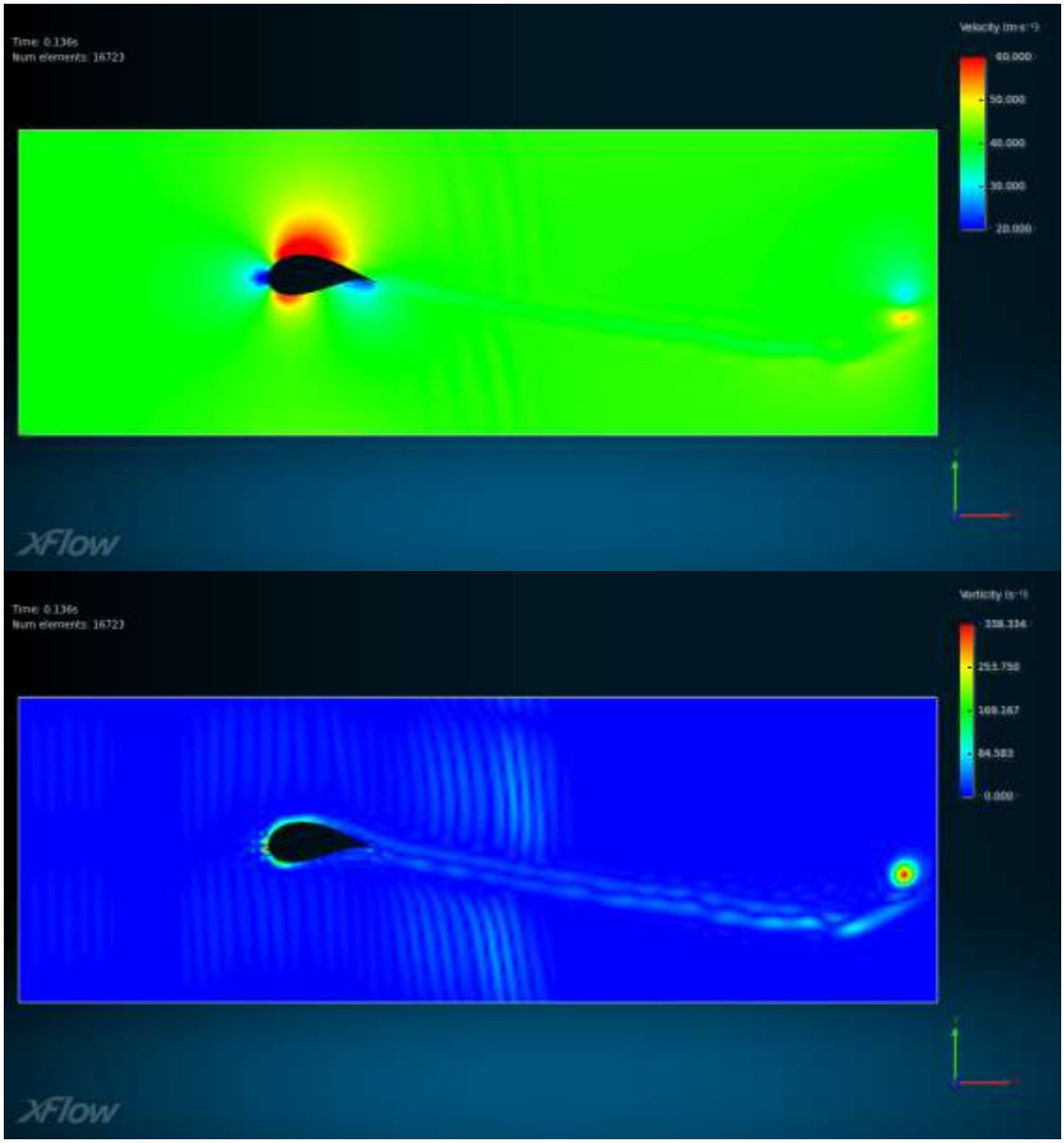


Figura 6.2.3.f. Forma de ala óptima de una de las optimizaciones. La imagen superior muestra la velocidad del aire y la imagen inferior muestra la vorticidad. La forma del ala no es simétrica y su curvatura es la que le permite generar *lift*. En la vorticidad se puede apreciar como la estela baja y como las bandas verticales están desincronizadas, signo de que el flujo de aire superior tiene más velocidad que el de la parte inferior.

7. Conclusiones y Trabajo Futuro

Se ha conseguido desarrollar un sistema que de manera no supervisada sea capaz de interactuar con BayesOpt y XFlow, permitiendo optimizar con BayesOpt simulaciones creadas mediante XFlow. Además, las mejoras introducidas sobre BayesOpt han asegurado la robustez del sistema ante posibles caídas.

Los experimentos de este trabajo han sido generados mediante la interfaz desarrollada, la cual ha cubierto las necesidades de funcionalidad y facilidad de uso requeridas para la generación y ejecución de los experimentos propuestos, asegurando así la validez de la interfaz en un entorno de trabajo ante problemas reales.

Podemos concluir que la optimización Bayesiana es un método de optimización agnóstico, en cuanto a que no necesita conocer información sobre el problema que esta optimizando, solo necesita conocer las muestras y sus resultados. Por tanto, el reto de aplicar optimización bayesiana a un ámbito distinto consiste en adaptar la entrada como parámetros e interpretar la salida como un resultado numérico.

Las simulaciones propuestas en este trabajo han sido creadas como concepto, ya que su objetivo principal es que permitan comprobar lo factible que es la optimización Bayesiana en simulaciones de fluidos. Por tanto, sería conveniente dotar este sistema de optimización a profesionales que requieran el uso de simuladores de fluidos en su entorno de trabajo con el fin de evaluar mejor las necesidades que puedan surgir de la interfaz desarrollada entre BayesOpt y XFlow, mejorando la interfaz de acuerdo a dichas necesidades.

Por otro lado, dado que las simulaciones presentadas en este trabajo han sido únicamente de un único material y mediante túnel de viento. Sería interesante experimentar con simulaciones diseñadas en otras condiciones mediantes XFlow, como por ejemplo: de transferencia de calor o con más de un material.

La idea de este trabajo ha girado en torno a reducir el número de muestras necesarias, de ahí el uso de la optimización Bayesiana. Pero otra forma habitual de reducir el tiempo total de ejecución es distribuir la carga de trabajo entre distintas mediante paralelización.

En optimización Bayesiana la paralelización sería la capacidad de poder seleccionar nuevas muestras de forma eficiente sin haber terminado de evaluar alguna muestra anterior. La solución no es trivial, pero por suerte existen formuladas distintas aproximaciones para dotar de paralelización a la optimización Bayesiana como GP-BUCB (24) o penalización local (25) entre otros. Sería interesante en el futuro implementar la paralelización a BayesOpt, adaptar la interfaz desarrollada para la ejecución de manera distribuida y repetir las simulaciones para formalizar en resultados el aumento en rendimiento.

8. Bibliografía

1. *BayesOpt: A Bayesian Optimization Library for Nonlinear Optimization, Experimental Design and Bandits.* **Martinez-Cantin, Ruben.** 2014, Journal of Machine Learning Research, págs. 3735-3739.

2. **NextLimit Technologies.** XFlow CFD. [En línea] [Citado el: 13 de Septiembre de 2015.] http://xflowcfd.com/.

3. **matplotlib.** matplotlib: a python plotting library. [En línea] [Citado el: 13 de Septiembre de 2015.] http://matplotlib.org/.

4. **Foundation, Blender.** Blender project - Free and Open 3D Creation Software. [En línea] [Citado el: 13 de Septiembre de 2015.] https://www.blender.org/.

5. **FreeCAD.** FreeCAD: An open-source parametric 3D CAD modeler. [En línea] [Citado el: 13 de Septiembre de 2015.] http://www.freecadweb.org/.

6. **Brochu, Eric, Cora, Vlad M. y De Freitas, Nando.** *A Tutorial on Bayesian Optimization of Expensive Cost Functions, with Application to Active User Modeling and Hierarchical Reinforcement Learning.* 2010.

7. **Shahriari, Bobak, y otros.** *Taking the Human Out of the Loop: A Review of Bayesian Optimization.* Universities of Harvard, Oxford, Toronto, and Google DeepMind. 2015.

8. Bayes Theorem. In Wikipedia. [En línea] [Citado el: 12 de Septiembre de 2014.] https://en.wikipedia.org/wiki/Bayes%27_theorem.

9. **Rasmussen, Carl Edward y Williams, Christopher K. I.** *Gaussian Processes for Machine Learning.* s.l. : the MIT Press, 2006.

10. Maximun Likelihood. In Wikipedia. [En línea] [Citado el: 23 de Septiembre de 2015.] https://en.wikipedia.org/wiki/Maximum_likelihood.

11. Markov Chain Monte Carlo. In Wikipedia. [En línea] [Citado el: 23 de Septiembre de 2015.] https://en.wikipedia.org/wiki/Markov_chain_Monte_Carlo.

12. *Practical Bayesian Optimization of Machine Learning Algorithms.* **Snoek, Jasper, Larochelle, Hugo y Adams, Ryan P.** 2012. NIPS.

13. Latin Hypercube Sampling. In Wikipedia. [En línea] [Citado el: 12 de Septiembre de 2015.] https://en.wikipedia.org/wiki/Latin_hypercube_sampling.

14. Courant–Friedrichs–Lewy condition. In Wikipedia. [En línea] [Citado el: 18 de Septiembre de 2015.] https://en.wikipedia.org/wiki/Courant%E2%80%93Friedrichs%E2%80%93Lewy_condition.

15. Student's t-distribution. In Wikipedia. [En línea] [Citado el: 12 de Septiembre de 2015.] https://en.wikipedia.org/wiki/Student%27s_t-distribution.

16. **Temporal Images.** Cessna Skyhawk II / 100 Performance Assessment. [En línea] [Citado el: 12 de Septiembre de 2015.] http://temporal.com.au/c172.pdf.

17. **Kamermans, Mike.** A Primer on Bézier Curves. [En línea] http://pomax.github.io/bezierinfo/.

18. STL File Format. In Wikipedia. [En línea] [Citado el: 12 de Septiembre de 2015.] https://en.wikipedia.org/wiki/STL_(file_format).

19. **AutoDesk.** AutoCAD 2010 User Documentation: Extrude Objects. [En línea]

20. ISO 10303-21, STEP-File. In Wikipedia. [En línea] [Citado el: 12 de Septiembre de 2015.] https://en.wikipedia.org/wiki/ISO_10303-21.

21. **STEP Tools.** STEP File Schema. [En línea] [Citado el: 12 de Septiembre de 2015.] http://www.steptools.com/support/stdev_docs/express/step_irs/html/index.html.

22. Constrained Optimization. In Wikipedia. [En línea] [Citado el: 23 de Septiembre de 2015.] https://en.wikipedia.org/wiki/Constrained_optimization.

23. Prior probability: Uninformative priors. In Wikipedia. [En línea] [Citado el: 18 de Septiembre de 2015.] https://en.wikipedia.org/wiki/Prior_probability#Uninformative_priors.

24. **Desautels, Thomas, Krause, Andreas y Burdick, Joel.** *Parallelizing Exploration–Exploitation Tradeoffs with Gaussian Process Bandit Optimization.* 2012.

25. **González, Javier, y otros.** *Batch Bayesian Optimization via Local Penalization.* 2015.

26. Maximum Likelihood. In Wikipedia. [En línea] [Citado el: 23 de Septiembre de 2015.] https://en.wikipedia.org/wiki/Maximum_likelihood.

Anexos

Anexo I. Manual de Usuario

Modificación de la Interfaz

En este manual se explicará paso a paso un fichero de interfaz que configura una optimización y el conjunto de funciones disponibles para facilitar la tarea al usuario. La interfaz se ha programado en Python, por tanto, es necesario que el usuario esté familiarizado con Python o algún otro lenguaje similar. Los conocimientos de Python necesarios dependerán de la complejidad del problema que se quiera diseñar.

Es necesario instalar xFlow y BayesOpt en el mismo equipo. Se recomienda al usuario a seguir los manuales de instalación de xFlow (http://xflowcfd.com/)y de BayesOpt (http://rmcantin.bitbucket.org/html/install.html).

Destacar que durante la instalación de BayesOpt hay que indicar a CMake que compile la API en Python. Asegurarse que está accesible el modulo bayesopt.so desde PYTHONPATH o sys.path para que funcione correctamente. Asegurarse también de tener instalado el paquete NumPy (http://www.numpy.org/) en Python.

Procederemos a desglosar las distintas partes de un fichero de una optimización diseñada. Corresponde a la optimización de la posición de un aerogenerador obstruido por otros 2 aerogeneradores. El fichero completo es el siguiente:

```
# Imports
from time import clock
import bayesopt
from bayesoptmodule import BayesOptContinuous
import numpy as np

from xflow_simulation import *

# XFlow Optimization Settings
xflow_dir = "/home/rmcantin/xflow_cfd/"
sim_dir = "/mnt/data/xflow/xflow_turbine/"
template_file = "TurbinesPosition_Template.xfp"
project_name = "TurbinesPosition"

# Output Settings
folder = os.path.basename(__file__).split(".")[0]
xflow_log = os.path.join(folder,"xflow.log")
numdata_folder = folder

xflow = XFlowSimulation(xflow_dir, sim_dir, template_file,
project_name, xflow_log, numdata_folder)

@xflow.input_modifier()
def modify_input(Xin):
  return [-30 + x * 60 for x in Xin]

@xflow.configure_log()
def testfunc(Xin):
    # Create .xfp
    xflow.create_from_template(Xin)
```

```
    # Put pre-optimized Courant number
    xflow.modify_xfp(courant_number=0.7)

    # Simulate with created .xfp
    xflow.simulate()

    # Get the required data from simulation
    wx_data = xflow.loadnumdata("blades", "Wx")

    # Get only the average of the last quarter of the data
    y = 0
    init = 3*len(wx_data)/4
    for x in xrange(init, len(wx_data)):
        y = y + wx_data[x]
    y = y / float(len(wx_data)-init)

    return y

# Bayes opt parameters
params = {}
params['n_iterations'] = 30
params['n_init_samples'] = 5
params['n_iter_relearn'] = 1
params['l_type'] = 'L_MCMC'
params['force_jump'] = 5
params['load_save_flag'] = 2
params['save_filename'] = os.path.join(folder,'turbines_position.dat')

n = 2
lb = np.zeros((n,))
ub = np.ones((n,))
start = clock()

mvalue, x_out, error = bayesopt.optimize(testfunc, n , lb, ub, params)

print "Result", x_out
print "Seconds", clock() - start
```

- Se empezará analizando en orden la parte de importación en Python:

```
# Imports
from time import clock
import bayesopt
from bayesoptmodule import BayesOptContinuous
import numpy as np

from xflow_simulation import *
```

· Se importa la función *clock*() del modulo *time*. Esta únicamente se utiliza para mostrar el tiempo total al finalizar la optimización, no es fundamental.

· Importamos el modulo *bayesopt* que nos permitira acceder a BayesOpt. Seguidamente, importar *BayesOptContinuous* o *BayesOptDiscrete* en función de si los parámetros de entrada a la optimización son continuos o discretos (en esta explicación solo se tendrá en cuenta el caso continuo)

·Importar *NumPy* e importar todas las funciones de *xflow_simulation.py*. Más adelante se detallarán las funciones de *xflow_simulation* disponibles.

● Detalles de la configuración de XFlow:

```
# XFlow Optimization Settings
xflow_dir = "/home/rmcantin/xflow_cfd/"
sim_dir = "/mnt/data/xflow/xflow_turbine/"
template_file = "TurbinesPosition_Template.xfp"
project_name = "TurbinesPosition"
```

Aquí se indica el path y nombre de cada uno de los componentes necesarios:

· *xflow_dir*: es el directorio de instalación de XFlow

· *sim_dir*: es el directorio donde se encuentra la simulación de XFlow

· *template_file*: en caso de usarse el sistema de plantilla con etiquetas para modificar el XML (se detallará más adelante), se indica el fichero que actuará como plantilla. Debe estar situado dentro del directorio *sim_dir*.

· *project_name*: nombre del proyecto de la simulación de XFlow. Todos los ficheros de dicho proyecto tienen que estar dentro del *sim_dir*.

● Configuración de directorio destino e instancia de la clase *XFlowSimulation*:

```
# Output Settings
folder = os.path.basename(__file__).split(".")[0]
xflow_log = os.path.join(folder,"xflow.log")
numdata_folder = folder

xflow = XFlowSimulation(xflow_dir, sim_dir, template_file,
project_name, xflow_log, numdata_folder)
```

· *folder*: directorio destino en el que se guardarán los resultados y logs. Tal y como aparece, se generá una carpeta con el mismo nombre que el fichero Python. Se puede modificar si es necesario.

·*xflow_log*: los logs que genera XFlow durante cada una de las simulaciones pueden ser guardados si se especifica un fichero. Tal y como aparece, concatena *folder* y "xflow.log" como fichero de log. Se puede modificar si es necesario.

· *numdata_folder*: puede ser conveniente almacenar los ficheros binarios que almacenan los resultados de cada simulación en un directorio para un futuro análisis (se sobrescriben en cada simulación). Se puede modificar si es necesario.

La clase *XFlowSimulation* permite simplificar el acceso a las funciones de xflow_simulation.py, evitando repetir los argumentos en cada llamada. Además permite su uso permite el uso de logs. Siempre que sea posible se aconseja utilizar *XFlowSimulation* en lugar de llamar directamente a las funciones.

● Modificador de parámetros de entrada:

```
@xflow.input_modifier()
def modify_input(Xin):
  return [-30 + x * 60 for x in Xin]
```

Existen dos alternativas para modificar el rango de valores de los parámetros de entradas: definir una función que los modifique o definir los intervalos *lb* y *lu* en la ejecución de BayesOpt (se explica más adelante).

Utilizando el decorador *@xflow.input_modifier()* sobre una función permite indicar que la función actuará como modificador de los parámetros de entrada. En este caso, la función definida *modify_input(Xin)* recibirá los parámetros de BayesOpt (por defecto están entre 0 y 1) y los modifica al intervalo (-30, 30). Esta función debe devolver el mismo número de parámetros que entran.

- Función coste a optimizar:

```
@xflow.configure_log()
def testfunc(Xin):
    # Create .xfp
    xflow.create_from_template(Xin)

    # Put pre-optimized Courant number
    xflow.modify_xfp(courant_number=0.7)

    # Simulate with created .xfp
    xflow.simulate()

    # Get the required data from simulation
    wx_data = xflow.loadnumdata("blades", "Wx")

    # Get only the average of the last quarter of the data
    y = 0
    init = 3*len(wx_data)/4
    for x in xrange(init, len(wx_data)):
        y = y + wx_data[x]
    y = y / float(len(wx_data)-init)

    return y
```

La función coste se define aquí y más tarde deberá ser indicada como objetivo de la optimización en la llamada a *optimize()*.

De nuevo utilizamos un decorador *@xflow.configure_log* que permite a la clase *XFlowSimulation* identificar cada vez que se la función es ejecutada. De esta forma puede generar ficheros logs y también le permite identificar cuando termina la función para proceder a copiar los ficheros binarios de resultados a la carpeta *numdata_folder*.

Esta función debe recibirá el conjunto de parámetros de entrada. Como se ha utilizado el decorador *@xflow.input_modifier()*, los parámetros ya vendrán modificados de acuerdo a la función.

El contenido de esta función dependerá de lo que se necesite modificar de la simulación, las simulaciones a ejecutar y de los resultados a tener en cuenta.

En esta simulación especificada por el fichero tenemos las siguientes operaciones:

· *xflow.create_from_template(Xin)*: generar a partir de una plantilla etiquetada el XML de configuración de la simulación XFlow de acuerdo a los parámetros de entrada *Xin.* El fichero XML generado sustituirá al ya existente.

·xflow.modify_xfp(courant_number=0.7): facilita la modificación del XML de configuración de la simulación XFlow. De momento solo contempla la modificación del número de Courant (*courant_number)* y del porcentaje de la simulación que se quiere simular (*sim_percent)*. En este caso se está modificando el numero de Courant a 0.7. Importante realizar este tipo de operaciones siempre después de *xflow.create_from_template(Xin)*, puesto que dicha operación sobrescribe completamente el fichero XML.

·xflow.simulate(): ejecutar el proceso de simulación de XFlow. Esta llamada bloqueará hasta la finalización de la simulación.

·xflow.loadnumdata("blades", "Wx"): tomar del componente "blades" los valores de "Wx" de los resultados de la simulación.

La parte restante de la función corresponde sintetizar los resultados de la optimización en un único valor numérico para ser devuelto como resultado de la función. En este caso se realiza la media de los valores de *Wx,* pero solo del último cuarto del total.

- Parámetros de BayesOpt:

```
# Bayes opt parameters
params = {}
params['n_iterations'] = 30
params['n_init_samples'] = 5
params['n_iter_relearn'] = 1
params['l_type'] = 'L_MCMC'
params['force_jump'] = 5
params['load_save_flag'] = 2
params['save_filename'] = os.path.join(folder,'turbines_position.dat')
```

Aquí se especifican los parámetros de configuración de BayesOpt. Se recomienda consultar la documentación de BayesOpt disponer de una lista completa de los parámetros y su significado. No obstante, se explicarán los siguientes:

· *n_iterations*: permite indicar el número de iteraciones de optimización. Según la simulación y el número de parámetros de entrada, habrá que modificar el valor para dar suficientes iteraciones para alcanzar el óptimo.

· *n_init_samples*: número de muestras iniciales. Se recomienda como mínimo 2 veces el número de parámetros de entrada.

· *save_filename*: fichero en el que se guardará el estado de la optimización. Una vez finalizada servirá para analizar los resultados de la optimización. En este caso concatenamos el directorio *folder* junto a "*turbines_position.dat*". Se puede modificar según sea necesario.

Para el resto de parámetros se recomienda no modificarlos o consultar la documentación de BayesOpt referente a los parámetros. Aquellos parámetros no especificados mantienen su valor por defecto.

● Optimización mediante BayesOpt:

```
n = 2
lb = np.zeros((n,))
ub = np.ones((n,))
start = clock()

mvalue, x_out, error = bayesopt.optimize(testfunc, n , lb, ub, params)

print "Result", x_out
print "Seconds", clock() - start
```

· *n:* es el número de parámetros de entrada a utilziar. En este caso estamos modificando la posición (x,y) del aerogenerador, por tanto, 2 parámetros.

· *lb* y *ub:* cota inferior y superior de cada parámetro. Puede modificarse para alterar los intervalos. Si se utiliza *@xflow.input_modifier()* se recomienda dejar los valores tal y como aparecen, es decir, intervalo entre 0 y 1.

·*bayesopt.optimize(testfunc, n, lb, ub, params):* esta es la función que inicia la optimización. Cualquier configuración debe efectuarse antes de esta llamada. Se le indica la función a optimizar (en este caso es *testfunc*), el número de parámetros de entrada *n*, las cotas inferior y superior de cada parámetro *lb* y *yb* y el conjunto de parámetros *params*. Esta llamada nos devuelve el óptimo y, en caso de error, el error ocurrido.

● Otras consideraciones a tener en cuenta:

Si ocurre algún error de Python una vez iniciada la ejecución de la optimización, la traza de error de Python no aparecerá, sino que aparecerá un error genérico. Se recomienda modificar la interfaz y ejecutar la función de coste antes de la llamada a *optimize()*, por ejemplo:

```
import sys
testfunc([0.5,0.5])
sys.exit()
```

Estas líneas ejecutarán la función de coste directamente y seguidamente terminará mediante *sys.exit().* Debido a que ejecutamos la función de coste directamente en Python en lugar de a través de BayesOpt, si se produce un error al ejecutar la función de coste, la traza aparecerá correctamente. Una vez identificado y corregido el error retirar estas líneas de código para ejecutar la optimización de manera normal.

Funciones disponibles para la Interfaz

Aquí se enumerarán las distintas funciones disponibles a través del script xflow_simulation.py. Estas funciones facilitan la modificación de la interfaz para la simulación a optimizar.

Si se está utilizando la clase XFlowSimulation, cuenta con las mismas funciones, pero como almacena los argumentos durante la creación de la instancia XFlowSimulation, las funciones prescinden de dichos argumentos. A continuación mostramos las funciones con todos los argumentos:

- función: *loadnumdata(directory,entity_name,component_name, xml_file="numericaldata.xml", bin_file="numericaldata.bin")*

Esta función permite leer los ficheros binarios generados como resultado de las simulaciones. Hay que indicarle el directorio donde se encuentran los binarios (directory).

Las entidades (entity_name) y componentes (componente_name) disponibles son los mismos que aparecen en el *Function Viewer* del programa XFlow en su ejecución con GUI.

Se puede cambiar el nombre del fichero binario (bin_file) y del fichero que guarda el índice de datos (xml_file) por si sus nombre no fuese el de por defecto.

Devuelve un vector de elementos con los valores en cada instante de la simulación. Si se índica varías entidad-componente, devuelve el vector de vectores.

- función: *execute(command, exec_dir=None,log_filename=None)*

Permite ejecutar un comando con facilidad. A pesar de que ha sido creado para ser usado por otras funciones, se puede usar desde la modificación de la interfaz si fuese necesario ejecutar algún proceso.

La manera de ejecutar un comando (command) es similar a utilizar un proceso desde el CLI. También se puede cambiar el directorio de ejecución (exec_dir) (conocido como CWD , current working directory), por si fuera necesario la ejecución como si se ejecutase desde cierto directorio.

Si se indica un fichero log (log_filename) se redireccionará la salida a dicho fichero.

- función: *get_simulation_folder(xfp_file)*

Permite recuperar con facilidad la carpeta en la que se guardarán los datos de la simulación a partir del fichero XML de la configuración de la simulación (xfp_file).

- función: *modify_xfp(xfp_file, courant_number=None, sim_percent=None)*

Modifica con facilidad campos comunes de los XML de configuración de XFlow (xfp_file) como el número de Courant (courant_number) o el porcentaje de la simulación a ejecutar (sim_percent).

- función: *simulate(sim_project, xflow_dir, log_filename=None)*

Ejecuta la simulación de un proyecto (sim_project), indicandole su directorio (xflow_dir). Permite redireccionar la salida de XFlow hacia un fichero (log_filename).

● función: *create_from_template(xin, template_file, output_file , tag="XXXX_")*

Permite modificar el XML de configuración de XFlow mediante etiquetas. Se sustituyen las etiquetas con un prefijo (tag) seguido de un número (por ejemplo: "XXX_1") por el valor de entrada (xin) que ocupa la posición del número. Por tanto, necesita un fichero etiquetado (template_file) y el fichero que generará (output_file).

● decorador*: input_modifier(self)*

Solo disponible mediante la clase XFlowSimulation. Requiere el uso del decorador *configure_log* para funcionar.

Utilizando este decorador sobre una función, permite indicar que dicha función se ocupará de modificar los valores de entrada, los Xin del punto a evaluar.

La función decorada deberá, por tanto, aceptar los datos de entrada Xin (un vector de valores) y devolverá otro de misma dimensión con los datos modificados.

● decorador: *configure_log(self)*

Solo mediante la clase XFlowSimulation.

Permite indicar qué función es la que está siendo optimizada (la que se utiliza en la función de *bayesopt.optimize()*) para que estructure mejor el fichero de logging(creando separadores en el fichero de *logging* entre cada simulación) y automatiza la copia de los ficheros binarios generados como resultado de cada simulación, ya que se perderían al ser sobrescritos en cada simulación ejecutada.