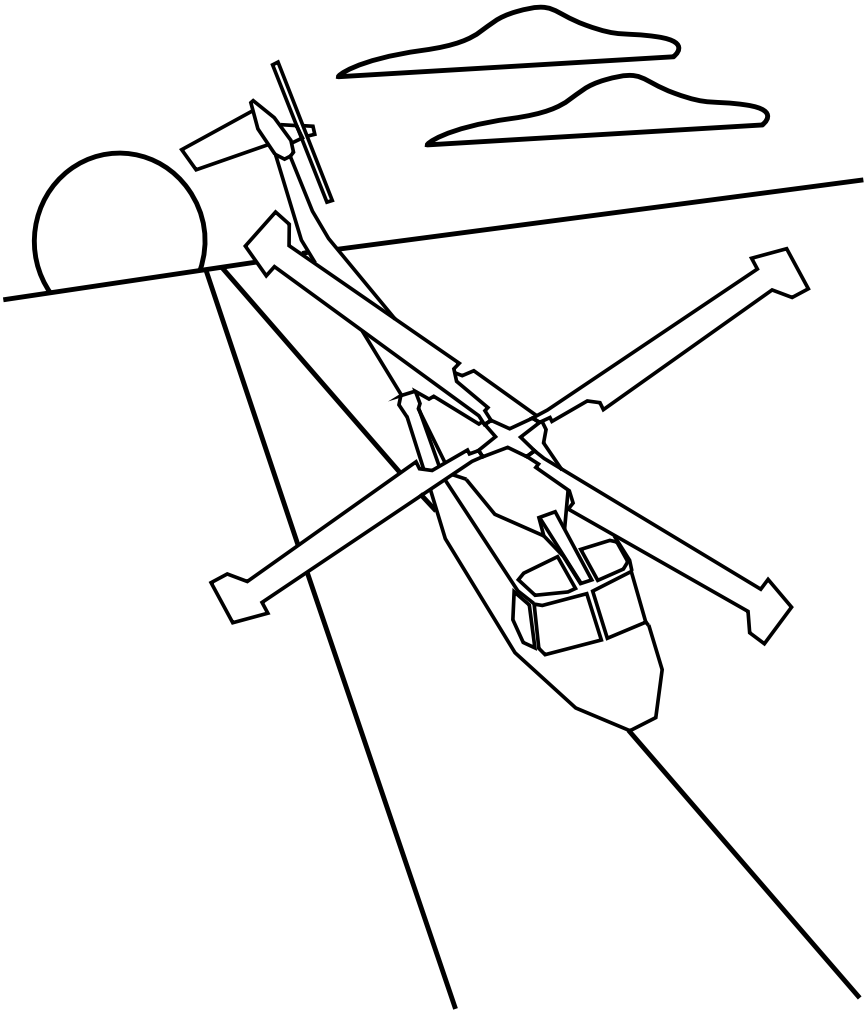
Simulador de Helicóptero



Pedro López-Adeva Fernández-Layos
Proyecto Fin de Carrera
Ingeniería Aeronáutica
Universidad Politécnica de Madrid

Dedicado a mis amigos Álvaro, Inés, Alex y Juan por aguantarme todo este tiempo e intentar mantenerme cuerdo. Sé que ha tenido mérito.
A mi novia Maica porque me ha apoyado y motivado constantemente y hace que me sea más fácil cualquier dificultad.
Sobre todo dedicado a mi madre, porque cualquier mérito mío es suyo, y a mi hermano, de quien he aprendido las lecciones más importantes.

Índice general

Capítulo 1

Introducción

Se ha desarrollado un simulador de procedimientos para helicópteros de cuatro palas, un solo rotor principal y un rotor de cola. El simulador utiliza un modelo de 6 grados de libertad de sólido rígido para el fuselaje. Para el rotor calcula la solución del grado de libertad de batimiento de cada pala y 3 grados de libertad para la velocidad inducida. Por último, incluye un grado de libertad para la velocidad de giro del rotor y el par del motor. Dados los cuatro controles que ejerce el piloto y los datos atmosféricos puede calcular la respuesta del helicóptero en cualquier situación de vuelo.

Los parámetros del helicóptero se especifican en un fichero aparte. Se incluyen dos ficheros de ejemplo para el Lynx y el BlackHawk.

Por otra parte el simulador tiene las siguientes limitaciones:

- No simula correctamente los límites de operación del helicóptero. No predice la entrada en pérdida del rotor.
- No simula correctamente maniobras bruscas.
- Sistema de control demasiado simple. Sistemas reales llenos de no linealidades y mucho mas complejos.
- No tiene en cuenta efectos de interacción entre rotor, rotor de cola, estabilizadores y fuselaje.
- No calcula el consumo de combustible y la variación de la masa y los momentos de inercia debido a ello.

Adicionalmente se incluye código, que se puede utilizar desde otros programas, o desde la consola interactiva de python para el cálculo no interactivo de la respuesta, el cálculo de derivadas de estabilidad y el cálculo del trimado del helicóptero para diversas condiciones de vuelo. Éste último con la limitación de que no converge bien para las situaciones con resbalamiento.

Capítulo 2

Sistemas de referencia

A lo largo de todo este documento, utilizaremos los siguientes sistemas de referencia:

- Ejes geocéntricos
- Ejes inerciales de referencia
- Ejes locales de referencia
- Ejes cuerpo
- Ejes rotor
- Ejes rotor-viento
- Ejes rotor de cola
- Ejes rotor de cola-viento
- Ejes pala
- Ejes rotación

Para dejar claro de en qué sistema de referencia estamos expresando las componentes de una magnitud vectorial, añadiremos siempre un subíndice, que indicamos más adelante en la explicación en detalle de cada sistema de referencia. Como excepciones a esta regla, si no se indica subíndice para la velocidad y la velocidad angular se entiende que hablamos de ejes cuerpo.

2.1. Ejes geocéntricos

Ejes con origen en el centro de la tierra, el plano xy coincidente con el plano ecuatorial, con el eje x dirigido hacia el meridiano de Greenwich y el eje z hacia el polo norte (ver 2.1). Suponemos que estos ejes son inerciales despreciando el movimiento de la tierra en torno al sol y el movimiento de rotación sobre su propio eje. La posición del helicóptero la expresamos en coordenadas cartesianas, utilizando el subíndice “G”. Debido a que los datos geográficos se encuentran

en coordenadas geodésicas, y el subsistema visual de la simulación exige expresar la posición en dichas coordenadas, será necesario transformar la posición a coordenadas geodésicas de longitud λ, latitud ϕ y altitud h. Información sobre este sistema se encuentra disponible en numerosas referencias, pero aquí por comodidad se hace un breve resumen (ver 183-203 de [10]):

2.1.1. Coordenadas geodésicas

Si modelamos la superficie de la tierra mediante un elipsoide de revolución, podemos especificar la posición de cualquier punto respecto a la tierra mediante el siguiente conjunto de coordenadas (ver figuras 2.1 y 2.2):

- altitud h: la distancia desde el punto a su proyección sobre la superficie del elipsoide.
- longitud λ: el ángulo que forma el meridiano donde se encuentra la proyección del punto con el meridiano de referencia (Greenwich). La longitud geodésica es equivalente a la longitud geocéntrica.
- latitud ϕ: el ángulo que forma la vertical local (la normal a la superficie del elipsoide) con el plano ecuatorial.

El paso de coordenadas geodésicas a coordenadas cartesianas geocéntricas es muy sencillo:

$$\begin{aligned} x_G &= (N+h)\cos\phi\cos\lambda \\ y_G &= (N+h)\cos\phi\sin\lambda \\ z_G &= \left[N(1-e^2)+h\right]\sin\phi \end{aligned}$$

N es un radio de curvatura y vale:

$$N = \frac{a}{\sqrt{1-f(2-f)\sin^2\phi}}$$

Donde los parámetros que aparecen en las anteriores ecuaciones definen la forma del elipsoide:

$$\begin{aligned} e &= \sqrt{\frac{a^2-b^2}{a^2}} \\ f &= \frac{a-b}{a} \end{aligned}$$

e es la excentricidad, f el achatamiento y a y b los semiejes mayor y menor del elipsoide respectivamente (que se encuentra revolucionado según el semieje menor, es decir, a es también el radio ecuatorial). Como a partir de sólo dos parámetros el elipsoide queda definido normalmente se suelen dar como datos el achatamiento y el radio ecuatorial.

Debido a que la superficie de la tierra no es obviamente un elipsoide se pueden especificar numerosos elipsoides, ajustando los parámetros de modo que mediante mínimos cuadrados se minimice la distancia entre la superficie de

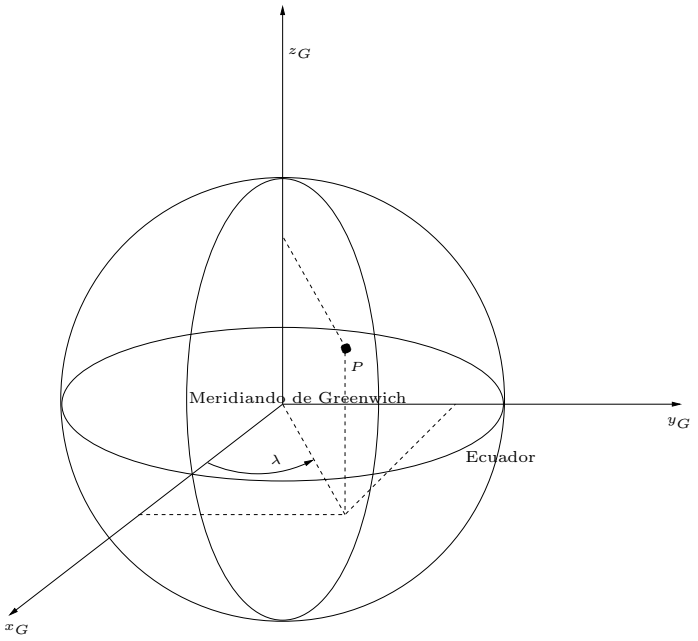


Figura 2.1: Ejes geocéntricos y longitud geodésica

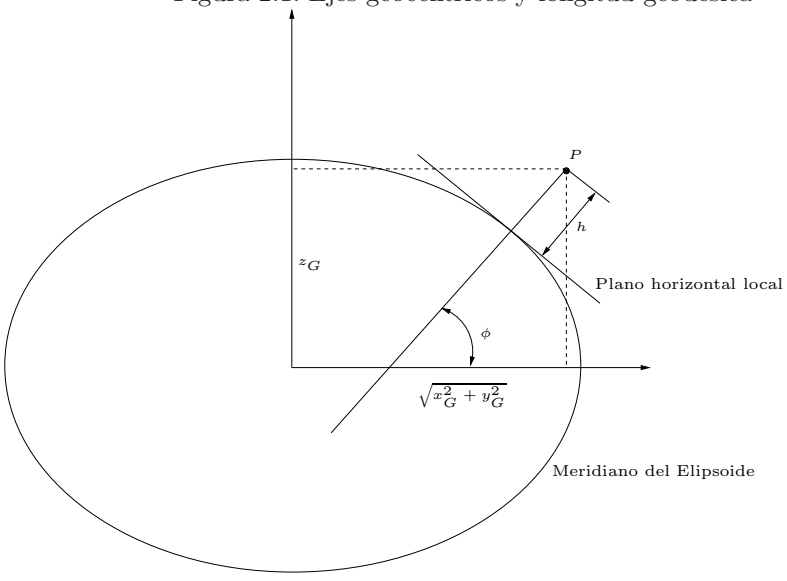


Figura 2.2: Latitud y altitud geodésicas

la tierra y el elipsoide. Si la minimización se aplica a una región de la tierra obtenemos un elipsoide local y si se aplica a toda la superficie de la tierra un elipsoide global. En concreto el elipsoide que utilizaremos es el WGS84 (World Geodetic System 1984), un elipsoide global, cuyo centro coincide con el centro de la tierra y con los siguientes parámetros:

$$a = 6378137m$$
$$\frac{1}{f} = 298{,}257223563$$

La definición exacta y otras propiedades se pueden consultar en [11].

El paso de coordenadas cartesianas a geodésicas es más complicado y lo resolvemos numéricamente:

$$\tan\lambda = \frac{y_G}{x_G}$$
$$\tan\phi_2 = \frac{1}{\sqrt{x_G^2+y_G^2}}\left(z_G + e^2 a \frac{\tan\phi_1}{\sqrt{1+(\tan\phi_1)^2(f-1)^2}}\right)$$
$$h = \frac{\sqrt{x_G^2+y_G^2}}{\cos\phi} - N$$

Donde la segunda fórmula es recursiva y da el siguiente valor estimado de latitud a partir del anterior, denotados respectivamente con los subíndices 2 y 1. Como valor inicial de latitud en la iteración utilizaremos obviamente la latitud en el instante anterior del helicóptero, pero para el instante inicial necesitamos otra estimación, por ejemplo, la latitud para altitud cero:

$$\tan\phi_1 = \frac{1}{1-e^2}\frac{z_G}{\sqrt{x_G^2+y_G^2}}$$

Al aproximarnos a los polos la anterior fórmula deja de ser válida, por lo que resolvemos la inversa de la tangente, y cambiamos la ecuación para la altura:

$$h = \frac{z_G}{\sin\phi} - N(1-e^2)$$

2.2. Ejes inerciales de referencia

Ejes con el plano xy coincidente con el suelo, de forma precisa: el plano paralelo al geoide, con el eje x apuntando hacia el sur (tangente al meridiano local), el eje y apuntando hacia el este (tangente al paralelo local) y el eje z según la vertical local. El origen de los ejes inerciales de referencia es arbitrario, pudiendo hacerlo coincidir, por ejemplo, con la posición inicial del helicóptero. Por comodidad se pueden cambiar dichos ejes más adelante transformando adecuadamente la orientación de los ejes cuerpo y el vector de posición del helicóptero. Denotamos a estos ejes mediante el subíndice "I".

La matriz de cambio de coordenadas entre los ejes inerciales y los ejes geocéntricos depende de la latitud y longitud del origen de los ejes inerciales:

$$L_{IG} = \begin{bmatrix} \sin\phi\cos\lambda & \sin\phi\sin\lambda & -\cos\phi \\ -\sin\lambda & \cos\lambda & 0 \\ \cos\phi\cos\lambda & \cos\phi\sin\lambda & \sin\phi \end{bmatrix}$$

2.2.1. Ejes locales de referencia

Definidos de manera similar a los ejes inerciales de referencia pero su origen se desplaza al moverse el helicóptero, de forma que coincide con la proyección del centro del helicóptero sobre la superficie del elipsoide El plano xy es de nuevo tangente al elipsoide en el punto proyectado, con el eje x hacia el sur, el eje y hacia el este y el eje z según la vertical local. Identificamos a estos ejes con el subíndice "L" y son útiles ya que la orientación del helicóptero viene especificada mediante los ángulos de Euler respecto a estos ejes.

La matriz de cambio L_{LG} es idéntica a L_{IG} con la única diferencia que el origen de los ejes cambia con el tiempo.

Para aclarar conceptos: con la posición inicial del helicóptero de latitud, longitud y altura calculamos los ejes inerciales y la posición inicial en coordenadas cartesianas geocéntricas. Respecto a estos ejes se da el desplazamiento del helicóptero x_I, y_I, z_I y se tiene su orientación con el cuaternio q_0, q_1, q_2, q_3. Se utilizan estos ejes en vez de los geocéntricos cartesianos, que también son inerciales, porque es más intuitivo imaginar el desplazamiento y orientación del helicóptero respecto a la posición inicial y respecto a la horizontal local inicial que respecto al centro de la tierra y el plano ecuatorial, aunque físicamente sea indiferente. En cada instante de tiempo, como hemos calculado la posición inicial en coordenadas cartesianas geocéntricas y tenemos el desplazamiento calculamos las coordenadas x_G, y_G, z_G del helicóptero y pasamos estas coordenadas a geodésicas λ, ϕ, h porque el subsistema visual nos pide indicarle la posición del helicóptero en dichas coordenadas. Además nos pide indicar la orientación mediante ángulos de Euler medidos respecto a la horizontal local, por lo que con las coordenadas geodésicas calculamos los ejes locales de referencia, y respecto a ellos calculamos los ángulos de Euler.

2.3. Ejes cuerpo

Ejes ligados al helicóptero, concretamente al fuselaje (que suponemos rígido), con origen en el centro de masas del helicóptero y el plano xz contenido en el plano de simetría del helicóptero, el eje z dirigido "hacia abajo", el eje x dirigido hacia el morro del helicóptero y el eje y perpendicular al plano xz, formando un triedro a derechas y de forma que el plano xy es paralelo al plano del rotor. En realidad los ejes cuerpo son también hasta cierto punto arbitrarios ya que el helicóptero no es totalmente simétrico, ni el plano yz es totalmente paralelo al plano del rotor. Denotamos estos ejes mediante el subíndice "B"

2.3.1. Cambio de ejes cuerpo a ejes inerciales de referencia

Guardamos la orientación del helicóptero respecto a los ejes inerciales de referencia añadiendo al vector de estado las magnitudes q_0, q_1, q_2 y q_3, que representan las cuatro componentes del cuaternio unidad q con parte escalar q_0 y parte vectorial (q_1, q_2, q_3), que denotamos de la forma $q = [q_0, (q_1, q_2, q_3)]$, tal que dado un vector $\vec{r}$ cuyas componentes en ejes inerciales son (x_I, y_I, z_I) (ver [2]) . Entonces las componentes en ejes inerciales del vector rotado son la parte vectorial del cuaternio:

$$[0, (x_B, y_B, z_B)] = q\,[0, (x_I, y_I, z_I)]\,\bar{q}$$

Donde $\bar{q}$ representa el conjugado del cuaternio q. Es decir, q rota de los ejes inerciales a los ejes cuerpo. Aplicando la definición de multiplicación de cuaternios, podemos obtener la matriz de rotación de ejes inerciales a ejes cuerpo R_{BI} y la matriz de cambio de base de ejes inerciales a cuerpo L_{IB} es simplemente la transpuesta de la matriz de rotación:

$$L_{IB} = R_{BI} = \begin{bmatrix} 1-2(q_2^2+q_3^2) & 2(-q_0q_3+q_1q_2) & 2(q_0q_2+q_1q_3) \\ 2(q_0q_3+q_1q_2) & 1-2(q_1^2+q_3^2) & 2(-q_0q_1+q_2q_3) \\ 2(-q_0q_2+q_1q_3) & 2(q_0q_1+q_2q_3) & 1-2(q_1^2+q_2^2) \end{bmatrix} \quad (2.1)$$

Si la velocidad angular en ejes inerciales viene dada por $\vec{\omega}_I$ entonces en un instante diferencial de tiempo provoca una rotación dada por el cuaternio:

$$\delta q = \left[\cos\left(\frac{|\vec{\omega}_I|dt}{2}\right), \sin\left(\frac{|\vec{\omega}_I|dt}{2}\right)\frac{\vec{\omega}_I}{|\vec{\omega}_I|}\right]$$

El cuaternio de rotación de ejes inerciales a cuerpo será en un instante de tiempo posterior $q(t+dt) = \delta q q(t)$, por lo que podemos expresar la derivada del cuaternio en función de la velocidad angular:

$$\dot{q} = \lim_{dt\to 0}\frac{\delta q - [1,(0,0,0)]}{dt}q = \frac{1}{2}\left[0,\vec{\omega}_I\right]q$$

La anterior ecuación dada por componentes es:

$$\dot{q}_0 = -\frac{1}{2}(p_I q_1 + q_I q_2 + r_I q_3) \quad (2.2a)$$

$$\dot{q}_1 = \frac{1}{2}(p_I q_0 + q_I q_3 - r_I q_2) \quad (2.2b)$$

$$\dot{q}_2 = \frac{1}{2}(-p_I q_3 + q_I q_0 + r_I q_1) \quad (2.2c)$$

$$\dot{q}_3 = \frac{1}{2}(p_I q_2 - q_I q_1 + r_I q_0) \quad (2.2d)$$

Añadiremos estas cuatro ecuaciones cinemáticas a la ecuación diferencial para el vector de estado. Hay que tener cuidado de fijarse que el vector velocidad angular está dado en las anteriores relaciones en coordenadas inerciales, por lo que habrá que transformar primero a estas coordenadas ya que en el vector de estado la velocidad angular se encuentra en ejes cuerpo:

$$\begin{Bmatrix} p \\ q \\ r \end{Bmatrix}_I = L_{IB}\begin{Bmatrix} p \\ q \\ r \end{Bmatrix}_B \quad (2.3)$$

2.3.2. Ángulos de Euler

A pesar de que el modelo no los utiliza en absoluto el subsistema visual espera que para la orientación se le suministren los 3 ángulos de Euler, por lo que es necesario poder obtenerlos en función del cuaternio de orientación. Surge el problema de qué pasa para $\theta = \frac{\pi}{2}$ en cuyo caso hay dudas para los valores de ψ y ϕ. Si escribimos la matriz de cambio de base en función de los ángulos de Euler:

$$L_{BI} = \begin{bmatrix} \cos\theta\cos\psi & \cos\theta\sin\psi & -\sin\theta \\ -\cos\phi\sin\psi + \sin\phi\sin\theta\cos\psi & \cos\phi\cos\psi + \sin\phi\sin\theta\sin\psi & \sin\phi\cos\theta \\ \sin\phi\sin\psi + \cos\phi\sin\theta\cos\psi & -\sin\phi\cos\psi + \cos\phi\sin\theta\sin\psi & \cos\phi\cos\theta \end{bmatrix}$$

Podemos comparar con la matriz de cambio de base en función del cuaternio de rotación y obtener para $\theta \neq \pm\frac{\pi}{2}$:

$$\begin{aligned} \sin\theta &= -2(-q_0q_2 + q_1q_3) \\ \tan\psi &= \frac{2(q_1q_2 + q_0q_3)}{1 - 2(q_2^2 + q_3^2)} \\ \tan\phi &= \frac{2(q_0q_1 + q_2q_3)}{1 - 2(q_1^2 + q_2^2)} \end{aligned}$$

Si es $\theta = \pm\frac{\pi}{2}$ entonces:

$$\begin{aligned} \psi &= 0 \\ \tan\phi &= \frac{q_1q_2 - q_0q_3}{q_0q_2 + q_1q_3} \end{aligned}$$

Hacer $\psi = 0$ es totalmente arbitrario, ya que existen una infinidad de valores de ψ y ϕ que para $\theta = \pm\frac{\pi}{2}$ dan la orientación correcta de los ejes cuerpo.

El cuaternio de rotación está dado sin embargo respecto al sistema inercial que hemos definido con anterioridad, mientras que los ángulos de Euler se obtienen calculando las rotaciones respecto a un sistema de referencia similar al inercial local sólo que con el eje z y el eje x en sentidos contrarios, por lo que corregimos los valores anteriores:

$$\begin{aligned} \theta_2 &= -\theta_1 \\ \psi_2 &= \pi - \psi_1 \\ \phi_2 &= \pi + \phi_1 \end{aligned}$$

2.4. Ejes rotor

Ejes con el origen el centro de rotación de las palas, y girados ligeramente por el ángulo $-\gamma_s$, alrededor del eje y_B respecto a los ejes cuerpo. Denotamos estos ejes con el subíndice "h" (ver figura 2.3)

La matriz de cambio de base de ejes cuerpo a ejes rotor es:

$$L_{hB} = \begin{bmatrix} \cos\gamma_s & 0 & \sin\gamma_s \\ 0 & 1 & 0 \\ -\sin\gamma_s & 0 & \cos\gamma_s \end{bmatrix} \tag{2.4}$$

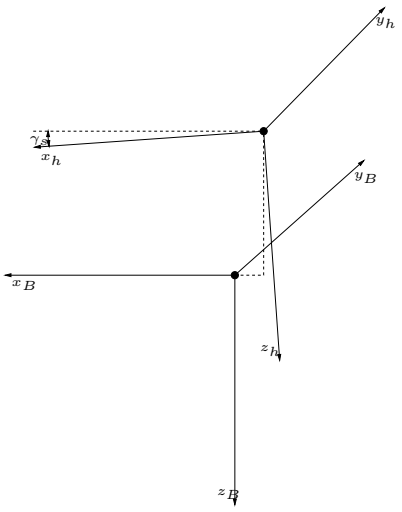


Figura 2.3: Ejes rotor

2.5. Ejes rotor-viento

Ejes con el mismo origen que los ejes rotor, pero rotados alrededor del eje z_h un ángulo ψ_w (ver figura 2.5), que se determina según la incidencia del aire sobre el rotor, y que por lo tanto cambia con el movimiento del helicóptero:

$$\cos(\psi_w) = \frac{u_{hrwh}}{\sqrt{u_{hrwh}^2 + v_{hrwh}^2}}$$
$$\sin(\psi_w) = \frac{v_{hrwh}}{\sqrt{u_{hrwh}^2 + v_{hrwh}^2}}$$

Donde u_{hrwh}, v_{hrwh}, w_{hrwh} representan las componentes de la velocidad del origen del rotor, en ejes rotor. La calculamos a partir de la velocidad relativa al viento del helicóptero, la velocidad angular y los parámetros geométricos de la posición del rotor:

$$\left\{ \begin{array}{c} u_{hrw} \\ v_{hrw} \\ w_{hrw} \end{array} \right\}_h = L_{hB} \left\{ \begin{array}{c} u_{hrw} \\ v_{hrw} \\ w_{hrw} \end{array} \right\}_B$$

L_{hB} ya se ha calculado (ver ecuación 2.4)

$$\left\{ \begin{array}{c} u_{hrw} \\ v_{hrw} \\ w_{hrw} \end{array} \right\}_B = \left\{ \begin{array}{c} u_{rw} \\ v_{rw} \\ w_{rw} \end{array} \right\}_B + \vec{\omega}_B \wedge \left\{ \begin{array}{c} -x_{cg} \\ 0 \\ -h_R \end{array} \right\}_B$$

La velocidad angular de los ejes cuerpo en ejes cuerpo:

$$\vec{\omega_B} = \left\{ \begin{array}{c} p \\ q \\ r \end{array} \right\}$$

Si es $u_{hrwh} = v_{hrwh} = 0$ entonces $\psi_w = 0$ Lo subíndices "rw" indican "relativo al viento", y u, v, w siempre denotan las componentes x, y, z de la velocidad. Denotamos estos ejes con el subíndice "w"

2.5.1. Magnitudes derivadas en ejes viento

Es necesario calcular más adelante el valor de la derivada de las dos componentes de la velocidad angular de los ejes viento:

$$p_w = p_h \cos\psi_w + q_h \sin\psi_w \tag{2.5}$$
$$q_w = -p_h \sin\psi_w + q_h \cos\psi_w \tag{2.6}$$

Derivando respecto al viento:

$$\dot{p}_w = \dot{p}_h \cos\psi_w + \dot{q}_h \sin\psi_w + \dot{\psi}_w \left(-p_h \sin\psi_w + q_h \cos\psi_w\right)$$
$$\dot{q}_w = -\dot{p}_h \sin\psi_w + \dot{q}_h \cos\psi_w + \dot{\psi}_w \left(-p_h \cos\psi_w - q_h \sin\psi_w\right)$$

El problema está en que $\dot{\psi_w}$ no se encuentra definido en determinadas condiciones ya que ψ_w puede experimentar saltos finitos en un instante de tiempo:

$$\dot{\psi_w} = \frac{-v\dot{u}_{rwh} + u\dot{v}_{rwh}}{u_{rwh}^2 + v_{rwh}^2}$$

Esto es debido a que los ejes viento cambian bruscamente por lo que no podemos en realidad utilizar la clásica ecuación de derivación en ejes no inerciales:

$$\frac{d\vec{r}}{dt} = \frac{\partial\vec{r}}{\partial t} + \vec{\omega} \wedge \vec{r}$$

para calcular la derivada del vector $\vec{r}$. Lo que podemos hacer en esos casos es calcular dicho vector en los ejes inerciales, e inventarnos dos vectores $\frac{\partial\vec{r}}{\partial t}$ y $\vec{\omega}$ que den el resultados correcto. Hacemos entonces:

$$\frac{\partial\vec{r}}{\partial t} + \vec{\omega} \wedge \vec{r} = L_{wh} \left[\begin{Bmatrix} \dot{x}_h \\ \dot{y}_h \\ \dot{z}_h \end{Bmatrix} + \vec{\omega}_h \wedge \begin{Bmatrix} x_h \\ y_h \\ z_h \end{Bmatrix} \right]$$

Como cuando se produce el cambio brusco de ejes viento hemos seleccionado que sea $\psi_w = 0$, entonces L_{wh} es la matriz unidad, por lo que podemos escoger:

$$\frac{\partial\vec{r}}{\partial t} = \begin{Bmatrix} \dot{x}_h \\ \dot{y}_h \\ \dot{z}_h \end{Bmatrix} \qquad\qquad \vec{\omega} = \vec{\omega}_h$$

Esto es equivalente a suponer:

$$\dot{\psi}_w = 0$$

2.5.2. Velocidad relativa al viento

Suponemos que conocemos la velocidad del viento en ejes inerciales, entonces la velocidad del centro de masas respecto al viento en ejes cuerpo es:

$$\begin{Bmatrix} u_{rw} \\ v_{rw} \\ w_{rw} \end{Bmatrix}_B = \begin{Bmatrix} u \\ v \\ w \end{Bmatrix}_B + L_{BI} \begin{Bmatrix} u_w \\ v_w \\ w_w \end{Bmatrix}_I \tag{2.7}$$

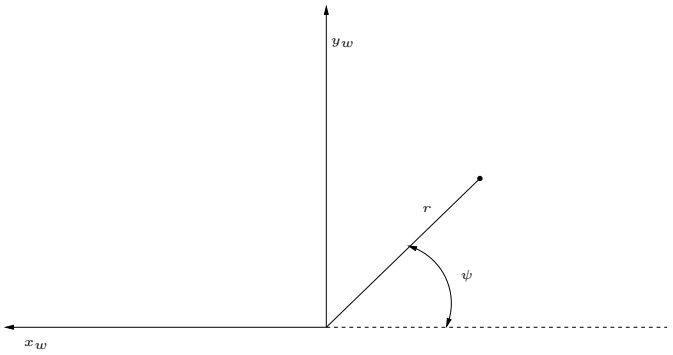


Figura 2.4: Coordenadas rotor

2.5.3. Velocidad relativa al viento en ejes viento

Todo el cálculo de las fuerzas sobre el rotor se realizan utilizando ejes viento para simplificar las expresiones. La matriz de cambio de base de ejes rotor a ejes viento es:

$$L_{wh} = \begin{bmatrix} \cos\psi_w & \sin\psi_w & 0 \\ -\sin\psi_w & \cos\psi_w & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Y la velocidad relativa al viento del rotor en estos ejes es:

$$u_{hrww} = \sqrt{u_{hrwh}^2 + v_{hrwh}^2} \tag{2.8}$$

$$v_{hrww} = 0 \tag{2.9}$$

$$w_{hrww} = w_{hrwh} \tag{2.10}$$

2.5.4. Coordenadas rotor

Las coordenadas de un punto arbitrario contenido en el plano del rotor se dan mediante el azimut ψ y la distancia al eje de rotación r (ver figura 2.4). El azimut se encuentra medido, tal como se indica en la figura, respecto a los ejes viento. Algunas distribuciones de magnitudes sobre el rotor se encuentran dadas en función del azimut pero medido respecto a ajes rotor, llamemos a este azimut ψ_h, entonces haciendo el cambio $\psi = \psi_h + \psi_w$ tenemos para el paso y batimiento:

$$\begin{Bmatrix} \theta_{1sw} \\ \theta_{1cw} \end{Bmatrix} = \triangle \begin{Bmatrix} \theta_{1s} \\ \theta_{1c} \end{Bmatrix}$$

$$\begin{Bmatrix} \beta_{1sw} \\ \beta_{1cw} \end{Bmatrix} = \triangle \begin{Bmatrix} \beta_{1s} \\ \beta_{1c} \end{Bmatrix}$$

$$\triangle = \begin{bmatrix} \cos\psi_w & \sin\psi_w \\ -\sin\psi_w & \cos\psi_w \end{bmatrix}$$

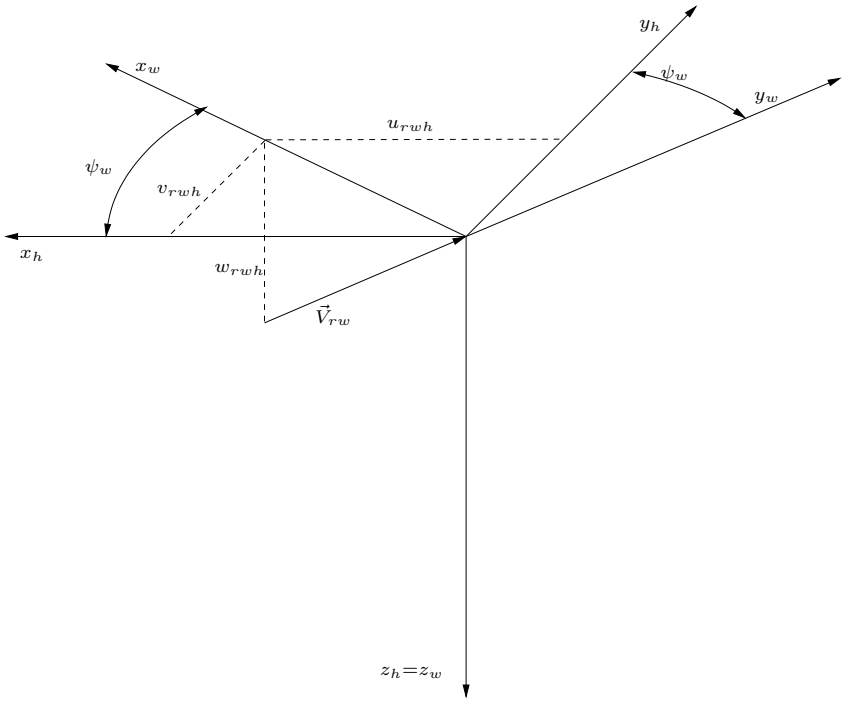


Figura 2.5: Ejes viento

La matriz de cambio de base de ejes rotor a viento es:

$$L_{wh} = \begin{bmatrix} & \triangle & & 0 \\ & & & 0 \\ 0 & 0 & & 1 \end{bmatrix}$$

Si añadimos un número como subíndice a ψ nos referimos al azimut de una pala. Para un rotor de N_b palas tenemos:

$$\psi_i = \bar{t} + \frac{i-1}{N_b} 2\pi$$
$$i = 1, 2, \cdots, N_b$$

De manera similar cuando añadamos un subíndice numérico a una magnitud que esté dada en función de ψ se entiende que sustituimos ψ por ψ_i, es decir, particularizamos dicha magnitud en la i-ésima pala.

$\bar{t} = \Omega t$ es el tiempo adimensional. También aparece frecuentemente en las expresiones la distancia al origen adimensional $\bar{r} = \frac{r}{R}$.

2.6. Ejes rotor de cola

Definidos de forma que el eje z_T es opuesto a la rotación del rotor de cola, el eje x_T paralelo al al x_b y el eje y_T es perpendicular a los anteriores (ver figura 2.6). Para dar la posibilidad de un rotor de cola inclinado se define un ángulo K que es el ángulo que forma y_T con z_b.

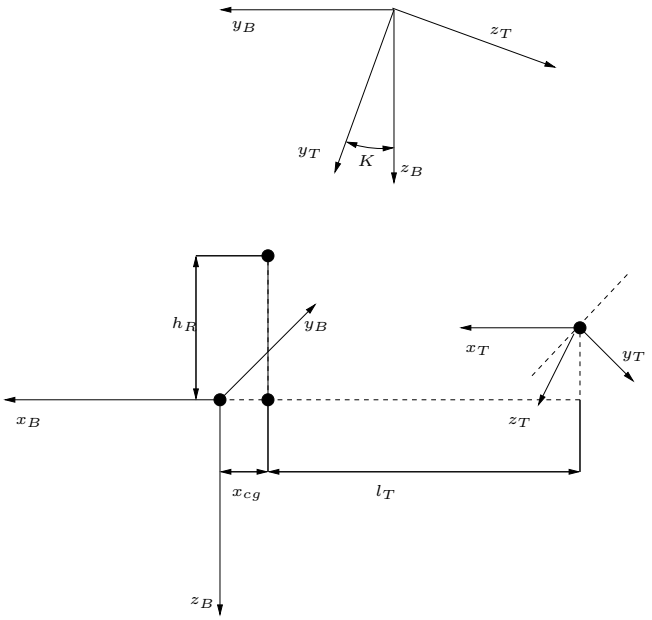


Figura 2.6: Ejes rotor de cola

La matriz de cambio de base de ejes rotor de cola a ejes cuerpo es:

$$L_{BT} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \sin K & -\cos K \\ 0 & \cos K & \sin K \end{bmatrix} \tag{2.11}$$

2.7. Ejes viento del rotor de cola

Definidos de la misma forma que los ejes viento del rotor principal, pero respecto a los ejes cola:

$$\cos(\psi_{wT}) = \frac{u_{rwT}|_T}{\sqrt{u_{rwT}^2|_T + v_{rwT}^2|_T}}$$
$$\sin(\psi_{wT}) = \frac{v_{rwT}|_T}{\sqrt{u_{rwT}^2|_T + v_{rwT}^2|_T}}$$

2.8. Ejes pala

Con origen en el centro de rotación del rotor, el eje x_b se encuentra dirigido a lo largo de la pala y el y_b hacia el borde de salida y contenido en el plano del rotor (ver figura 2.7). Para referirnos a estos ejes utilizamos el subíndice "b"

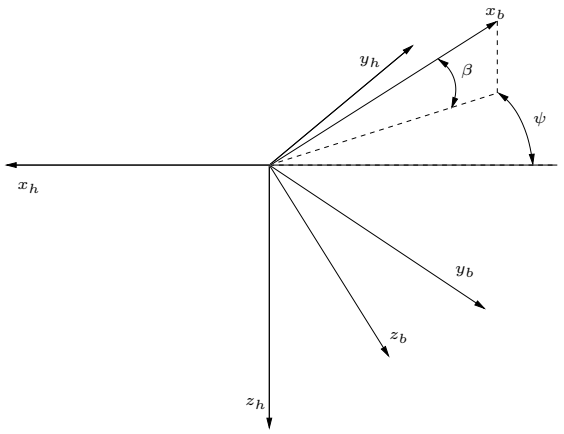


Figura 2.7: Ejes pala

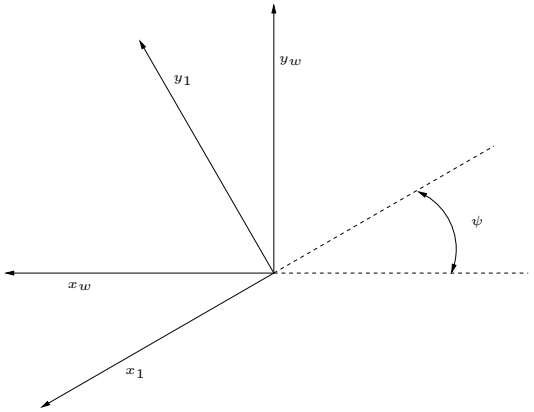


Figura 2.8: Ejes de rotación

2.9. Ejes de rotación

Ejes auxiliares que utilizamos para la deducción de las ecuaciones de batimiento. Se obtienen girando los ejes viento un ángulo $-\psi$ alrededor del eje z_w (ver figura 2.8). Los indicamos con un subíndice 1.

Capítulo 3

Ecuaciones del modelo

De todas las magnitudes asociadas al helicóptero, la mecánica del vuelo se ocupa ante todo del desplazamiento y giro del fuselaje del helicóptero, que es lo que percibe el piloto. Además de estas magnitudes de interés directo hay que incluir aquellos grados de libertad que no siendo el objeto del estudio, afectan el comportamiento del helicóptero, por ejemplo, el batimiento de las palas o la estela del helicóptero. Una vez hayamos planteado las ecuaciones del sistema, agruparemos las variables del problema en un vector de estado $\vec{x}$ de forma que la evolución del sistema esté dada por una ecuación diferencial de la forma:

$$\frac{d\vec{x}}{dt} = \vec{F}(\vec{x}, \vec{u})$$

donde el vector $\vec{u}$ es el vector de control y representa la acción del piloto sobre el helicóptero y otras mangitudes externas.

Para modelizar el helicóptero lo dividimos en un conjunto de subsistemas, que más tarde acoplaremos para dar lugar a nuestro sistema de ecuaciones. De esta forma modularizando el modelo es posible más adelante realizar mejoras de algunos subsistemas sin tener que modificar el resto. Los subsistemas del helicóptero son:

- Rotor principal
- Estela
- Fuselaje
- Rotor de cola
- Grupo motopropulsor
- Estabilizador horizontal
- Estabilizador vertical
- Controles
- Tren de aterrizaje

3.1. Vector de estado

El vector de estado $\vec{x}$ del helicóptero lo forman las siguientes variables:

- u: Componente x de la velocidad en ejes cuerpo
- v: Componente y de la velocidad en ejes cuerpo
- w: Componente z de la velocidad en ejes cuerpo
- p: Componente x de la velocidad angular en ejes cuerpo
- q: Componente y de la velocidad angular en ejes cuerpo
- r: Componente z de la velocidad angular en ejes cuerpo
- q_0: Primera componente del cuaternio de rotación
- q_1: Segunda componente del cuaternio de rotación
- q_2: Tercera componente del cuaternio de rotación
- q_3: Cuarta componente del cuaternio de rotación
- Ω: Velocidad de rotación del rotor principal
- $\dot{Q}_1$: Derivada del par de un motor
- Q_1: Par de un motor
- x_I: Coordenada x de la posición en ejes inerciales
- y_I: Coordenada y de la posición en ejes inerciales
- z_I: Coordenada z de la posición en ejes inerciales

Además existen una serie de variables externas que es necesario aportar para calcular las fuerzas y que forman el vector de control $\vec{u}$:

- η_{0p}: Control colectivo aplicado por el piloto
- η_{1sp}: Control longitudinal aplicado por el piloto
- η_{1cp}: Control lateral aplicado por el piloto
- η_{pp}: Control de pedales aplicado por el piloto
- ρ: Densidad del aire
- T: Temperatura del aire
- h_G: Altura del suelo
- u_w: Componente x en ejes inerciales de la velocidad del viento
- v_w: Componente y en ejes inerciales de la velocidad del viento
- w_w: Componente z en ejes inerciales de la velocidad del viento

Finalmente el vector de salida hacia el sistema visual está dado por:

- λ: Latitud en coordenadas geodésicas
- ϕ: Longitud en coordenadas geodésicas
- h: Altura en coordenadas geodésicas
- θ: Ángulo de Euler, cabeceo
- ϕ: Ángulo de Euler, balance
- ψ: Ángulo de Euler, guiñada

3.2. Rotor principal

El rotor se compone de un número N_b de palas, articuladas o no. Debido a las fuerzas de inercia y aerodinámicas las palas se deforman, siendo fundamental predecir esta deformación para calcular las fuerzas que ejerce el rotor sobre el fuselaje. La resolución exacta del movimiento de las palas, incluso considerándolas suficientemente alargadas como para aplicar el modelo clásico de viga con deformaciones lineales, es muy difícil ya que el problema se encuentra acoplado con la aerodinámica de la estela. En nuestro modelo simplificado supondremos que el movimiento de las palas se compone de un único grado de libertad por pala, su batimiento β y que la pala se mueve como sólido rígido. A pesar de que obviamente para una pala no articulada esto no es cierto, ya que se encuentra empotrada en la cabeza del rotor, constituye una buena aproximación excepto dentro de una pequeña región cerca del empotramiento, donde simularemos las fuerzas estructurales debido a la curvatura mediante un muelle ficticio cuya constante de elasticidad ajustaremos de forma que la frecuencia natural de batimiento del modelo simplificado coincida con la primera frecuencia de batimiento del sistema real. Para clarificar ideas he aquí la lista de aproximaciones utilizadas, de menor a mayor exigencia, y un esquema que compara nuestro modelo simplificado con otro más completo:

- Pala alargada, teoría clásica de vigas
- Deformaciones lineales
- Despreciamos arrastre y torsión, sólo consideramos batimiento
- Consideramos que el batimiento se realiza con la pala rígida

La justificación de esta aproximación se puede realizar a partir de un modelo más completo y para nuestra aplicación que trata movimientos de baja frecuencia en comparación con la rotación de las palas y amplitudes altas, es suficiente.

3.2.1. Cinemática de la pala

Para plantear la ecuación de batimiento necesitamos determinar las fuerzas de inercia y aerodinámicas que actúan sobre la sección de la pala, por lo que necesitamos calcular las aceleraciones y la velocidad relativa al viento de una sección.

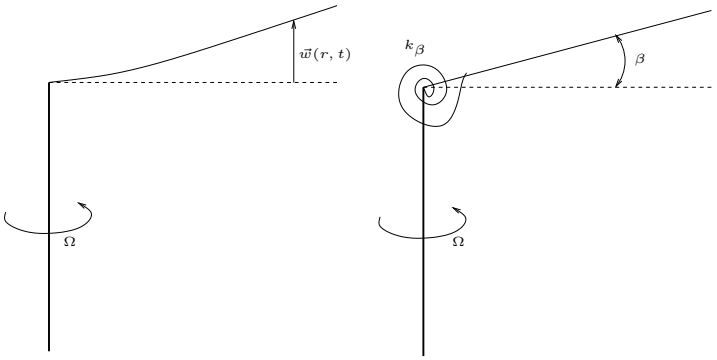


Figura 3.1: Modelo simplificado de batimiento

Sean $\vec{i}_w, \vec{j}_w, \vec{k}_w$ los vectores unitarios asociados a los ejes viento del rotor, anteriormente descritos en 2.5, sea ψ el ángulo que la pala ha rotado desde la posición de referencia (azimut) y sea β el ángulo de batimiento de la pala. Entonces la matriz de cambio de base de ejes viento a ejes pala viene dada por:

$$\begin{Bmatrix} \vec{i}_b \\ \vec{j}_b \\ \vec{k}_b \end{Bmatrix} = \begin{bmatrix} -\cos\beta & 0 & -\sin\beta \\ 0 & -1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{bmatrix} \begin{bmatrix} \cos\psi & \sin\psi & 0 \\ \sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{Bmatrix} \vec{i}_w \\ \vec{j}_w \\ \vec{k}_w \end{Bmatrix}$$

Suponiendo $\beta \ll 1$ y despreciando términos no lineales, la matriz de cambio de base de ejes viento a ejes pala, que denotaremos como L_{bw} queda:

$$L_{bw} = \begin{bmatrix} -\cos\psi & \sin\psi & -\beta \\ -\sin\psi & \cos\psi & 0 \\ -\beta\cos\psi & \beta\sin\psi & 1 \end{bmatrix}$$

La posición de una sección de pala arbitraria, a una distancia r del origen del rotor, viene dada por:

$$\vec{r} = r\vec{i}_b$$

Para derivar la velocidad y aceleración de la sección de pala, utilizaremos unos ejes auxiliares, los ejes de rotación. La velocidad angular de estos ejes respecto a unos ejes inerciales es:

$$\vec{\omega}_1 = \omega_x\vec{i}_1 + \omega_y\vec{j}_1 + \omega_z\vec{k}_1$$

Donde:

$$\begin{aligned} \omega_x &= p_w\cos\psi - q_w\sin\psi \\ \omega_y &= p_w\sin\psi + q_w\cos\psi \\ \omega_z &= r_w - \Omega \end{aligned}$$

p_w, q_w, r_w es la velocidad angular de los ejes viento en ejes viento (ver ecuación 2.6)

La velocidad del origen de los ejes rotor en ejes viento es (ver ecuación 2.10)

$$\vec{V_0} = u_{hrww}\vec{i}_w + w_{hrww}\vec{k}_w$$

Con todos estos datos aplicamos las fórmulas de derivación en ejes no inerciales:

$$\vec{v} = \vec{V_0} + \vec{\omega}_1 \wedge \vec{r} + \frac{\partial \vec{r}}{\partial t}$$
$$\vec{a} = \vec{a}_0 + \vec{\omega}_1 \wedge (\vec{\omega}_1 \wedge \vec{r}) + \vec{\alpha_1} \wedge \vec{r} + 2\vec{\omega_1} \wedge \frac{\partial \vec{r}}{\partial t} + \frac{\partial^2 \vec{r}}{\partial t^2}$$

Con lo que las componentes de la velocidad relativa al viento de la sección en ejes palas es:

$$u_b = -\cos\psi u_{hrww} + \beta w_{hrww}$$
$$v_b = -\sin\psi u_{hrww} + r(-\beta\omega_x + \omega_z)$$
$$w_b = -\beta\cos\psi u_{hrww} + w_{hrww} + r(\omega_y - \dot{\beta})$$

Para calcular la velocidad de la sección de la pala relativa al viento suponemos una velocidad inducida sobre el rotor del tipo:

$$\lambda(\bar{r}, \psi, \bar{t}) = \lambda_0(\bar{t}) + \bar{r}\lambda_1(\bar{t})$$
$$\lambda_1(\psi, \bar{t}) = \lambda_{1cw}(\bar{t})\cos\psi + \lambda_{1sw}(\bar{t})\sin\psi$$

Donde $\lambda = \frac{v_i}{\Omega R}$ es la velocidad inducida adimensionalizada, y se toma positiva según el eje z_w (hacia abajo). En la sección dedicada al modelado de la estela calcularemos las ecuaciones que nos determinan la anterior velocidad, dejándola de momento como una incógnita en las ecuaciones del batimiento.

Las componentes de la velocidad relativa al viento, ya adimensionalizadas con ΩR valen:

$$\bar{U}_T = \bar{r}(1 + \bar{\omega}_x\beta) + \mu\sin\psi$$
$$\bar{U}_P = \mu_z - \lambda_0 - \beta\mu\cos\psi + \bar{r}(\bar{\omega_y} - \beta^* - \lambda_1)$$

El asterisco indica derivación respecto al tiempo adimensional y

$$\mu = \frac{u_{hrww}}{\Omega R}$$
$$\mu_z = \frac{w_{hrww}}{\Omega R}$$

son respectivamente el parámetro de avance y el parámetro de descenso. Para ver el significado de U_P y U_T ver figura 3.2.

También obtenemos la aceleración de un elemento de pala, donde despreciamos los términos no lineales en la velocidad angular (de nuevo limitamos el

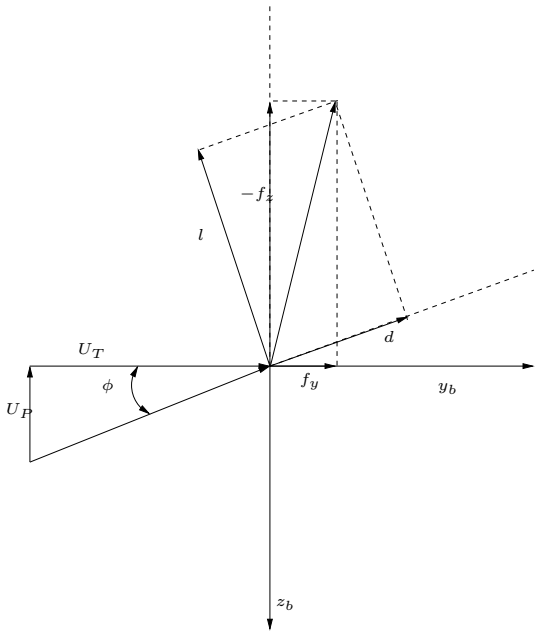


Figura 3.2: Fuerzas aerodinámicas sobre el perfil

modelo a las zonas de baja frecuencia), la aceleración del helicóptero y la componente de la velocidad angular r_w. De las componentes de la aceleración sólo nos interesa a_{zb}, ya que el resto no las vamos a incluir en las resultantes sobre la pala, de todas formas por completitud y pensando en mejoras futuras del modelo he aquí todas las componentes:

$$
\begin{aligned}
a_{xb} &= r(-\Omega^2 + 2\dot{\beta}\omega_y - 2\Omega\beta\omega_x) \\
a_{yb} &= -\omega_y\Omega - \beta\dot{\omega}_x + \dot{\omega}_z - 2\dot{\beta}(\omega_x + \omega_z\beta) \\
a_{zb} &= r\left[2\Omega\left\{\left(p_w + \frac{q_w^*}{2}\right)\cos\psi + \left(-q_w + \frac{p_w^*}{2}\right)\sin\psi\right\} - \Omega^2\beta - \ddot{\beta}\right]
\end{aligned}
$$

3.2.2. Fuerzas aerodinámicas

Suponiendo ángulos pequeños de ataque, aplicamos la teoría linealizada de perfiles (ver figura 3.2).

La sustentación y resistencia de la pala por unidad de longitud es:

$$
\begin{aligned}
l &= \frac{1}{2}\rho\left(U_T^2 + U_P^2\right) c a_0 c_L \\
d &= \frac{1}{2}\rho\left(U_T^2 + U_P^2\right) c\delta
\end{aligned}
$$

Aproximamos los coeficientes:

$$c_L = a_0 (\theta + \phi)$$
$$\phi \simeq \frac{U_P}{U_T}$$
$$\delta = \delta_0 + \delta_2 c_L^2$$

Además aproximaremos el coeficiente de resistencia por un coeficiente de resistencia medio:

$$\delta = \delta_0 + \delta_1 c_T + \delta_2 c_T^2$$

donde c_T es el coeficiente de tracción del rotor (Ver páginas 54-55 de [8])

Proyectando la sustentación y resistencia sobre los ejes pala, y aplicando las siguientes aproximaciones:

$$U_T^2 + U_P^2 \simeq U_T^2$$
$$f_z = -(l\cos\phi + d\sin\phi) \simeq -l - d\phi \simeq -l$$
$$f_y = d\cos\phi - l\sin\phi \simeq d - l\phi$$

Desarrollando un poco más podemos poner estas fuerzas aerodinámicas en función de las velocidades adimensionalizadas:

$$f_z = -\frac{\rho c a_0 (\Omega R)^2}{2}\left(\bar{U}_T^2\theta + \bar{U}_T\bar{U}_P\right)$$
$$f_y = -\frac{\rho c a_0 (\Omega R)^2}{2}\left(\bar{U}_P\bar{U}_T\theta + \bar{U}_P^2 - \frac{\bar{U}_T^2\delta}{a_0}\right)$$

3.2.3. Ecuación de batimiento

Ecuación de batimiento para las palas individuales

Aplicamos la ecuación de momentos sobre la articulación o empotramiento de la pala en el eje del rotor:

$$\int_0^2 r\,(f_z - m a_{zb})\,dr + K_\beta\beta = 0$$

De ahora en adelante utilizamos las siguientes definiciones:

$$I_\beta = \int_0^R mr^2 dr \qquad \lambda_\beta^2 = 1 + \frac{K_\beta}{I_\beta\Omega^2}$$
$$\bar{p}_w = \frac{p_w}{\Omega} \qquad \bar{q}_w = \frac{q_w}{\Omega}$$
$$\bar{p}_w^* = \frac{d\bar{p}_w}{d\bar{t}} = \frac{\dot{p}_w}{\Omega^2} \qquad \bar{q}_w^* = \frac{d\bar{q}_w}{d\bar{t}} = \frac{\dot{q}_w}{\Omega^2}$$
$$\gamma = \frac{\rho c a_0 R^4}{I_\beta} \qquad \bar{r} = \frac{r}{R}$$

En general, una barra sobre la variable significa que está adimensionalizada, un punto, que es la derivada respecto al tiempo, y un asterisco que es la derivada

respecto al tiempo adimensional $\bar{t} = t\Omega$. La ecuación de batimiento para la i-ésima pala queda entonces:

$$\beta_i{}^{**} + \lambda_\beta^2 \beta_i = 2\left[\left(\bar{p}_w + \frac{\bar{q}_w^*}{2}\right)\cos\psi_i - \left(\bar{q}_w - \frac{\bar{p}_w^*}{2}\right)\sin\psi_i\right] + \\ + \frac{\gamma}{2}\int_0^1 \left(\bar{U}_{T_i}^2\theta_i + \bar{U}_{P_i}\bar{U}_{T_i}\right)\bar{r}d\bar{r}$$

Tenemos N_b ecuaciones como la anterior, una por cada pala del helicóptero.

Merece la pena discutir el significado en la ecuación de algunos coeficientes que aparecen en las ecuaciones y sus valores típicos:

- λ_β es la frecuencia natural de batimiento de las palas (adimensionalizada con Ω) y debido a la gran velocidad de rotación de las palas y la inercia de batimiento se encuentra próxima a uno. Valores típicos suelen ser del orden de 1.1 . Debido a que las fuerzas aerodinámicas se encuentran controladas por el paso de las palas, y éste se aplica con la frecuencia de giro del rotor, las palas se encuentran excitadas muy próximas a su resonancia, lo que tiene dos consecuencias: primero, que el sistema de control de paso es muy efectivo y segundo, que hay un desfase muy próximo a 90^o entre paso y batimiento.

- γ es el número de Locke y da una idea de la relación fuerzas aerodinámicas/fuerzas de inercia, valores típicos caen entre 5 y 7.

Sustituyendo las expresiones para le velocidad relativa al viento obtenemos una ecuación de coeficientes periódicos para cada pala del rotor:

$$\begin{aligned} M &= \frac{1}{2}\int_0^1 \left(\bar{U}_T^2\theta + \bar{U}_P\bar{U}_T\right)\bar{r}d\bar{r} \\ &= M_\beta\beta + M_{\beta^*}\beta^* + \\ &\quad M_{\theta_0}\theta_0 + M_{\theta_t}\theta_t + M_{\theta_{1cw}}\theta_{1cw} + M_{\theta_{1sw}}\theta_{1sw} + \\ &\quad M_{\bar{p}_w}\bar{p}_w + M_{\bar{q}_w}\bar{q}_w + \\ &\quad M_{\lambda_0}(\mu_z - \lambda_0) + M_{\lambda_{1cw}}\lambda_{1cw} + M_{\lambda_{1sw}}\lambda_{1sw} \end{aligned}$$

Donde los coeficientes anteriores son función únicamente de ψ y de μ y vienen dados por:

$$
\begin{aligned}
M_{\beta} &= -\mu\left(\frac{1}{6}\cos\psi + \frac{\mu}{8}\sin 2\psi\right)\\
M_{\beta^*} &= -\left(\frac{1}{8} + \frac{\mu}{6}\sin\psi\right)\\
M_{\lambda_0} &= \frac{1}{6} + \frac{\mu}{4}\sin\psi\\
M_{\lambda_{1cw}} &= -\left(\frac{1}{8}\cos\psi + \frac{\mu}{12}\sin 2\psi\right)\\
M_{\lambda_{1sw}} &= -\frac{\mu}{12} - \frac{1}{8}\sin\psi + \frac{1}{12}\cos 2\psi\\
M_{\bar{p}_w} &= -M_{\lambda_{1sw}}\\
M_{\bar{q}_w} &= -M_{\lambda_{1cw}}\\
M_{\theta_0} &= \frac{1}{8}(1+\mu^2) + \frac{1}{3}\mu\sin\psi - \frac{\mu^2}{8}\cos 2\psi\\
M_{\theta_t} &= \frac{1}{10} + \frac{\mu^2}{12} + \frac{\mu}{4}\sin\psi - \frac{\mu^2}{12}\cos 2\psi\\
M_{\theta_{1cw}} &= \left(\frac{1}{8} + \frac{\mu^2}{16}\right)\cos\psi + \frac{\mu}{6}\sin 2\psi - \frac{\mu^2}{16}\cos 3\psi\\
M_{\theta_{1sw}} &= \frac{\mu}{6} + \left(\frac{1}{8} + \frac{3}{16}\mu^2\right)\sin\psi - \frac{\mu}{6}\cos 2\psi - \frac{1}{16}\sin 3\psi
\end{aligned}
$$

Ecuación de batimiento en coordenadas multipala

Transformamos el sistema de ecuaciones de las palas individuales a coordenadas multipala, que para un rotor de cuatro palas se encuentran definidas de la forma:

$$
\begin{aligned}
\beta_0 &= \frac{1}{4}\sum_{i=1}^{4}\beta_i\\
\beta_d &= \frac{1}{4}\sum_{i=1}^{4}\beta_i(-1)^2\\
\beta_{1cw} &= \frac{1}{2}\sum_{i=1}^{4}\beta_i\cos\psi_i\\
\beta_{1sw} &= \frac{1}{2}\sum_{i=1}^{4}\beta_i\sin\psi_i
\end{aligned}
$$

Si agrupamos los batimientos de las palas y las coordenadas multipala en dos vectores:

$$
\beta_I = \begin{Bmatrix} \beta_1 \\ \beta_2 \\ \beta_3 \\ \beta_4 \end{Bmatrix} \qquad\qquad \beta_M = \begin{Bmatrix} \beta_0 \\ \beta_d \\ \beta_{1cw} \\ \beta_{1sw} \end{Bmatrix}
$$

Podemos escribir las anteriores transformaciones en forma matricial:

$$\vec{\beta}_I = L_\beta \vec{\beta}_M$$

Donde la anterior matriz de cambio de coordenadas es:

$$L_\beta = \begin{bmatrix} 1 & -1 & \cos\bar{t} & \sin\bar{t} \\ 1 & 1 & -\sin\bar{t} & \cos\bar{t} \\ 1 & -1 & -\cos\bar{t} & -\sin\bar{t} \\ 1 & 1 & \sin\bar{t} & -\cos\bar{t} \end{bmatrix}$$

$$L_\beta^{-1} = \frac{1}{4} \begin{bmatrix} 1 & 1 & 1 & 1 \\ -1 & 1 & -1 & 1 \\ 2\cos\bar{t} & -2\sin\bar{t} & -2\cos\bar{t} & 2\sin\bar{t} \\ 2\sin\bar{t} & 2\cos\bar{t} & -2\sin\bar{t} & -2\cos\bar{t} \end{bmatrix}$$

Agrupamos también en vectores las siguientes magnitudes:

$$\vec{\theta} = \begin{Bmatrix} \theta_0 \\ \theta_t \\ \theta_{1sw} \\ \theta_{1cw} \end{Bmatrix} \qquad \vec{\lambda} = \begin{Bmatrix} \lambda_{1sw} \\ \lambda_{1cw} \end{Bmatrix} \tag{3.1}$$

$$\vec{\omega} = \begin{Bmatrix} \bar{p}_w \\ \bar{q}_w \end{Bmatrix} \qquad \vec{\omega}^* = \begin{Bmatrix} \bar{p}^*_w \\ \bar{q}^*_w \end{Bmatrix}$$

La ecuación de batimiento en coordenadas multipala es entonces:

$$\vec{\beta}^{**}_M + C_M \vec{\beta}^{*}_M + D_M \vec{\beta_M} = H_{M_\theta}\vec{\theta} + H_{M_\lambda}\vec{\lambda} + \vec{H}_{M_{\lambda_0}}(\mu_z - \lambda_0) + H_{M_\omega}\vec{\omega} + H_{M_{\omega^*}}\vec{\omega}^*$$

Las anteriores matrices valen:

$$C_M = \frac{\gamma}{8}\begin{bmatrix} 1 & 0 & 0 & \frac{2}{3}\mu \\ 0 & 1 & -\frac{2}{3}\mu\sin 2\bar{t} & \frac{2}{3}\mu\cos 2\bar{t} \\ 0 & -\frac{4}{3}\mu\sin 2\bar{t} & 1 & \frac{16}{\gamma} \\ \frac{4}{3}\mu & \frac{4}{3}\mu\cos 2\bar{t} & -\frac{16}{\gamma} & 1 \end{bmatrix}$$

$$D_M = \frac{\gamma}{8}\begin{bmatrix} \frac{8\lambda_\beta^2}{\gamma} & -\mu^2\sin 2\bar{t} & 0 & 0 \\ -\mu^2\sin 2\bar{t} & \frac{8\lambda_\beta^2}{\gamma} & -\frac{4}{3}\mu\cos 2\bar{t} & -\frac{4}{3}\mu\sin 2\bar{t} \\ \frac{4}{3}\mu & -\frac{4}{3}\mu\cos 2\bar{t} & \frac{8}{\gamma}(\lambda_\beta^2-1)+\frac{\mu^2}{2}\sin 2\bar{t} & 1+\frac{\mu^2}{2}-\frac{\mu^2}{2}\cos 2\bar{t} \\ 0 & -\frac{4}{3}\mu\sin 2\bar{t} & -1+\frac{\mu^2}{2}-\frac{\mu^2}{2}\cos 2\bar{t} & \frac{8}{\gamma}(\lambda_\beta^2-1)-\frac{\mu^2}{2}\sin 2\bar{t} \end{bmatrix}$$

$$H_{M_\theta} = \frac{\gamma}{8}\begin{bmatrix} 1+\mu^2 & \frac{4}{5}+\frac{2}{3}\mu^2 & \frac{4}{3}\mu & 0 \\ \mu^2\cos 2\bar{t} & \frac{2}{3}\mu^2\cos 2\bar{t} & \frac{4}{3}\mu\cos 2\bar{t} & -\frac{4}{3}\mu\sin 2\bar{t} \\ 0 & 0 & -\frac{1}{2}\sin 4\bar{t} & 1+\frac{\mu^2}{2}-\frac{\mu^2}{2}\cos 4\bar{t} \\ \frac{8}{3}\mu & 2\mu & 1+\frac{3}{2}\mu^2+\frac{1}{16}\cos 4\bar{t} & -\frac{\mu^2}{2}\sin 4\bar{t} \end{bmatrix}$$

$$\vec{H}_{M_{\lambda_0}} = \frac{\gamma}{8}\begin{Bmatrix} \frac{4}{3} \\ 0 \\ 0 \\ 2\mu \end{Bmatrix}$$

$$H_{M_\lambda} = \frac{\gamma}{8}\begin{bmatrix} -\frac{2}{3}\mu & 0 \\ -\frac{2}{3}\cos 2\bar{t} & \frac{2}{3}\mu\sin 2\bar{t} \\ 0 & -1 \\ -1 & 0 \end{bmatrix}$$

$$H_{M_\omega} = \frac{\gamma}{8}\begin{bmatrix} \frac{2}{3}\mu & 0 \\ \frac{2}{3}\cos 2\bar{t} & -\frac{2}{3}\mu\sin 2\bar{t} \\ \frac{16}{\gamma} & 1 \\ 1 & -\frac{16}{\gamma} \end{bmatrix}$$

$$H_{M_{\omega^*}} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 1 \\ 1 & 0 \end{bmatrix}$$

Aplicamos las siguientes simplificaciones:

- Suponemos que respecto al rotor el fuselaje se mueve suficientemente lento como para suponer valores estacionarios de velocidad angular, coeficiente de avance, etc....
- Eliminamos los coeficientes periódicos, quedándonos con un sistema de ecuaciones diferenciales en coeficientes constantes.
- Sustituimos el sistema de ecuaciones diferenciales por la solución estacionaria. Para que esto sea justificable es necesario que se cumpla la hipótesis de que los autovalores del rotor y del fuselaje se encuentren suficientemente separados, en cuyo caso el rotor sólo influye al fuselaje a través de su solución estacionaria. Al realizar esta simplificación también suponemos velocidad inducida estacionaria.

Finalmente obtenemos la siguiente ecuación de batimiento:

$$A_{\beta\beta}\vec{\beta} + A_{\beta\lambda}\vec{\lambda} = A_{\beta\theta}\vec{\theta} + A_{\beta\omega}\omega + A_{\beta\dot{\omega}}\dot{\vec{\omega}} + \vec{A}_{\beta\lambda_0}\left(\mu_z - \lambda_0\right)$$

Donde las anteriores matrices valen:

$$A_{\beta\beta} = \begin{bmatrix} \frac{8\lambda_\beta^2}{\gamma} & 0 & 0 \\ \frac{4}{3}\mu & \frac{8\left(\lambda_\beta^2 - 1\right)}{\gamma} & 1 + \frac{\mu^2}{2} \\ 0 & -\left(1 - \frac{\mu^2}{2}\right) & \frac{8\left(\lambda_\beta^2 - 1\right)}{\gamma} \end{bmatrix}$$

$$A_{\beta\lambda} = \begin{bmatrix} \frac{2}{3}\mu & 0 \\ 0 & 1 \\ 1 & 0 \end{bmatrix}$$

$$A_{\beta\theta} = \begin{bmatrix} 1 + \mu^2 & 4\left(\frac{1}{5} + \frac{\mu^2}{6}\right) & \frac{4}{3}\mu & 0 \\ 0 & 0 & 0 & 1 + \frac{\mu^2}{2} \\ \frac{8}{3}\mu & 2\mu & 1 + \frac{3}{2}\mu^2 & 0 \end{bmatrix}$$

$$A_{\beta\dot{\omega}} = \begin{bmatrix} 0 & 0 \\ 0 & \frac{8}{\gamma} \\ \frac{8}{\gamma} & 0 \end{bmatrix}$$

$$A_{\beta\omega} = \begin{bmatrix} \frac{2}{3}\mu & 0 \\ \frac{16}{\gamma} & 1 \\ 1 & -\frac{16}{\gamma} \end{bmatrix}$$

$$\vec{A}_{\beta\lambda_0} = \begin{Bmatrix} \frac{4}{3} \\ 0 \\ 2\mu \end{Bmatrix}$$

Y $\vec{\beta}$ no contiene a β_d que es nulo:

$$\vec{\beta} = \begin{Bmatrix} \beta_0 \\ \beta_{1cw} \\ \beta_{1sw} \end{Bmatrix} \tag{3.2}$$

que es un sistema de ecuaciones lineales (porque las hemos linealizado) que nos liga el control, la velocidad inducida y la velocidad y aceleración angular con el batimiento de las palas.

La anterior ecuación todavía no puede ser resuelta ya que desconocemos los valores de la velocidad inducida. De hecho, dichos valores dependen de los momentos sobre las palas, que a su vez dependen del batimiento, por lo que necesitamos completar la anterior ecuación de batimiento más adelante para resolver conjuntamente con las ecuaciones de la velocidad inducida.

3.2.4. Fuerzas del Rotor

Integrando para cada pala las fuerzas que actúan sobre cada perfil, obtenemos las resultantes sobre el rotor. Despreciando las aceleraciones de inercia en cada sección por cancelarse al integrar sobre todas las palas los términos significativos, las resultantes dependen únicamente de las fuerzas aerodinámicas:

$$X_{hw} = \sum_{i=1}^{N_b} \int_0^R \{-f_{z_i}\beta_i \cos(\psi_i) - f_{y_i} \sin(\psi_i)\}\, dr$$
$$Y_{hw} = \sum_{i=1}^{N_b} \int_0^R \{f_{z_i}\beta_i \sin(\psi_i) - f_{y_i} \cos(\psi_i)\}\, dr$$
$$Z_{hw} = \sum_{i=1}^{N_b} \int_0^R f_{z_i} dr$$

Desarrollando las anteriores expresiones y llamando:

$$F^{(1)}(\psi_i) = \int_0^1 \left[\bar{U}_T^2\theta_i + \bar{U}_P\bar{U}_T\right] d\bar{r}$$
$$F^{(2)}(\psi_i) = \int_0^1 \left[\bar{U}_P\bar{U}_T\theta_i + \bar{U}_P^2 - \frac{\delta_i\bar{U}_T^2}{a_0}\right] d\bar{r}$$

tenemos para los coeficientes de fuerzas:

$$C_{xw} = \frac{X_{hw}}{\frac{1}{2}\rho(\Omega R)^2\pi R^2}$$
$$C_{yw} = \frac{Y_{hw}}{\frac{1}{2}\rho(\Omega R)^2\pi R^2}$$
$$C_{zw} = \frac{Z_{hw}}{\frac{1}{2}\rho(\Omega R)^2\pi R^2}$$

las siguientes expresiones donde se han eliminado términos no estacionarios:

$$\left(\frac{2C_{xw}}{a_0 s}\right) = \left(\frac{F_0^{(1)}}{2} + \frac{F_{2c}^{(1)}}{4}\right)\beta_{1cw} + \frac{F_{1c}^{(1)}}{2}\beta_0 + \frac{F_{2s}^{(1)}}{4}\beta_{1sw} + \frac{F_{1s}^{(2)}}{2}$$
$$\left(\frac{2C_{yw}}{a_0 s}\right) = \left(-\frac{F_0^{(1)}}{2} + \frac{F_{2c}^{(1)}}{4}\right)\beta_{1sw} - \frac{F_{1s}^{(1)}}{2}\beta_0 - \frac{F_{2s}^{(1)}}{4}\beta_{1cw} + \frac{F_{1c}^{(2)}}{2}$$
$$\left(\frac{2C_{zw}}{a_0 s}\right) = -F_0^{(1)}$$

Los términos armónicos de las funciones $F^{(1)}$ y $F^{(2)}$ valen:

$$F_0^{(1)} = \theta_0\left(\frac{1}{3}+\frac{\mu^2}{2}\right)+\frac{\mu}{2}\left(\theta_{1sw}+\frac{\bar{p}_w}{2}\right)+\frac{\mu_z-\lambda_0}{2}+\frac{1}{4}\left(1+\mu^2\right)\theta_t$$
$$F_{1s}^{(1)} = \frac{\alpha_{1sw}}{3}+\mu\left(\theta_0+\mu_z-\lambda_0+\frac{2}{3}\theta_t\right)$$
$$F_{1c}^{(1)} = \frac{\alpha_{1cw}}{3}-\mu\frac{\beta_0}{2}$$
$$F_{2s}^{(1)} = \frac{\mu}{2}\left[\frac{\alpha_{1cw}}{2}+\frac{\theta_{1cw}-\beta_{1sw}}{2}-\mu\beta_0\right]$$
$$F_{2c}^{(1)} = -\frac{\mu}{2}\left[\frac{\alpha_{1sw}}{2}+\frac{\theta_{1sw}+\beta_{1cw}}{2}+\mu\left(\theta_0+\frac{\theta_t}{2}\right)\right]$$

$$\begin{aligned}F_{1s}^{(2)} = {} & \frac{\mu^2}{2}\beta_0\beta_{1sw}+\left(\mu_z-\lambda_0-\frac{\mu}{4}\beta_{1cw}\right)\left(\alpha_{1sw}-\theta_{1sw}\right)-\frac{\mu}{4}\beta_{1sw}\left(\alpha_{1cw}-\theta_{1cw}\right)\\ & +\theta_0\left[\frac{\alpha_{1sw}-\theta_{1sw}}{3}+\mu\left(\mu_z-\lambda_0\right)-\frac{\mu^2}{4}\beta_{1cw}\right]\\ & +\theta_t\left[\frac{\alpha_{1sw}-\theta_{1sw}}{4}+\frac{\mu}{2}\left(\mu_z-\lambda_0-\frac{\beta_{1cw}\mu}{4}\right)\right]\\ & +\theta_{1sw}\left[\frac{\mu_z-\lambda_0}{2}+\mu\left(\frac{3}{8}(\bar{p}_w-\lambda_{1sw})+\frac{\beta_{1cw}}{4}\right)\right]\\ & +\theta_{1cw}\frac{\mu}{4}\left(\frac{\bar{q}_w-\lambda_{1cw}}{2}-\beta_{1sw}-\mu\beta_0\right)\\ & -\frac{\delta\mu}{a_0}\end{aligned}$$

$$\begin{aligned}F_{1c}^{(2)} = {} & \left(\alpha_{1cw}-\theta_{1cw}-2\beta_0\mu\right)\left(\mu_z-\lambda_0-\frac{3}{4}\mu\beta_{1cw}\right)-\frac{\mu}{4}\beta_{1cw}(\alpha_{1sw}-\theta_{1sw})\\ & +\theta_0\left[\frac{\alpha_{1cw}-\theta_{1cw}}{3}-\frac{\mu}{2}\left(\beta_0+\frac{\mu}{2}\beta_{1sw}\right)\right]\\ & +\theta_t\left[\frac{\alpha_{1cw}-\theta_{1cw}}{4}-\mu\left(\frac{\beta_0}{3}+\frac{\beta_{1sw}\mu}{8}\right)\right]\\ & +\theta_{1cw}\left[\frac{\mu_z-\lambda_0}{2}+\frac{\mu}{4}\left(\frac{\bar{p}_w-\lambda_{1sw}}{2}-\beta_{1cw}\right)\right]\\ & +\theta_{1sw}\frac{\mu}{4}\left(\frac{\bar{q}_w-\lambda_{1cw}}{2}-\beta_{1sw}-\mu\beta_0\right)\end{aligned}$$

Donde los ángulos efectivos de ataque de la pala son:

$$\alpha_{1sw} = \bar{p}_w-\lambda_{1sw}+\beta_{1cw}+\theta_{1sw}$$
$$\alpha_{1cw} = \bar{q}_w-\lambda_{1cw}-\beta_{1sw}+\theta_{1cw}$$

Las resultantes de momentos son más sencillas, se pueden obtener directamente de la acción del muelle equivalente y del par del rotor:

$$L_{hw} = -\frac{N_b}{2} K_\beta \beta_{1sw} - \frac{Q_R}{2} \beta_{1cw}$$
$$M_{hw} = -\frac{N_b}{2} K_\beta \beta_{1cw} + \frac{Q_R}{2} \beta_{1sw}$$

El par sobre el rotor:

$$N_h = \sum_{i=1}^{N_b} \int_0^R r_b \left(f_y - m a_{yb}\right)_i dr_b$$

De forma aproximada podemos escribir para el coeficiente de par aerodinámico:

$$\frac{2C_Q}{a_0 s} = -(\mu_z - \lambda_0)\left(\frac{2c_T}{a_0 s}\right) + \mu\left(\frac{2C_{xw}}{a_0 s}\right) + \frac{\delta}{4a_0}\left(1 + 3\mu^2\right)$$

El par total queda:

$$N_h = \frac{1}{2}\rho(\Omega R)^2 \pi R^2 c_Q + I_{rt}\dot{\Omega}$$

Donde I_{rt} es el momento de inercia del rotor y sistema de transmisión respecto al eje de rotación.

3.3. Estela

En la anterior sección determinamos los movimientos y fuerzas que actúan sobre el rotor principal pero dejamos sin calcular la velocidad inducida por la estela en el plano del rotor. A continuación calculamos dicha velocidad y resolvemos finalmente el batimiento y la velocidad inducida simultáneamente.

A continuación se calculan expresiones que permiten calcular la velocidad inducida media basándose en la Teoría de la Cantidad de Movimiento Modificada (TCMM) y la teoría del elemento pala (ver [17]). Para calcular el efecto suelo teniendo en cuenta la velocidad de avance del helicóptero nos basamos en [7]. Por último mejoramos la aproximación de velocidad uniforme mediante el modelo de estela de Pitt-Peters, ver [15] y [14].

3.3.1. TCMM

Utilizamos la TCMM para eliminar el problema de soluciones múltiples en vuelo de descenso a baja velocidad. La TCMM nos proporciona una relación entre la sustentación y la velocidad inducida:

$$T = 2\rho S \sqrt{\frac{1}{k_\nu^2} V_x^2 + \left(\frac{1}{k_\nu^2} - \frac{1}{k_i^2}\right) V_z^2 + \frac{1}{k_i^2}\left(V_z - v_i\right)^2} \frac{v_i}{k_i}$$

Donde las constantes k_i y k_ν se determinan experimentalmente para obtener los valores esperados en autorrotación y en molinete frenante, y valen:

$$k_i = \left(\frac{9}{5}\right)^{\frac{1}{4}}$$
$$k_\nu = \left(\frac{5}{4}\right)^{\frac{1}{4}}$$

Y T representa la tracción media del rotor, que calcularemos mediante la teoría del elemento pala, más adelante.

3.3.2. Efecto suelo

A continuación se describe el modelado del efecto suelo, explicando un poco más detalladamente los resultados de [7].

Efecto suelo en vuelo a punto fijo

Sustituimos el rotor por un manantial de intensidad tal que el flujo del manantial sea el mismo que el del rotor. De aquí en adelante, de igual forma que en la referencia, suponemos que manantial es cualquier solución elemental de la ecuación del potencial de velocidades tal que la velocidad es radial, es decir, la velocidad en un punto tiene la misma dirección que la recta que une el origen del manantial con el punto. La velocidad inducida a una distancia r del manantial es:

$$V = \frac{Av_i}{4\pi r^2}$$

Donde v_i es la velocidad inducida en el plano del rotor y $A = \pi R^2$ es el área del rotor.

Si el rotor se encuentra a una altura z sobre el suelo situamos un rotor imagen a una distancia $2z$ del rotor del helicóptero, entonces la velocidad en el rotor será la inducida v_i menos la que induce el rotor imagen:

$$\delta V = \frac{v_i R^2}{16 z^2}$$

Por lo que finalmente:

$$v_{iG} = v_i \left(1 - \frac{1}{16\left(\frac{z}{R}\right)^2}\right)$$

Donde v_{iG} representa la velocidad inducida teniendo en cuenta el efecto suelo, en contraposición con la velocidad inducida sin tener en cuenta el efecto suelo v_i.

Efecto suelo en vuelo de avance

Supongamos el rotor a una distancia z del suelo, inclinado respecto a la horizontal un ángulo i y con una velocidad de avance V. v_i es la velocidad inducida por el rotor normal al plano del rotor, θ es el ángulo que forma la velocidad resultante de la velocidad de avance y la inducida con la normal al

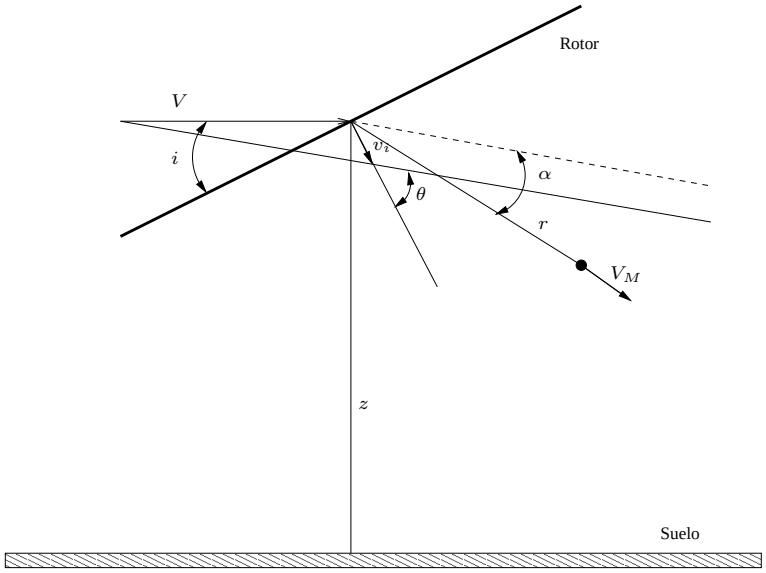


Figura 3.3: Definición de magnitudes

rotor (para i pequeño aproximadamente arctan $\frac{V}{v_i}$), V_M es la velocidad inducida por un manantial que definiremos más adelante, r la distancia medida desde centro del rotor y α un ángulo medido a partir de la resultante, tal como se indica en la figura.

Sustituimos el rotor por un manantial de la forma:

$$V_M = \frac{Q}{r^2}\cos^n \alpha$$

Donde las constantes Q (intensidad del manantial) y $n \geq 1$ están todavía por determinar, para lo cual calculamos el flujo a través de una esfera que contenga al manantial y suponemos que el flujo del manantial es proporcional al del rotor. El valor más pequeño de n para el cual obtenemos flujo no nulo es 2, en cuyo caso el flujo a través de cualquier esfera es $\frac{4\pi Q}{3}$. Igualando dicho flujo al flujo a través del rotor por una constante k tenemos:

$$V_M = \frac{3kAu}{4\pi}\frac{\cos^2 \alpha}{r^2}$$

donde $u = V \sin i + v_i$ es la velocidad resultante normal al rotor.

La velocidad inducida por el rotor imagen sobre el rotor real se obtiene sin más que sustituir $r = 2z$ y $\alpha = \theta + i$:

$$\delta v_i = \frac{3kuR^2}{4\pi r^2}\cos^2(\theta + i)$$

Determinamos la constante k haciendo que la anterior expresión coincida con la que obtuvimos en el caso a de vuelo a punto fijo por lo que $k = \frac{1}{3}$

Por último despreciando i frente a θ llegamos a la expresión:

$$u_G = u\left[1 - \frac{1}{16\left(\frac{z}{R}\right)^2}\frac{1}{1+\left(\frac{\mu}{\lambda}\right)^2}\right]$$

3.3.3. Teoría del elemento pala

Aplicando la teoría linealizada de perfiles bidimensionales y estacionarios obtenemos la siguiente expresión para la sustentación que, por unidad de longitud, realiza la pala i:

$$L_i = \frac{1}{2}\rho a_0 c(\Omega R)^2 \left[\bar{U}_{T_i}^2 \theta_i + \bar{U}_{P_i}\bar{U}_{T_i}\right]$$

3.3.4. Modelo de estela de Pitt-Peters

Hemos calculado la velocidad uniforme que aparece debido al efecto de la tracción media del rotor y hemos corregido dicha velocidad debido al efecto suelo, pero hay una serie de efectos que no estamos prediciendo todavía:

1. La componente no uniforme de velocidad inducida que aparece en vuelo de avance, incluso suponiendo tracción constante en todo el rotor.

2. La componente no uniforme de velocidad inducida que aparece en cualquier situación debido a los momentos (y otras componentes no uniformes de la tracción) del rotor.

3. La variación en la componente uniforme de la velocidad inducida debido a los momentos (y otras componentes no uniformes de la tracción) del rotor.

Para tener en cuenta dichos efectos utilizamos el modelo de estela de Pitt-Peters. A continuación se presentan las ecuaciones no estacionarias y no lineales del modelo y más adelante se simplifican restringiendo la no linealidad a la componente uniforme y eliminando los términos no estacionarios.

El modelo nos relaciona la velocidad inducida con los momentos aerodinámicos mediante la ecuación diferencial:

$$M \begin{Bmatrix} \lambda_0 \\ \lambda_{1sw} \\ \lambda_{1cw} \end{Bmatrix}^* + VL^{-1} \begin{Bmatrix} \lambda_0 \\ \lambda_{1sw} \\ \lambda_{1cw} \end{Bmatrix} = \begin{Bmatrix} c_T \\ -c_L \\ -c_M \end{Bmatrix}_a$$

Donde el subíndice a indica que sólo se consideran fuerzas aerodinámicas. Las matrices anteriores valen:

$$M = \begin{bmatrix} \frac{128}{75\pi} & 0 & 0 \\ 0 & \frac{16}{45\pi} & 0 \\ 0 & 0 & \frac{16}{45\pi} \end{bmatrix}$$

$$L = \begin{bmatrix} \frac{1}{2} & 0 & -\frac{15\pi}{64}X \\ 0 & 2(1+X^2) & 0 \\ \frac{15\pi}{64}X & 0 & 2(1-X^2) \end{bmatrix}$$

$$V = \begin{bmatrix} V_T & 0 & \\ 0 & \bar{V} & 0 \\ 0 & 0 & \bar{V} \end{bmatrix}$$

X es sólo función del ángulo que la estela forma con el eje del rotor, y vale:

$$X = \tan\left(\frac{\chi}{2}\right)$$

V_T es la velocidad adimensional en el rotor, como estamos utilizando la TCMM es entonces:

$$V_T = \sqrt{\frac{1}{k_\nu^2}\mu^2 + \left(\frac{1}{k_\nu^2} - \frac{1}{k_i^2}\right)\mu_z^2 + \frac{1}{k_i^2}\left(\mu_z - \lambda_0\right)^2}$$

y $\bar{V}$ es una velocidad corregida:

$$\bar{V} = V_T + \lambda_0 \frac{\partial V_T}{\partial \lambda_0} = V_T\left[1 - \frac{(\mu_z - \lambda_0)\lambda_0}{k_i^2 V_T^2}\right]$$

Los coeficientes de tracción y momentos son:

$$\begin{aligned}
c_T &= \frac{1}{\rho\left(\Omega R\right)^2 \pi R^2} \sum_{i=1}^{N_b} \int_0^R L_i dr \\
c_L &= \frac{1}{\rho\left(\Omega R\right)^2 \pi R^3} \sum_{i=1}^{N_b} \int_0^R L_i(-r\sin\psi_i) dr \\
c_M &= \frac{1}{\rho\left(\Omega R\right)^2 \pi R^3} \sum_{i=1}^{N_b} \int_0^R L_i(-r\cos\psi_i) dr
\end{aligned}$$

Sustituyendo en las anteriores expresiones de los coeficientes la sustentación dada por la TCP, tenemos que el lado derecho del modelo de la estela queda:

$$\left\{ \begin{array}{c} c_T \\ -c_L \\ -c_M \end{array} \right\} = \frac{1}{N_b}\frac{a_0 s}{2} \sum_{i=1}^{N_b} \left\{ \begin{array}{c} F_i \\ 2M_i \sin\psi_i \\ 2M_I \cos\psi_i \end{array} \right\}$$

La anterior ecuación queda para cuatro palas del siguiente modo, donde se han agrupado las magnitudes en la forma vectorial usual (ver 3.1 y 3.2):

$$\left\{ \begin{array}{c} c_T \\ -c_L \\ -c_M \end{array} \right\} = \frac{a_0 s}{4}\Big(N_\theta \vec{\theta} + N_\lambda \vec{\lambda} + N_\omega \vec{\omega} + \vec{N}_{\lambda_0}(\mu_z - \lambda_0) + N_{\beta_M}\vec{\beta}_M + N_{\beta_M^*}\vec{\beta}_M^*\Big)$$

Y las anteriores matrices son:

$$N_\theta = \begin{bmatrix} \frac{2}{3}+\mu^2 & \frac{1}{2}(1+\mu^2) & \mu & 0 \\ \frac{2}{3}\mu & \frac{\mu}{2} & \frac{1}{4}\left(1+\frac{3}{2}\mu^2+\frac{1}{2}\cos 4\bar{t}\right) & -\frac{\mu^2}{8}\sin 4\bar{t} \\ 0 & 0 & -\frac{1}{8}\sin 4\bar{t} & \frac{1}{4}\left(1+\frac{\mu^2}{2}-\frac{1}{2}\cos 4\bar{t}\right) \end{bmatrix}$$

$$N_\lambda = -N_\omega = \begin{bmatrix} -\frac{\mu}{2} & 0 \\ -\frac{1}{4} & 0 \\ 0 & -\frac{1}{4} \end{bmatrix}$$

$$N_{\lambda_0} = \begin{Bmatrix} 1 \\ \frac{\mu}{2} \\ 0 \end{Bmatrix}$$

$$N_{\beta_M} = \begin{bmatrix} 0 & \mu^2\sin 2\bar{t} & -\mu\cos 2\bar{t} & -\mu\sin 2\bar{t} \\ 0 & \frac{\mu}{3}\sin 2\bar{t} & \frac{1}{4}\left(1-\frac{\mu^2}{2}+\frac{\mu^2}{2}\cos 4\bar{t}\right) & \frac{\mu^2}{8}\sin 4\bar{t} \\ -\frac{\mu}{3} & \frac{\mu}{3}\cos 2\bar{t} & -\frac{\mu^2}{8}\sin 4\bar{t} & -\frac{1}{4}\left(1+\frac{\mu^2}{2}-\frac{\mu^2}{2}\cos 4\bar{t}\right) \end{bmatrix}$$

$$N_{\beta_M^*} = \begin{bmatrix} -\frac{2}{3} & 0 & -\frac{\mu}{2}\sin 2\bar{t} & \frac{\mu}{2}\cos 2\bar{t} \\ -\frac{\mu}{3} & -\frac{\mu}{3}\cos 2\bar{t} & 0 & -\frac{1}{4} \\ 0 & \frac{\mu}{3}\sin 2\bar{t} & -\frac{1}{4} & 0 \end{bmatrix}$$

3.3.5. Ecuación de la velocidad inducida

A continuación simplificamos el modelo de estela de forma análoga a como hicimos con el batimiento: despreciando los términos que no sean la solución estacionaria. Además realizamos una simplificación adicional, que es despreciar el efecto que c_L y c_M tienen sobre la velocidad inducida uniforme λ_0 y también despreciamos el efecto que λ_{1sw} tiene sobre c_T. De esta forma pasamos de tener un sistema de 3 ecuaciones no lineales a tener una ecuación no lineal en λ_0 y una vez obtenida ésta, dos ecuaciones lineales para λ_{1cw} y λ_{1sw}.

Tras aplicar las anteriores simplificaciones, la primera ecuación de la estela nos da el resultado clásico de la TCM(M):

$$\lambda_0 = \frac{c_T}{2V_T}$$

Para incluir el efecto suelo hay que utilizar la velocidad inducida modificada por el efecto suelo en la TEP. El coeficiente de sustentación se encuentra calculado en función de la velocidad teniendo en cuenta el efecto suelo, por lo que lo reescribimos y ponemos en función de la velocidad inducida sin efecto suelo. Para ello aplicamos la relación que obtuvimos para el efecto suelo

$$\mu_z - \lambda_{0G} = (\mu_z - \lambda_0)\, K_{OG}(\lambda_0)$$

Donde se ha llamado:

$$K_{0G}(\lambda_0) = 1 - \frac{1}{16\left(\frac{z}{R}\right)^2}\frac{1}{1+\left(\frac{\mu}{\lambda_0}\right)^2}$$

El coeficiente de sustentación es por tanto:

$$c_T(\lambda_0) = \frac{as}{2}\left\{\theta_0\left(\frac{1}{3}+\frac{\mu^2}{2}\right)+\frac{\mu}{2}\left(\theta_{1sw}+\frac{\bar{p}_w}{2}\right)+K_{0G}(\lambda_0)\frac{\mu_z-\lambda_0}{2}+\frac{1}{4}\left(1+\mu^2\right)\theta_t\right\}$$

Ya podemos resolver la ecuación no lineal que tenemos para λ_0, para ello utilizamos el método de Newton o de la Secante; con el resultado ya podemos calcular λ_{0G} y K_{0G} y tenemos el siguiente par de ecuaciones lineales para λ_{1cw} y λ_{1sw}:

$$A_{\lambda\beta}\vec{\beta} + A_{\lambda\lambda}\vec{\lambda} = A_{\lambda\theta}\vec{\theta} + A_{\lambda\omega}\vec{\omega} + A_{\lambda\dot{\omega}}\dot{\vec{\omega}} + \vec{A}_{\lambda\lambda_0}\left(\mu_z - \lambda_0\right)$$

Donde las anteriores matrices tienen de componentes:

$$A_{\lambda\beta} = -\frac{a_0 s}{4} K N_{\beta 0} \qquad A_{\lambda\lambda} = I - \frac{a_0 s}{4} K N_{\lambda 0}$$
$$A_{\lambda\theta} = \frac{a_0 s}{4} K N_{\theta 0} \qquad A_{\lambda\lambda_0} = \frac{a_0 s}{4} K N_{\lambda_0 0}$$
$$A_{\lambda\omega} = \frac{a_0 s}{4} K N_{\omega 0} \qquad A_{\lambda\omega^*} = \frac{a_0 s}{4} K N_{\omega^* 0}$$

Y el subíndice 0 indica que sólo se han retenido términos estacionarios (y se ha eliminado el grado de libertad β_d, al igual que para la ecuación de batimiento):

$$N_{\theta 0} = \begin{bmatrix} \frac{2}{3}+\mu^2 & \frac{1}{2}\left(1+\mu^2\right) & \mu & 0 \\ \frac{2}{3}\mu & \frac{\mu}{2} & \frac{1}{4}\left(1+\frac{3}{2}\mu^2\right) & 0 \\ 0 & 0 & 0 & \frac{1}{4}\left(1+\frac{\mu^2}{2}\right) \end{bmatrix}$$
$$N_{\beta 0} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & \frac{1}{4}\left(1-\frac{\mu^2}{2}\right) & 0 \\ -\frac{\mu}{3} & 0 & -\frac{1}{4}\left(1+\frac{\mu^2}{2}\right) \end{bmatrix}$$
$$N_{\lambda 0} = -N_{\omega_0} = \begin{bmatrix} -\frac{\mu}{2} & 0 \\ -\frac{1}{4} & 0 \\ 0 & -\frac{1}{4} \end{bmatrix}$$
$$N_{\lambda_0 0} = \begin{Bmatrix} 1 \\ \frac{\mu}{2} \\ 0 \end{Bmatrix}$$

K es simplemente la matriz que resulta de eliminar la primera fila al producto de L por V^{-1}:

$$K = \begin{bmatrix} 0 & \frac{2(1+X^2)}{V} & 0 \\ \frac{15\pi}{64 V_T} X & 0 & \frac{2(1-X^2)}{V} \end{bmatrix}$$

3.3.6. Esquema de cálculo del rotor

A continuación agrupamos los resultado que se han obtenido en toda esta sección y detallamos el cálculo de las fuerzas sobre el rotor:

1. Calculamos la velocidad inducida media en el rotor sin efecto suelo λ_0 resolviendo numéricamente la siguiente ecuación no lineal:

$$\frac{\lambda_0}{k_i} = \frac{c_T}{2V_T}$$

2. Con λ_0 calculado utilizamos K_{G0} para calcular la velocidad inducida con efecto suelo.

3. Resolvemos las 5 ecuaciones lineales que nos determinan el batimiento y las componentes no uniformes de la velocidad inducida:

$$\begin{bmatrix} A_{\beta\beta} & A_{\beta\lambda} \\ A_{\lambda\beta} & A_{\lambda\lambda} \end{bmatrix} \begin{Bmatrix} \vec{\beta} \\ \vec{\lambda} \end{Bmatrix} = \begin{bmatrix} A_{\beta\theta} \\ A_{\lambda\theta} \end{bmatrix} \vec{\theta} + \begin{bmatrix} A_{\beta\omega} \\ A_{\lambda\omega} \end{bmatrix} \vec{\omega} + \begin{Bmatrix} A_{\beta\lambda_0} \\ A_{\lambda\lambda_0} \end{Bmatrix} (\mu_z - \lambda_0) + \begin{bmatrix} A_{\beta\dot{\omega}} \\ A_{\lambda\dot{\omega}} \end{bmatrix} \dot{\vec{\omega}}$$

   Hay que tener en cuenta que los resultados no sólo dependen del vector de estado del helicóptero sino también de su derivada ($\dot{p}_w$ y $\dot{q}_w$) por lo que invirtiendo la matriz del lado izquierdo de las ecuaciones tenemos unos resultados de la forma:

$$\vec{\beta} = \vec{\beta}_0 + \beta_1 \begin{Bmatrix} \dot{p}_w \\ \dot{q}_w \end{Bmatrix}$$
$$\vec{\lambda} = \vec{\lambda}_0 + \lambda_1 \begin{Bmatrix} \dot{p}_w \\ \dot{q}_w \end{Bmatrix}$$

   Donde β_1 y λ_1 son matrices de 3x2 y 2x2 respectivamente. Las fuerzas en el plano del rotor y el par del rotor dependen únicamente de términos de segundo orden por lo que no necesitamos para ellas los términos dependientes en la aceleración angular ya que los despreciamos. La sustentación del rotor depende de los términos linealmente pero no del batimiento ni de las no uniformes de la velocidad inducida así que tampoco nos preocupan los términos en aceleración angular. Finalmente los momentos de balance y cabeceo dependen linealmente del batimiento, recordamos que:

$$L_{hw} = -\frac{N_b}{2} K_\beta \beta_{1sw} - \frac{Q_R}{2} \beta_{1cw}$$
$$M_{hw} = -\frac{N_b}{2} K_\beta \beta_{1cw} + \frac{Q_R}{2} \beta_{1sw}$$

   por lo que los momentos contribuirán con un término de amortiguamiento al movimiento del helicóptero, que habrá que pasar al lado izquierdo de la ecuación diferencial.

4. Calculamos las fuerzas y momentos del rotor y los añadimos a la ecuaciones del helicóptero utilizando las fórmulas de las sección 3.2.4.

5. Transportamos las fuerzas y momentos al centro de masas y obtenemos las fuerzas y momentos del rotor sobre el centro de masas en la forma:

$$\vec{F}_R = \vec{F}_{R_0} + F_{R_1} \dot{\vec{x}}$$
$$\vec{M}_R = \vec{M}_{R_0} + M_{R_1} \dot{\vec{x}}$$

3.4. Fuselaje

3.4.1. Ecuaciones del sólido rígido

Modelamos el fuselaje como un sólido rígido, sobre el que actúan las fuerzas del resto de subsistemas. Las ecuaciones de movimiento del sólido rígido son:

1. Momento lineal
$$\vec{F} = m\frac{d\vec{v}}{dt}$$
2. Momento angular
$$\vec{M} = \frac{d(I\vec{\omega})}{dt}$$

Donde $\vec{v}$ es la velocidad del centro de masas respecto a un sistema inercial, $\vec{\omega}$ es la velocidad angular de unos ejes ligados al sólido respecto a los ejes inerciales y I es la matriz de inercia del sólido, que de ahora en adelante supondremos tiene el plano xz de simetría por lo que la matriz de inercia se simplifica:

$$I = \begin{bmatrix} I_x & 0 & -J_{xz} \\ 0 & I_y 0 & \\ -J_{xz} & 0 & I_z \end{bmatrix}$$

Podemos expresar las anteriores ecuaciones expresando los vectores en ejes ligados al cuerpo, en cuyo caso queda:

$$\vec{F} = m\left(\frac{\partial\vec{v}}{\partial t} + \vec{\omega}\wedge\vec{v}\right)$$
$$\vec{M} = I\frac{\partial\vec{\omega}}{\partial t} + \vec{\omega}\wedge(I\vec{\omega})$$

Desarrollando las anteriores ecuaciones en componentes:

$$\begin{aligned} X &= m(\dot{u} - rv + qw) \\ Y &= m(\dot{v} + ru - pw) \\ Z &= m(\dot{w} - qu + pv) \\ L &= I_x\dot{p} - J_{xz}\dot{r} + (I_z - I_y)qr - J_{xz}pq \\ M &= I_y\dot{q} - (I_z - I_x)pr + J_{xz}(p^2 - r^2) \\ N &= I_z\dot{r} - J_{xz}\dot{p} - (I_x - I_y)pq + J_{xz}qr \end{aligned}$$

La posición del centro de masas en cada instante se obtiene de las ecuaciones cinemáticas:

$$\begin{Bmatrix} \dot{x_I} \\ y_I \\ z_I \end{Bmatrix} = L_{IB}\begin{Bmatrix} u \\ v \\ w \end{Bmatrix}$$

Finalmente complementamos las ecuaciones cinemáticas con las ecuaciones para la variación de las componentes del cuaternio (ver ecuación 2.2)

3.4.2. Fuerzas aerodinámicas del fuselaje

Debe ser posible calcular para cada ángulo de ataque y de resbalamiento las fuerzas aerodinámicas que actúan sobre el fuselaje. Las fuerzas y momentos sobre el fuselaje son:

$$\begin{aligned} X_f =& \frac{1}{2}\rho V^2 S_p c_{x_f} \\ Y_f =& \frac{1}{2}\rho V^2 S_s c_{y_f} \\ Z_f =& \frac{1}{2}\rho V^2 S_p c_{z_f} \\ L_f =& \frac{1}{2}\rho V^2 S_s l_f c_{l_f} \\ M_f =& \frac{1}{2}\rho V^2 S_p l_f c_{m_f} \\ N_f =& \frac{1}{2}\rho V^2 S_s l_f c_{n_f} \end{aligned}$$

Donde S_s y S_p son una superficie lateral y frontal respectivamente del helicóptero y l_f una distancia. Como en general los datos aerodinámicos se encuentran dados en ejes viento, los anteriores coeficientes se obtienen de la forma:

$$\begin{aligned} \left\{ \begin{array}{c} c_x \\ c_y \\ c_z \end{array} \right\} =& L_{Bw_f} \left\{ \begin{array}{c} -c_D \\ c_Y \\ -c_L \end{array} \right\} \\ \left\{ \begin{array}{c} c_l \\ c_m \\ c_n \end{array} \right\} =& L_{Bw_f} \left\{ \begin{array}{c} c_l \\ c_m \\ c_n \end{array} \right\} \end{aligned}$$

Donde la matriz de cambio de ejes viento del fuselaje a ejes cuerpo es:

$$L_{Bw_f} = \left[\begin{array}{ccc} \cos\alpha\cos\beta & -\cos\alpha\sin\beta & -\sin\alpha \\ \sin\beta & \cos\beta & 0 \\ \sin\alpha\cos\beta & -\sin\alpha\sin\beta & \cos\alpha \end{array} \right]$$

Necesitamos por tanto dar para todo ángulo de ataque α y de resbalamiento β el valor de los coeficientes c_D, c_L, c_Y, c_l, c_m y c_n. Como el fuselaje presenta una forma muy compleja no existe un método fiable de calcular los anteriores coeficientes por lo que generalmente se recurre a ensayos realizados en tuneles aerodinámicos. Sin embargo, es posible que sólo se dispongan de datos para ángulos pequeños por lo que si es así se utilizan las siguientes aproximaciones,

tal como vienen en [3]:

$$\begin{aligned}
c_{D_f} &= c_{D_{f_\alpha}} + c_{D_{f_\beta}} \\
c_{D_{f_\alpha}} &= c_{D_{f_{\alpha=90^o}}} |\sin\alpha| \sin^2\alpha \\
c_{D_{f_\beta}} &= c_{D_{f_{\beta=90^o}}} |\sin\beta| \sin^2\beta \\
c_{L_f} &= c_{D_{f_{\alpha=90^o}}} |\sin\alpha| \sin\alpha\cos\alpha \\
c_{m_f} &= c_{m_{f_{\alpha=90^0}}} |\sin\alpha| \sin\alpha \\
c_{Y_f} &= -c_{D_{f_{\beta=90^o}}} |\sin\beta| \sin\beta\cos\beta \\
c_{l_f} &= c_{l_{f_{\beta=90^o}}} |\sin\beta| \sin\beta \\
c_{n_f} &= c_{n_{f_{\beta=90^o}}} |\sin\beta| \sin\beta
\end{aligned}$$

Si los datos para ángulos pequeños se encuentran dados en forma de tabla se interpolan mediante splines cúbicas, y para asegurar una transición suave de ángulos pequeños a grandes se interpola mediante una spline cúbica que va desde el último punto para ángulos pequeños a otros dos puntos a 45º y 50º respectivamente, calculados mediante las anteriores fórmulas. A partir del punto situado a 50º del ángulo pequeño se utilizan las anteriores fórmulas.

3.5. Rotor de cola

Modelamos el rotor de cola de forma similar al rotor principal. Para ello utilizamos los ejes que se describieron en el capítulo dedicado a sistemas de referencia.

3.5.1. Acoplamiento entre batimiento y paso

Al contrario que en el rotor principal, el eje de batimiento de la pala no se encuentra perpendicular a la pala, sino que se encuentra girado un ángulo δ_3 (figura 3.4).

Como consecuencia, al moverse la pala, su eje describe un cono de semiángulo $\frac{\pi}{2} - \delta_3$. Para calcular los ejes transformados de la pala calculamos el efecto de una rotación α alrededor del eje de dicho cono, el cuaternio de rotación en ejes rotación será entonces:

$$q = \left[\cos\frac{\alpha}{2}, \sin\frac{\alpha}{2}\vec{u}\right]$$

Donde $\vec{u}$ es el vector unitario alrededor del cual rota la pala:

$$\vec{u} = \sin\delta_3\vec{i}_1 + \cos\delta_3\vec{j}_1$$

A partir de dicho cuaternio es posible encontrar los ángulos de batimiento β, paso θ y arrastre ζ de la pala de modo similar a como se realizó con los ángulos de Euler. En la figura 3.5 se muestran los giros que transforman de ejes de rotación a ejes pala.

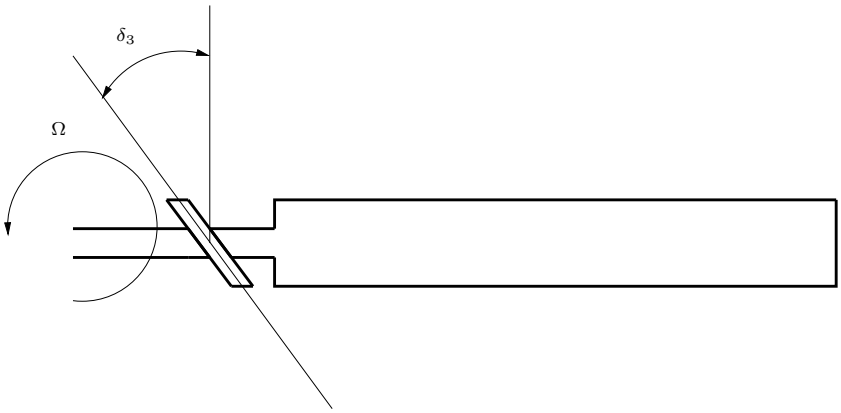


Figura 3.4: Ángulo δ_3

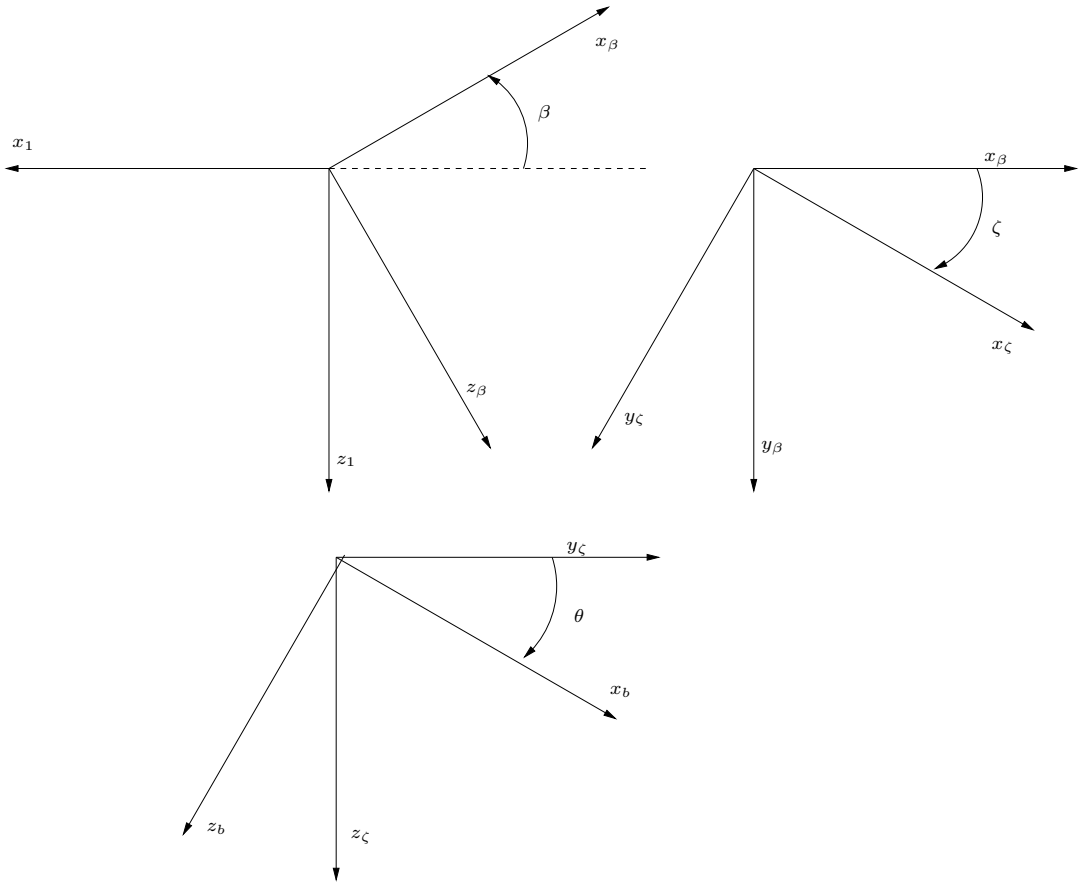


Figura 3.5: Ángulos de batimiento, arrastre y paso

Para ángulos pequeños la matriz de cambio de base es entonces:

$$\left\{ \begin{array}{c} \vec{i}_b \\ \vec{j}_b \\ \vec{k}_b \end{array} \right\} = \left[\begin{array}{ccc} 1 & \zeta & -\beta \\ -\zeta & 1 & \theta \\ \beta & -\theta & 1 \end{array} \right] \left\{ \begin{array}{c} -\vec{i}_1 \\ -\vec{j}_1 \\ \vec{k}_1 \end{array} \right\}$$

Y para ángulos pequeños de α la matriz de cambio de base (transpuesta de la matriz de rotación), aplicando la fórmula que se obtuvo para el cambio de base de ejes cuerpo a inerciales es:

$$\left\{ \begin{array}{c} \vec{i}_b \\ \vec{j}_b \\ \vec{k}_b \end{array} \right\} = \left[\begin{array}{ccc} 1 & 0 & -\alpha\cos\delta_3 \\ 0 & 1 & \alpha\sin\delta_3 \\ \alpha\cos\delta_3 & -\alpha\sin\delta_3 & 1 \end{array} \right] \left\{ \begin{array}{c} -\vec{i}_1 \\ -\vec{j}_1 \\ \vec{k}_1 \end{array} \right\}$$

Identificando términos y despejando α podemos obtener el paso provocado por el batimiento de la pala:

$$\theta = \beta \tan\delta_3$$

A partir de ahora llamamos $k_3 = \tan\delta_3$.

El paso total en las palas del rotor de cola será el debido a los controles más el debido al batimiento:

$$\begin{aligned} \theta^*_{0_T} &= \theta_{0_T} + k_3\beta_{0_T} \\ \theta^*_{1s_T} &= k_3\beta_{1s_T} \\ \theta^*_{1c_T} &= k_3\beta_{1c_T} \end{aligned}$$

Donde hemos supuesto que el único control que hace el piloto sobre el rotor de cola es sobre el paso colectivo.

3.5.2. Ecuación de batimiento

Aprovechamos las relaciones que se obtuvieron para el rotor principal despreciando el efecto de la velocidad angular y las componentes no uniformes de la velocidad inducida y reemplazamos el paso por el paso total debido al batimiento. Obtenemos el siguiente sistema de ecuaciones:

$$A_{\beta\beta_T} \left\{ \begin{array}{c} \beta_{0_T} \\ \beta_{1cw_T} \\ \beta_{1sw_T} \end{array} \right\} = \vec{A}_{\beta\theta_{0_T}}\theta_{0_T} + \vec{A}_{\beta\lambda_{0_T}}(\mu_{z_T} - \lambda_{0_T}) + \vec{A}_{\beta\theta_{t_T}}\theta_{t_T}$$

Donde las componentes de las anteriores matrices y vectores son:

$$
\begin{aligned}
A^{11}_{\beta\beta_T} &= \left(\frac{8\lambda_\beta^2}{\gamma}\right)_T - k_3(1+\mu_T^2)\\
A^{12}_{\beta\beta_T} &= 0\\
A^{13}_{\beta\beta_T} &= -k_3\frac{4}{3}\mu_T\\
A^{21}_{\beta\beta_T} &= \frac{4\mu_T}{3}\\
A^{22}_{\beta\beta_T} &= \left[\frac{8(\lambda_\beta^2-1)}{\gamma}\right]_T - k_3\left(1+\frac{\mu_T^2}{2}\right)\\
A^{23}_{\beta\beta_T} &= 1+\frac{\mu_T^2}{2}\\
A^{31}_{\beta\beta_T} &= -k_3\frac{8}{3}\mu_T\\
A^{32}_{\beta\beta_T} &= -1+\frac{\mu_T^2}{2}\\
A^{33}_{\beta\beta_T} &= \left[\frac{8(\lambda_\beta^2-1)}{\gamma}\right]_T - k_3\left(1+\frac{3}{2}\mu_T^2\right)\\
A^{1}_{\beta\theta_{0_T}} &= 1+\mu_T^2\\
A^{2}_{\beta\theta_{0_T}} &= 0\\
A^{3}_{\beta\theta_{0_T}} &= \frac{8}{3}\mu_T\\
A^{1}_{\beta\lambda_{0_T}} &= \frac{4}{3}\\
A^{2}_{\beta\lambda_{0_T}} &= 0\\
A^{3}_{\beta\lambda_{0_T}} &= 2\mu_T\\
A^{1}_{\beta\theta_{t_T}} &= 4\left(\frac{1}{5}+\frac{\mu^2}{6}\right)\\
A^{2}_{\beta\theta_{t_T}} &= 0\\
A^{3}_{\beta\theta_{t_T}} &= 2\mu
\end{aligned}
$$

Invirtiendo la matriz $A_{\beta\beta_T}$ obtenemos el batimiento en función de la velocidad inducida, y del batimiento obtenemos los ángulos de paso efectivos, en función de la velocidad inducida.

3.5.3. Ecuación de la velocidad inducida

De la misma forma que para el rotor principal, pero sin la necesidad de tener en cuenta el efecto suelo llegamos a la siguiente ecuación que resulta de aplicar la TCMM y la teoría del elemento pala al rotor de cola:

$$\frac{\lambda_{0_T}}{k_i} = \frac{c_{T_T}}{2\sqrt{V_T}}$$
$$c_{T_T}(\lambda_{0_T}) = \frac{a_{0_T} s_T}{2}\left\{\theta^*_{0_T}\left(\frac{1}{3}+\frac{\mu_T^2}{2}\right)+\frac{\mu_T}{2}\theta^*_{1s_T}+\frac{\mu_{z_T}-\lambda_{0_T}}{2}\right\}$$
$$V_T = \frac{1}{k_\nu^2}\mu_T^2 + \left(\frac{1}{k_\nu^2}-\frac{1}{k_i^2}\right)\mu_{z_T}^2 + \left(\frac{\mu_{z_T}-\lambda_{0_T}}{k_i}\right)^2$$

La anterior ecuación es no lineal y se resuelve al igual que para el rotor principal mediante el método de Newton. Una vez conocida la velocidad inducida podemos calcular el resto de las magnitudes, como el coeficiente de sustentación de la cola c_{T_T} y el batimiento β_{1cw_T} y β_{1sw_T}.

3.5.4. Cálculo de las fuerzas y momentos

De la misma forma que para el rotor principal calculamos el par:

$$\left(\frac{2c_{Q_T}}{a_{0_T} s_T}\right) = -(\mu_{z_T}-\lambda_{0_T})\left(\frac{2c_{T_T}}{a_{0_T} s_T}\right)+\frac{\delta_T}{4a_{0_T}}(1+3\mu_T^2)$$

De nuevo δ_T es un coeficiente de resistencia medio:

$$\delta_T = \delta_{T_0} + \delta_{T_2} c_{T_T}^2$$

Al contrario que para el rotor principal no retenemos los términos de segundo orden, por lo que en primera aproximación se puede suponer que la tracción del rotor de cola actúa perpendicular al plano de batimiento. Además el único momento que consideramos del rotor de cola es el Q_T. Pasando entonces de ejes rotor de cola-viento a rotor de cola, y de estos a ejes cuerpo llevando las resultantes al centro de masas del helicóptero, queda:

$$\begin{aligned}
X_T &= T_T \beta_{1c_T}\\
Y_T &= T_T\\
Z_T &= -T_T \beta_{1s_T}\\
L_T &= h_T Y_T\\
M_T &= (l_T + x_{cg}) Z_T - Q_T\\
N_T &= -(l_T + x_{cg}) Y_T
\end{aligned}$$

3.6. Estabilizador horizontal

Calculamos la velocidad en ejes cuerpo del estabilizador horizontal:

$$\begin{aligned}
u_{rw_{tp}} &= u_{rw} - q h_{tp}\\
v_{rw_{tp}} &= v_{rw} - r(l_{tp} + x_{cg}) + p h_{tp}\\
w_{rw_{tp}} &= w_{rw} + q(l_{tp} + x_{cg})
\end{aligned}$$

y de esa velocidad el ángulo de ataque y resbalamiento:

$$\alpha_{tp} = \arctan \frac{w_{rw_{tp}}}{u_{rw_{tp}}}$$
$$\beta_{tp} = \arctan \frac{v_{rw_{tp}}}{u_{rw_{tp}}}$$

Finalmente, suponemos conocido para todo ángulo de ataque y resbalamiento los coeficientes de resistencia y sustentación, con lo que podemos obtener las fuerzas sobre el estabilizador horizontal:

$$\begin{aligned} c_{X_{tp}} &= c_{L_{tp}} \sin\alpha - c_{D_{tp}} \cos\alpha \\ c_{Z_{tp}} &= -c_{D_{tp}} \sin\alpha - c_{L_{tp}} \cos\alpha \\ V_{tp}^2 &= u_{rw_{tp}}^2 + v_{rw_{tp}}^2 + w_{rw_{tp}}^2 \\ X_{tp} &= \frac{1}{2}\rho V_{tp}^2 S_{tp} c_{X_{tp}} \\ Z_{tp} &= \frac{1}{2}\rho V_{tp}^2 S_{tp} c_{Z_{tp}} \end{aligned}$$

La única dificultad estriba, por tanto, en conocer los coeficientes aerodinámicos. Para estimarlos utilizamos el método que viene descrito en el [3]:

1. Se pasa al programa la pendiente de sustentación a. Si no es así el programa realiza una estimación a partir del alargamiento del estabilizador:

$$a = \frac{2\pi}{1 + \frac{2}{AR}}$$

2. Se pasa al programa el máximo coeficiente de sustentación $c_{L_{max}}$. Si no es así se supone que se produce la entrada en pérdida para resbalamiento nulo al llegar a $\frac{\pi}{4}$.

3. Se calcula para el actual ángulo de resbalamiento la pendiente efectiva de la sutentación.

$$a_{ef} = a \cos^2(\beta + \Delta)$$

   Si se pasó al programa la pendiente de sustentación calculada teniendo en cuenta el efecto de la flecha Δ entonces deberá introducirse $\Delta = 0$ aunque el estabilizador tenga flecha no nula para evitar tenerlo en cuenta dos veces.

4. Con la pendiente efectiva se calcula el ángulo de entrada en pérdida:

$$\alpha_s = \frac{c_{L_{max}}}{a_{ef}}$$

5. Se define el siguiente ángulo auxiliar:

$$\alpha_1 = 1{,}2\alpha_s$$

6. Se transporta el ángulo de ataque dentro del entorno que va de 0 a $\frac{\pi}{2}$. Para ello definimos el siguiente ángulo de ataque:

$$\begin{aligned} \alpha_i &= \alpha & 0 \leq \alpha < \frac{\pi}{2} \\ \alpha_i &= -\alpha & -\frac{\pi}{2} \leq \alpha < 0 \\ \alpha_i &= \pi - \alpha & \frac{\pi}{2} \leq \alpha < \pi \\ \alpha_i &= \pi + \alpha & -\pi \leq \alpha < \frac{\pi}{2} \end{aligned}$$

7. Se calcula un coeficiente de sustentación:

$$\begin{aligned} c_{L_0} &= a_{ef}\alpha_i & 0 < \alpha_i < \alpha_s \\ c_{L_0} &= c_{L_{max}} - a_{ef}(\alpha_i - \alpha_s) & \alpha_s \leq \alpha_i < \alpha_1 \\ c_{L_0} &= 0{,}8c_{L_{max}}\left[1 - \left(\frac{\alpha_i - \alpha_1}{\frac{\pi}{2} - \alpha_1}\right)^2\right] & \alpha_1 \leq \alpha_i \leq \frac{\pi}{2} \end{aligned}$$

8. Dependiendo del cuadrante, obtenemos el coeficiente de sustentación final:

$$\begin{aligned} c_L &= 0{,}8c_{L_0} & -\pi \leq \alpha \leq -\frac{\pi}{2} \\ c_L &= -c_{L_0} & -\frac{\pi}{2} < \alpha < 0 \\ c_L &= c_{L_0} & 0 \leq \alpha < \frac{\pi}{2} \\ c_L &= -0{,}8c_{L_0} & \frac{\pi}{2} \leq \alpha \leq \pi \end{aligned}$$

9. Calculamos la componente no inducida del coeficiente de resistencia, c_{D_p}. Como es frecuente que la polar de resistencia del perfil que nos den sea no simétrica, se contempla esa posibilidad y modificamos ligeramente el memorandum definiendo un nuevo ángulo:

$$\begin{aligned} \alpha_{iD} &= \pi + \alpha & -\pi \leq \alpha \leq -\frac{\pi}{2} \\ \alpha_{iD} &= -\alpha & -\frac{\pi}{2} < \alpha < -0{,}60 \\ \alpha_{iD} &= \alpha & -0{,}60 \leq \alpha < \frac{\pi}{2} \\ \alpha_{iD} &= \pi - \alpha & \frac{\pi}{2} \leq \alpha \leq \pi \end{aligned}$$

10. Calculamos c_{D_p}:

$$\begin{aligned} c_{D_p} &= i1(\alpha_{iD}) & -0{,}60 \leq \alpha_{iD} < -0{,}35 \\ c_{D_p} &= \delta_0 + \delta_1\alpha + \delta_2\alpha^2 & -0{,}35 \leq \alpha_{iD} \leq 0{,}35 \\ c_{D_p} &= i2(\alpha_{iD}) & 0{,}35 < \alpha_{iD} < 0{,}60 \\ c_{D_p} &= -0{,}1254 + 0{,}09415\alpha + 0{,}977525\sin^2\alpha & 0{,}60 \leq \alpha_{iD} \leq \frac{\pi}{2} \end{aligned}$$

Donde $i1$, $i2$ son dos funciones que interpolan los puntos -0.60, -0.55, -0.35 y 0.35, 0.55, 0.60 respectivamente mediante splines cúbicas para asegurar una transición suave de la zona precisa de ángulos pequeños a la aproximada para ángulos grandes. Los coeficientes δ_0, δ_1 y δ_2 son suplidos externamente, en cuyo caso se supone que llevan incluidos la componente de velocidad inducida, por lo que se modifican de acuerdo a ello automáticamente para no incluir su efecto dos veces. En caso de no suplirse ninguno se asumirían los siguientes valores por defecto:

$$\delta_0 = 0{,}009$$
$$\delta_1 = 0.$$
$$\delta_2 = 0{,}11$$

11. La resistencia total, teniendo en cuenta la componente inducida:

$$c_D = c_{D_p} + \frac{c_L^2}{0{,}8\pi AR}$$

3.7. Estabilizador vertical

El cálculo es totalmente análogo al estabilizador horizontal, lo único que varía es la velocidad relativa al viento:

$$\begin{aligned} u_{rw_{fn}} &= u_{rw} - qh_{fn} \\ v_{rw_{fn}} &= v_{rw} - r(l_{fn} + x_{cg}) + h_{fn}p \\ w_{rw_{fn}} &= w_{rw} + q(l_{fn} + x_{cg}) \end{aligned}$$

También cambia la definición de ángulo de ataque y resbalamiento:

$$\begin{aligned} \alpha_{fn} &= \arctan \frac{v_{rw_{fn}}}{u_{rw_{fn}}} \\ \beta_{fn} &= \arctan \frac{w_{rw_{fn}}}{u_{rw_{fn}}} \end{aligned}$$

Finalmente las fuerzas son:

$$\begin{aligned} c_{X_{fn}} &= c_{L_{fn}} \sin\alpha_{fn} - c_{D_{fn}} \cos\alpha_{fn} \\ c_{Y_{fn}} &= -c_{D_{fn}} \sin\alpha_{fn} - c_{L_{fn}} \cos\alpha_{fn} \\ V_{fn}^2 &= u_{rw_{fn}}^2 + v_{rw_{fn}}^2 + w_{rw_{fn}}^2 \\ X_{fn} &= \frac{1}{2}\rho V_{fn}^2 S_{fn} c_{X_{fn}} \\ Y_{fn} &= \frac{1}{2}\rho V_{fn}^2 S_{fn} c_{Y_{fn}} \end{aligned}$$

Donde los coeficientes aerodinámicos se calculan de la misma forma que para el estabilizador horizontal.

3.8. Controles

Las entradas del piloto se encuentran dadas por los siguientes valores normalizados:

- η_{0p}: Colectivo dado por el piloto. Varía entre 0 para colectivo abajo y 1 para colectivo arriba.
- η_{1sp}: Cíclico longitudinal. Varía entre -1 hacia adelante y +1 hacia detrás.
- η_{1cp}: Cíclico lateral. Varía entre entre +1 a la izquierda y -1 a la derecha.
- η_{pp}: Pedal. Varía entre +1 a la izquierda y -1 a la derecha.

El siguiente paso es transformar los colectivos del piloto a través del sistema de control y de la unidad de mezcla. Para modelar el efecto del sistema de control se especifica tres matrices S tal que los controles finales valgan:

$$\begin{Bmatrix} \eta_0 \\ \eta_{1s} \\ \eta_{1c} \\ \eta_p \end{Bmatrix} = S_0 \begin{Bmatrix} \eta_{0p} \\ \eta_{1sp} \\ \eta_{1cp} \\ \eta_{pp} \end{Bmatrix} + S_1 \begin{Bmatrix} \dot{\phi} \\ \dot{\theta} \\ r \end{Bmatrix} + S_2 \begin{Bmatrix} \theta - \theta_0 \\ \phi - \phi_0 \\ \psi - \psi_0 \end{Bmatrix}$$

Y los valores de referencia θ_0, ϕ_0 y ψ_0 pueden ser constantes o funciones respectivamente de los mandos η_{1sp}, η_{1cp} y η_{pp}. De esta forma es posible implementar un sistema de control "Attitude Command/Attitude Hold" tal como viene descrito en [1].

Es posible especificar un limite para el control automático, de forma que la velocidad angular no pueda influir en más de un determinado porcentaje sobre los controles. Para ello en los ficheros de entrada se ajusta la variable L. Las diferencias entre los controles entre el antes y el después de aplicar el sistema de controles es:

$$e_0 = \eta_{0p} - \eta_0 \qquad e_{1s} = 2(\eta_{1sp} - \eta_{1s})$$
$$e_{1c} = 2(\eta_{1cp} - \eta_{1c}) \qquad e_p = 2(\eta_{pp} - \eta_p)$$

Limitamos el efecto de controles donde η se puede substituir por cualquiera de los cuatro controles del helicóptero.

$$\eta = \begin{cases} \eta_p - L & \text{si } e < -L \\ \eta_p + L & \text{si } e > +L \\ \eta & \text{si } -L \leq e \leq +L \end{cases}$$

Finalmente se transforman los controles en los pasos de las palas en ejes rotor a través de la unidad de mezcla, para ello se especifican el vector $\vec{c}_1$ y la matriz c_2 tales que:

$$\begin{Bmatrix} \theta_0 \\ \theta_{1s} \\ \theta_{1c} \\ \theta_{0_T} \end{Bmatrix} = \vec{c}_1 + c_2 \begin{Bmatrix} \eta_0 \\ \eta_{1s} \\ \eta_{1c} \\ \eta_p \end{Bmatrix}$$

3.9. Motor

El motor controla la velocidad angular del rotor para que permanezca aproximadamente constante. El modelo que se encuentra implementado es prácticamente el descrito en [12].

Simplificamos la ley de control que rige la inyección de combustible en función de la velocidad angular, de la forma:

$$\frac{\bar{\omega}_f}{\Omega_i - \Omega} = \frac{K_{e_1}}{1 + \tau_{e_1} s}$$

Donde K_{e_1} es una constante de proporcionalidad y τ_{e_1} de tiempos. La anterior ecuación representa una linealización del sistema de combustible en torno a la velocidad de giro Ω_i que es la velocidad que el sistema motor intenta mantener. Una simplificación adicional consiste en suponer que la inyección de combustible depende únicamente de la velocidad angular.

El comportamiento de la turbina viene dado en función del combustible inyectado, y se simplifica en la forma:

$$\frac{\bar{Q}_1}{\bar{\omega}_f} = K_{e_2} \left(\frac{1 + \tau_{e_2} s}{1 + \tau_{e_3} s} \right)$$

De nuevo K_{e_2} es una constante de proporcionalidad y τ_{e_2} y τ_{e_3} son constantes de tiempos. Q_1 es el par de uno sólo de los motores.

La ecuación de giro del rotor es simplemente aplicando momentos:

$$\dot{\Omega} = \dot{r} + \frac{1}{I_R} \left[nQ_1 - (1 + P)(Q_R + g_T Q_T) \right]$$

Donde I_R representa la inercia del sistema de transmisión y depende de n que es el número de motores activos y P es un factor para tener en cuenta las pérdidas en el sistema de transmisión.

Eliminando el combustible del anterior sistema de ecuaciones y escribiendolo en forma de sistema de ecuaciones diferenciales de primer orden, queda:

$$\begin{aligned} -\dot{r} + \dot{\Omega} &= \frac{1}{I_R} \left[nQ_1 - (1 + P)(Q_R + g_T Q_T) \right] \\ \dot{Q}_1 &= S \\ K_3 \tau_2 \dot{\Omega} + \tau_1 \tau_3 \dot{S} + (\tau_1 + \tau_3) \dot{Q}_1 &= -Q_1 + K_3 (\Omega_i - \Omega) \end{aligned}$$

En las anteriores ecuaciones K_3 y τ_1 se suponen constantes y τ_2 y τ_3 pueden variar con un par adimensionalizado. Para ello se definen 3 potencias del motor:

- W_{to}: potencia máxima de despegue, que es la máxima potencia que puede realizar el motor.
- W_{mc}: potencia máxima continua.
- W_{id}: potencia mínima, que es la potencia que se necesita para tener al motor en funcionamiento.

Para calcular las anteriores potencias en función de la altura definimos una potencia a nivel del mar, es decir $\rho = \rho_0 = 1{,}225$, que llamamos W_{to_0} y W_{mc_0}, y una exponente n de forma que es:

$$W = W_0 \left(\frac{\rho}{\rho_0} \right)^n$$

La densidad la calcula el simulador a partir de la presión y temperatura que el programa externo utilizando la ley para gases perfectos:

$$\rho = \frac{P}{R_a T}$$

donde $R_a = 286{,}9 \frac{J}{KgK}$ es una constante del aire.

En cada instante de vuelo dividimos por la velocidad de giro del rotor las anteriores potencias para tener 3 pares análogos: Q_{to}, Q_{mc}, Q_{id}. Nos aseguramos en todo momento que el par de un motor Q_1 entre dentro del intervalo $[Q_{id}, Q_{to}]$ y definimos el par adimensional como:

$$\bar{Q} = \frac{Q_1}{Q_{mc}}$$

τ_2 y τ_3 quedan entonces definidas por trozos dando sus valores en 3 puntos:

$$\begin{array}{lll} \bar{Q} = \dfrac{Q_{id}}{Q_{mc}} & \tau_2 = a_2 & \tau_3 = a_3 \\ \bar{Q} = 1 & \tau_2 = b_2 & \tau_3 = b_3 \\ \bar{Q} = \dfrac{Q_{to}}{Q_{mc}} & \tau_2 = c_2 & \tau_3 = c_3 \end{array}$$

3.10. Tren de aterrizaje

Suponemos que el helicóptero se encuentra dotado de patines. Para evitar la penetración suelo-patín y simular la fuerza de rozamiento definimos una serie de puntos, por lo menos dos en cada patín y especificamos las propiedades de muelles y amortiguadores que determinan las fuerzas de reacción y un coeficiente de rozamiento que determinará el valor máximo de la fuerza de rozamiento, tal como vemos en la figura 3.6.

El simulador comprueba en cada instante si existe penetración de algún punto de contacto en el suelo. Si en el instante anterior no existía ninguna penetración calcula un punto que llamamos de no deslizamiento que es la proyección sobre el suelo del punto que ha penetrado y a partir de la distancia entre la posición actual del punto y el punto de no deslizamiento y la velocidad de desplazamiento calculamos las fuerzas de reacción. De forma mas precisa: definimos los ejes locales de colisión como aquellos con origen en el punto de no deslizamiento, ejes x e y contenidos en el plano del suelo y arbitrarios y el eje z normal exterior al suelo y definimos un vector $\vec{e}$ de desplazamiento entre el punto de no deslizamiento y el punto de contacto cuyas componentes en ejes locales de colisión son e_s, e_t, e_n y las componentes de la velocidad del punto de contacto en ejes de colisión vienen dadas por vs, vt, vn.

Las fuerzas de reacción valen entonces:

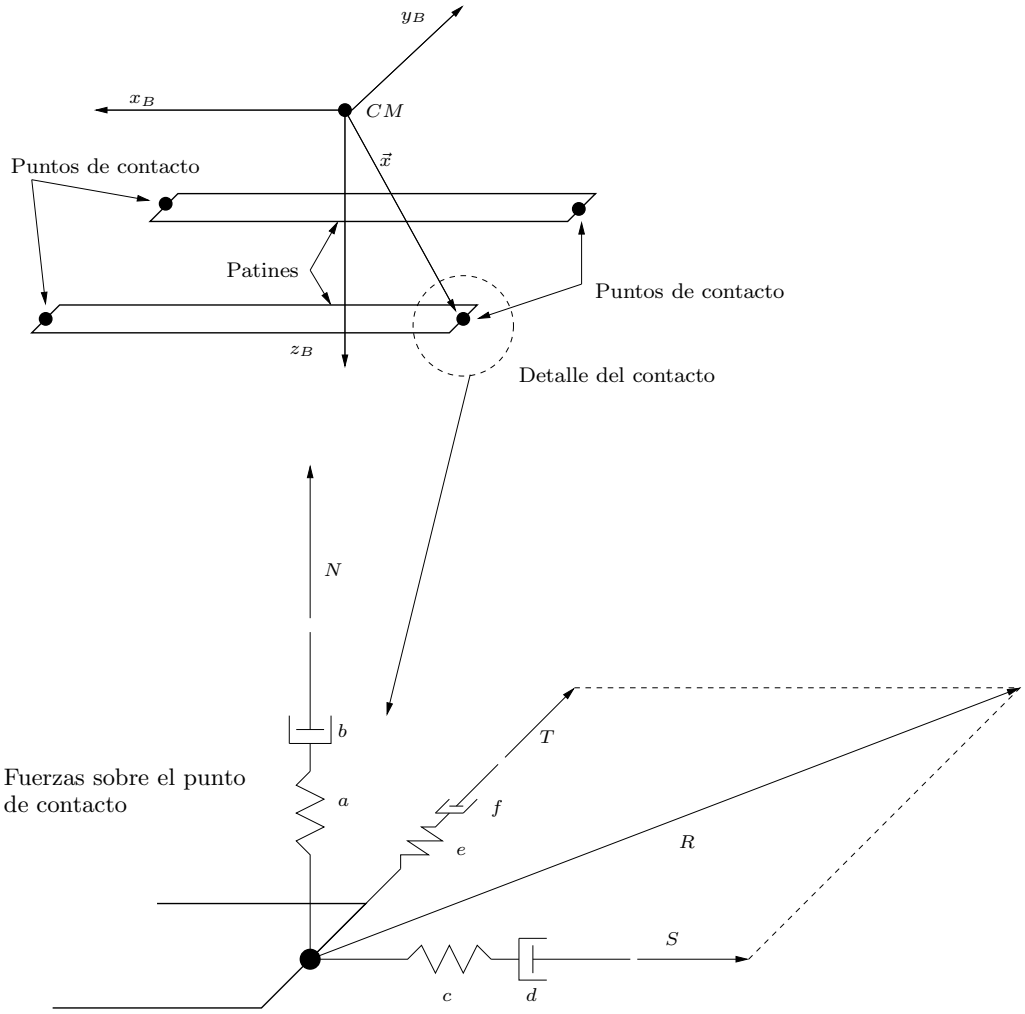


Figura 3.6: Magnitudes de los patines

- Fuerza normal: si hay penetración tiene que ser $e_n < 0$

$$N = N_1 + N_2$$

$$N_1 = -ae_n$$
$$N_2 = \begin{cases} -bv_n & , v_n < 0 \\ 0 & , v_n \geq 0 \end{cases}$$

- Fuerzas de rozamiento:

$$S = -(ce_s + dv_s)$$
$$T = -(ee_t + fv_t)$$

Definimos la fuerza de rozamiento máxima como:

$$R_{max} = nN$$

y el modulo de la fuerza de rozamiento calculada hasta ahora:

$$R = \sqrt{S^2 + T^2}$$

Si es $R < R_{max}$ el cálculo de la fuerza de rozamiento finaliza en este instante, si no entonces se modifican S y T:

$$S = R_{max}\frac{S}{R}$$
$$T = R_{max}\frac{T}{R}$$

y además se modifica el punto de no deslizamiento, ya que se ha producido deslizamiento, y su desplazamiento en ejes colisión viene dado por:

$$k = \frac{R_{max}}{\sqrt{(ce_s)^2 + (ee_t)^2}}$$
$$x : (1-k)e_s$$
$$y : (1-k)e_t$$
$$z : 0$$

Capítulo 4

Integración de las ecuaciones

Una vez que tenemos la ecuación diferencial que nos describe la evolución del tiempo procedemos a integrarla. Para ello aproximamos la solución en un conjunto discreto de instantes de tiempo, lo cual nos lleva a la primera decisión: la elección de dichos instantes. Sabemos que debido a las necesidades de interactividad con el usuario por lo menos necesitamos calcular la solución unas 35 veces por segundo. Dichos instantes de tiempo no se encuentran determinados de antemano, no los podemos elegir, ya que es un programa exterior FlightGear quien nos pide la solución en unos determinados instantes de tiempo. Además debido a numerosos factores externos, como por ejemplo la complejidad del escenario que se esté renderizando en ese momento, o incluso debido a la influencia de otros programas que estén consumiendo recursos los instantes de tiempo no se encuentran equiespaciados. En conclusión, desde el punto de vista de nuestro programa un integrador debe presentar la siguiente interfaz:

- Una rutina para colocar el integrador en las condiciones iniciales.
- Una rutina que avance el integrador al instante de tiempo actual. Sea cual sea el intervalo transcurrido. Es criterio del integrador si el avance se realiza en un sólo paso o en varios pasos.

Desde un punto de vista meramente informático no se requiere más y el usuario es libre de seleccionar en el fichero de entrada del modelo cualquier integrador que cumpla las anteriores condiciones, bien usando alguno de los que ya se encuentran disponibles en el fichero `integrador.py` o añadiendo su propio integrador utilizando como plantilla algunos de los anteriores.

4.1. Elección del integrador

Desde un punto de vista de numérico el integrador debe cumplir las siguientes condiciones:

- Debe conservar el carácter de estabilidad de la ecuación diferencial.
- Debe aproximar a la solución con un error tolerable.

- Debe ser posible evaluarlo en un tiempo adecuado.

Desgraciadamente las anteriores condiciones suelen ser excluyentes por lo que hay que alcanzar un compromiso. En general no se puede decir nada acerca de la estabilidad de una solución general de las ecuaciones diferenciales pero nos podemos hacer una idea a partir del análisis de las ecuaciones linealizadas. Se ha escrito un pequeño programa que calcula las derivadas de estabilidad en diversas condiciones de vuelo y calcula los autovalores de los modos propios del movimiento linealizado. A continuación se muestra la aplicación para el Lynx. El círculo corresponde a vuelo a punto fijo y el triángulo a vuelo de avance a 160 nudos (figura 4.2)

Como se ve los modos más exigentes son el de convergencia en balance y sobre todo el asociado al giro del motor, que presenta una componente imaginaria muy alta. Teniendo en cuenta esto y que el paso de tiempo es del orden de 0.03 podemos descartar aquellos esquemas cuyas zonas de estabilidad no cubran los anteriores modos escalados por el paso de tiempo.

A continuación se discuten en qué medida son satisfactorios los integradores que se encuentran ya implementados.

Para analizar la estabilidad del integrador se ha calculado la región de estabilidad de cada integrador resolviendo numéricamente el polinomio (ver [4]):

$$\sum_{j=0}^{p} \left(\alpha_j - \omega f_j(\omega)\right) r^{p-j} = 0$$

para diversos valores del número complejo ω. Se ha extraído en cada punto el valor de la raíz de mayor valor absoluto r_{max} y se han dibujado las líneas de nivel de contorno para $r_{max} \leq 1$. Los valores de los anteriores coeficientes vienen especificados en la discusión de cada integrador.

Para estimar la precisión de la solución se ha calculado la respuesta del helicóptero en un caso cualquiera con diferentes pasos de tiempo. Para ello se ha realizado un vuelo, se ha guardado en fichero las magnitudes en los diferentes instantes de tiempo, incluidos los controles, se ha "serializado" el fichero de logging mediante la utilidad `serializa` y mediante un pequeño programa `test_integrador` se ha calculado la respuesta del helicóptero en función del tiempo para pasos de 0.03 y 0.015 segundos. Se ha calculado el máximo error global que se presenta en el intervalo de tiempo calculado y se han trazado las gráficas comparando ambas respuestas. La línea continua corresponde al paso menor y por tanto en teoría más preciso de 0.015 segundos mientras que los puntos, dibujados sólo cada 0.3 segundos, corresponden al paso de 0.03 segundos. Como las gráficas a simple vista son indistinguibles unas de otras se muestra sólo una, la calculada para el Predictor-Corrector (figura 4.3) y otra, que no converge, para el Adams-Bashforth 3 (figura 4.8). En esta última se ve cómo se necesita un paso pequeño para que la zona de estabilidad abarque todos los autovalores de la ecuación diferencial.

4.1.1. Euler

Es el primer integrador que se implementó debido a su sencillez. Su uso no se recomienda ya que presenta pésimas características de estabilidad y un error bastante pobre. A su favor hay que decir que por ser el más simple también es

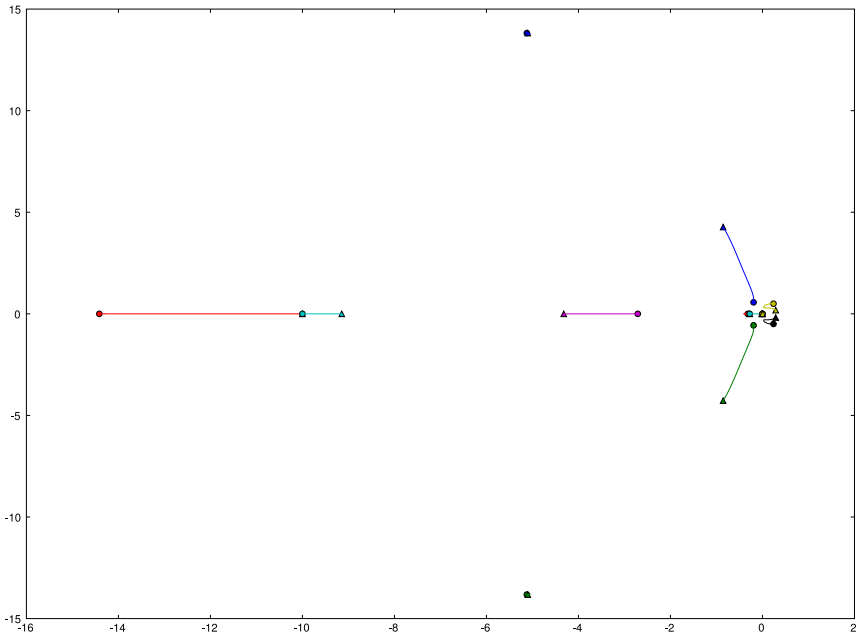


Figura 4.1: Modos propios del Lynx

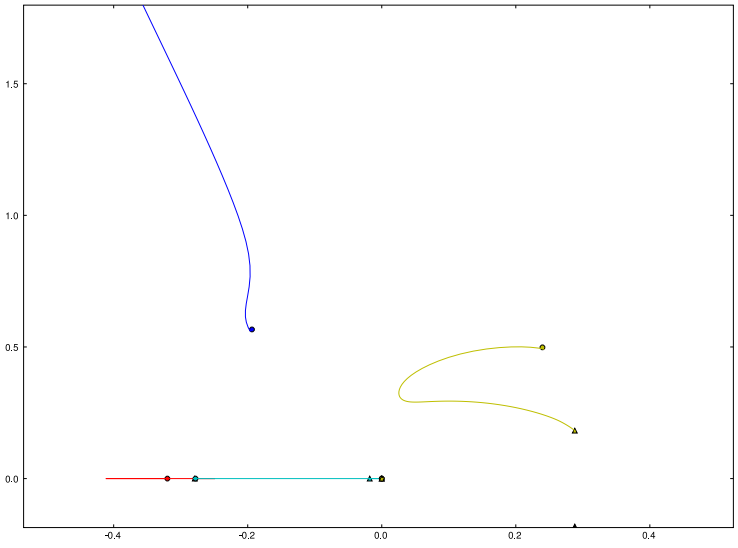


Figura 4.2: Detalle cerca del origen

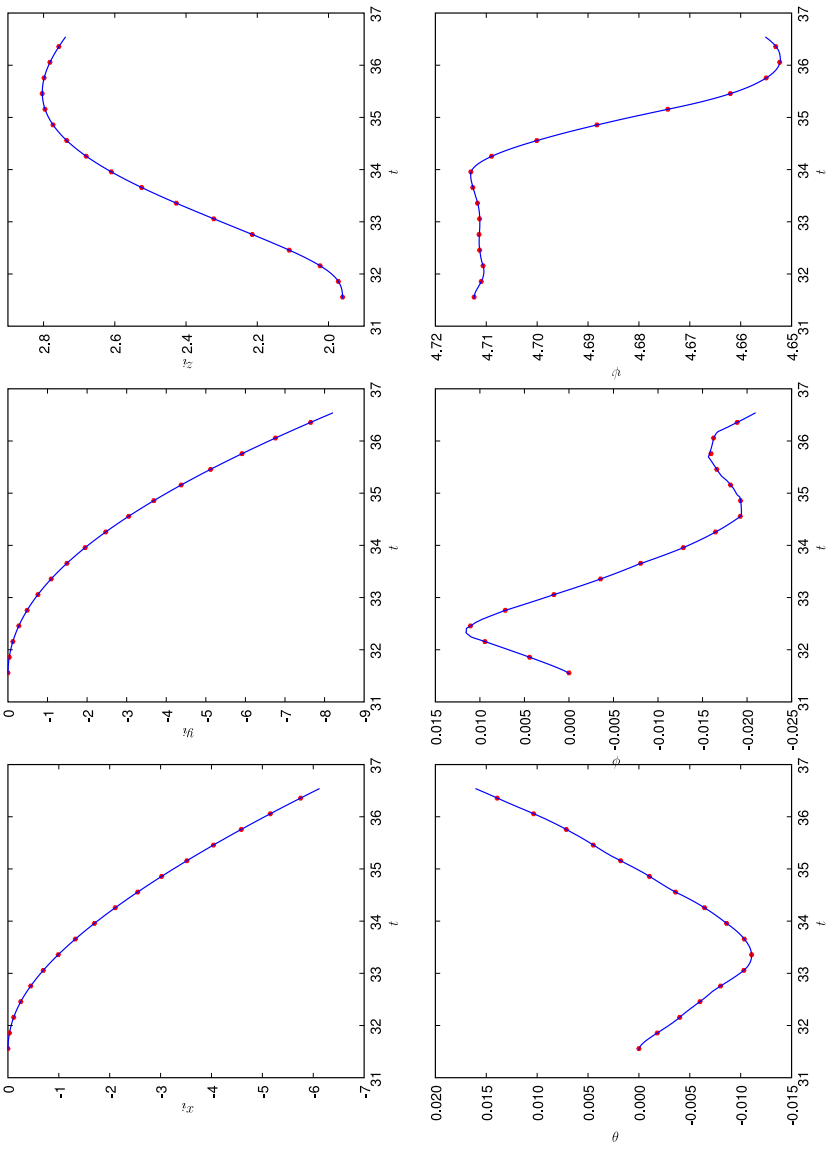


Figura 4.3: Cálculo de la posición y actitud del helicóptero en función del tiempo para dos pasos diferentes de integración

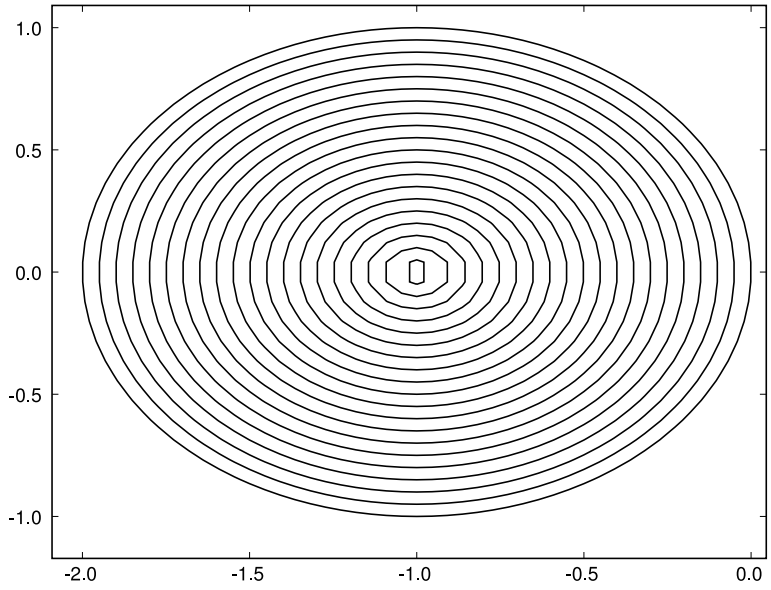


Figura 4.4: Región de estabilidad del Euler

el que menos tiempo requiere. En ocasiones ha llegado a fallar su estabilidad sobre todo debido al grado de libertad asociado al giro del motor. Se mantiene en el fichero de integradores para poder usarlo como plantilla para implementar otros integradores.

$$u^{n+1} = u^n + \triangle t F^n$$

Polinomio de estabilidad:

$$\alpha_0 = 1 \qquad f_0 = 0$$
$$\alpha_1 = -1 \qquad f_1 = 1$$

El máximo error global para el pequeño vuelo de test ha correspondido a y_i, y ha sido de 0.031 metros.

4.1.2. Runge-Kutta de orden 4

Fue el segundo integrador que se implementó debido a sus excelentes propiedades de estabilidad. Es el integrador a utilizar si se presenta algún problema de divergencia explosiva en las ecuaciones ya que puede ayudar a identificar si se trata de un error en el modelo o en el integrador. Desgraciadamente exige la evaluación de la función en cuatro puntos intermedios por lo que es el peor integrador a usar para ahorrar coste computacional. No se recomienda su uso si se va a utilizar el simulador interactivamente ya que debido a que se encuentra implementado en python, un lenguaje interpretado bastante lento (30 veces más

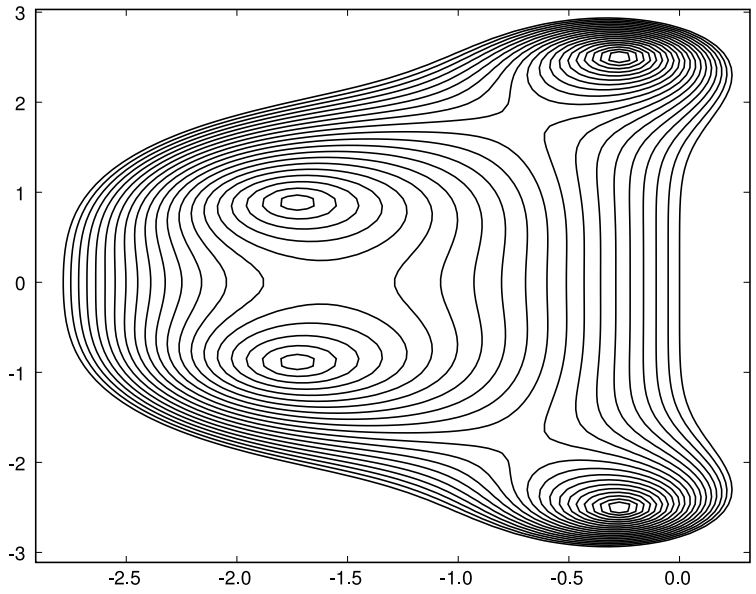


Figura 4.5: Región de estabilidad del Runge-Kutta 4

lento que C), se producirían desfases intolerables entre las acciones del piloto y la respuesta del helicóptero. Esto es un fallo de implementación del simulador, en futuras mejoras habría que reimplementar las partes que más tiempo consumen en C/C++ u otro lenguaje de bajo nivel.

$$
\begin{aligned}
u^{n+1} &= u^n + \frac{\triangle t}{6}(k_1 + 2k_2 + 2k_3 + k_4) \\
k_1 &= F(u^n, t_n) \\
k_1 &= F(u^n + \triangle t k_1/2, t_n + \triangle t/2) \\
k_1 &= F(u^n + \triangle t k_2/2, t_n + \triangle t/2) \\
k_1 &= F(u^n + \triangle t k_3, t_n + \triangle t)
\end{aligned}
$$

Polinomio de estabilidad:

$$
\begin{aligned}
\alpha_0 &= 1 & f_0 &= 0 \\
\alpha_1 &= -1 & f_1 &= \frac{1}{24}(w^3 + 4w^2 + 12w + 24)
\end{aligned}
$$

El máximo error global para el pequeño vuelo de test ha correspondido a y_i, y ha sido de 0.049 metros.

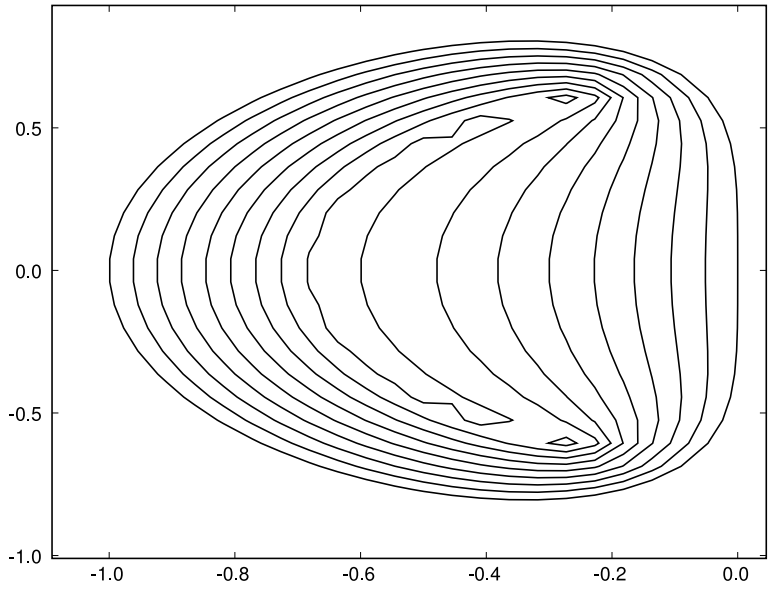


Figura 4.6: Región de estabilidad del Adams-Bashforth 2

4.1.3. Adams-Bashforth 2

Mejora las características del Euler y además no exige evaluaciones adicionales de la función. Es uno de los dos integradores recomendados para todo uso.

$$
\begin{aligned}
u^{n+1} &= u^n + \triangle t(\beta_1 F^n + \beta_2 F^{n-1}) \\
\beta_1 &= \frac{2\triangle t_2 + \triangle t_1}{2\triangle t_2} \\
\beta_2 &= -\frac{\triangle t_1}{2\triangle t_2}
\end{aligned}
$$

Polinomio de estabilidad (para paso constante):

$$
\begin{aligned}
\alpha_0 &= 1 & f_0 &= 0 \\
\alpha_1 &= -1 & f_1 &= \frac{3}{2} \\
& & f_2 &= -\frac{1}{2}
\end{aligned}
$$

El máximo error global para el pequeño vuelo de test ha correspondido a x_i, y ha sido de 0.001 metros.

4.1.4. Adams-Bashforth 3

Se ha implementado este integrador para ver si se podía mejorar la precisión del Adams-Bashfort 2. Desgraciadamente no es un integrador adecuado, ya que requiere pasos de tiempo la mitad de pequeños que el Adams-Bashofrth 2, por lo que puede llegar a diverger.

$$
\begin{aligned}
u^{n+1} &= u^n + \triangle t(\beta_1 F^n + \beta_2 F^{n-1} + \beta_3 F^{n-2}) \\
\beta_1 &= 1 + \frac{\triangle t_1\,(2\triangle t_1 + 6\triangle t_2 + 3\triangle t_3)}{6\triangle t_2\,(\triangle t_2 + \triangle t_3)} \\
\beta_2 &= -\frac{\triangle t_1\,(2\triangle t_1 + 3\triangle t_2 + 3\triangle t_3)}{6\triangle t_2 \triangle t_3} \\
\beta_3 &= \frac{\triangle t_1\,(2\triangle t_1 + 3\triangle t_2)}{6\triangle t_3\,(\triangle t_2 + \triangle t_3)}
\end{aligned}
$$

Polinomio de estabilidad (para paso constante):

$$
\begin{aligned}
\alpha_0 &= 1 & f_0 &= 0 \\
\alpha_1 &= -1 & f_1 &= \frac{23}{12} \\
& & f_2 &= -\frac{4}{3} \\
& & f_3 &= \frac{5}{12}
\end{aligned}
$$

Para paso de 0.03 segundos el método se ha vuelto inestable.

4.1.5. Predictor-Corrector Adams-Bashforth-Moulton 2

Finalmente el mejor integrador para esta aplicación presenta una zona de estabilidad suficientemente grande y sólo requiere dos evaluaciones de la función en cada instante de tiempo por lo que es suficientemente rápido para ejecutarlo en modo interactivo. Como valor añadido permite estimar el error numérico.

$$
\begin{aligned}
u_*^{n+1} &= u^n + \triangle t(\beta_{p1} F^n + \beta_{p2} F^{n-1}) \\
F_*^{n+1} &= F(u_*^{n+1}, t_{n+1}) \\
u^{n+1} &= u^n + \triangle t(\beta_{c0} F_*^{n+1} + \beta_{c1} F^n) \\
\beta_{p_1} &= \frac{2\triangle t_2 + \triangle t_1}{2\triangle t_2} & \beta_{c_0} &= \frac{1}{2} \\
\beta_{p_2} &= -\frac{\triangle t_1}{2\triangle t_2} & \beta_{c_1} &= \frac{1}{2}
\end{aligned}
$$

Polinomio de estabilidad (para paso constante):

$$
\begin{aligned}
\alpha_0 &= 1 & f_0 &= 0 \\
\alpha_1 &= -1 & f_1 &= 1 + \frac{3}{4}\omega \\
& & f_2 &= -\frac{1}{4}\omega
\end{aligned}
$$

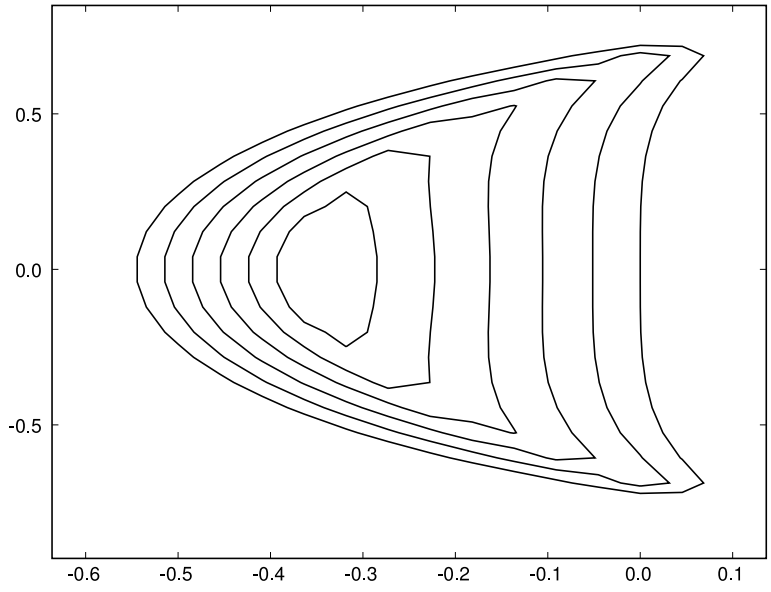


Figura 4.7: Región de estabilidad del Adams-Bashforth 3

El máximo error global para el pequeño vuelo de test ha correspondido a y_i, y ha sido de 0.061 metros.

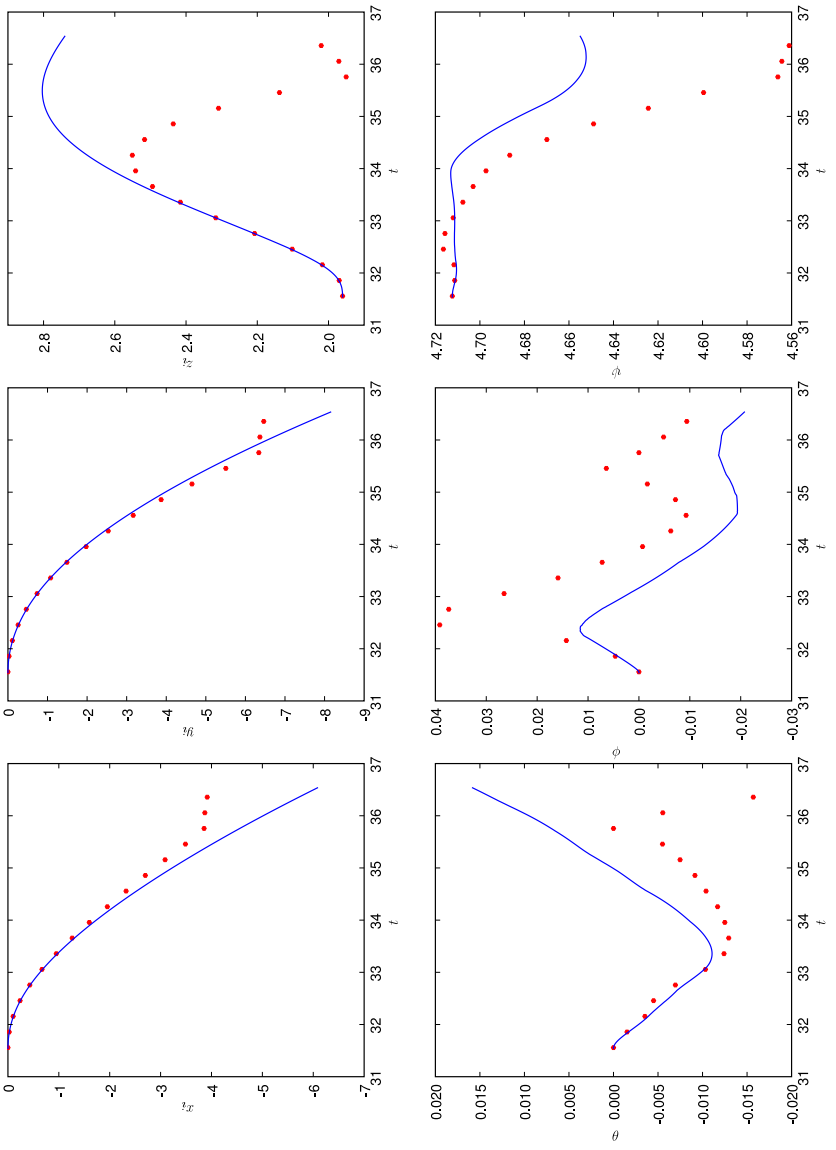


Figura 4.8: Inestabilidad del Adams-Bashforth 3

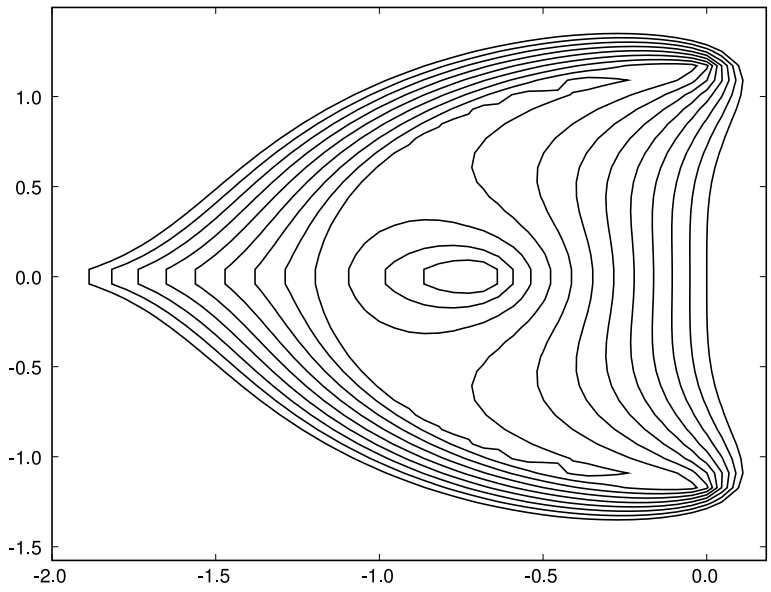


Figura 4.9: Región de estabilidad del Adams-Bashforth-Moulton 2

Capítulo 5

Trimado

Para calcular la posición de equilibrio del helicóptero es necesario resolver por lo menos un conjunto de 15 ecuaciones:

- 6 Ecuaciones de fuerzas y momentos del fuselaje.
- 3 Ecuaciones de orientación del helicóptero.
- 1 ecuación para la velocidad de rotación del rotor.
- 1 ecuación para el cálculo de par de motor a partir de la velocidad del rotor.
- 4 ecuaciones adicionales, tantas como controles tiene el helicóptero para igualar el número de incógnitas con el de ecuaciones.

Muchas de las anteriores ecuaciones son no lineales:

- Las fuerzas de inercia dependen cuadráticamente de la velocidad y la velocidad angular.
- La fuerza de la gravedad depende mediante funciones trigonométricas del cabeceo y del balance.
- También dependen de la misma forma las fuerzas aerodinámicas que además se pueden encontrar dadas en forma de tablas para interpolar.
- Las fuerzas del rotor dependen linealmente y cuadráticamente del batimiento y trigonométricamente de la orientación.
- La velocidad inducida en el rotor depende de forma no lineal del coeficiente de sustentación.
- El resto de ecuaciones son lineales: los controles en función de los pasos y las ecuaciones del batimiento y la velocidad inducida no uniforme.

Además, debido a las características del helicóptero, se trata de un sistema totalmente acoplado y que por ejemplo, el control longitudinal del paso del rotor influye sobre el batimiento lateral, lo que provoca momentos y fuerzas laterales, etc...

A falta de un mejor análisis matemático del problema la estrategia que se ha seguido para resolver el trimado ha sido siguiendo un razonamiento físico, similar al que seguiría un piloto a los mandos del helicóptero: primero y a pesar de todo, dividir el problema en una parte longitudinal y otra lateral. Se resuelven secuencialmente las ecuaciones longitudinales y se iteran hasta que converge un determinado parámetro, por ejemplo el cabeceo del helicóptero. Se hace de la misma forma para las ecuaciones laterales utilizando como parámetro el ángulo de balance. Finalmente se itera en la velocidad angular y se resuelven las ecuaciones

5.1. Condiciones adicionales

Especificamos las condiciones de trimado dadas, por ejemplo, por los siguientes 4 parámetros, aunque podrían haber sido otros:

- Velocidad de giro Ω_a.
- Velocidad V.
- Ángulo de subida γ_f.
- Ángulo de resbalamiento β.

5.2. Cálculo de la velocidad y velocidad angular

Conocidos estos cuatro parámetros, suponemos conocidos, ya sea de una estimación inicial o de un cálculo anterior el valor de cabeceo θ y el de balance ϕ, entonces podemos calcular la velocidad u, v, w y la velocidad angular p, q, r en ejes cuerpo. La velocidad angular se calcula rápidamente utilizando las relaciones usuales para ángulos de Euler:

$$\begin{aligned} p &= -\Omega_a \sin\theta \\ q &= \Omega_a \cos\theta \sin\phi \\ r &= \Omega_a \cos\theta \cos\phi \end{aligned}$$

El cálculo de la velocidad es un poco más elaborado. Para ello expresamos la velocidad en ejes cuerpo y en unos ejes inerciales:

$$\begin{aligned} \vec{V} &= V\left[\frac{u}{V}\vec{i}_b + \sin\beta\vec{j}_b + \frac{w}{V}\vec{k}_b\right] \\ &= V\left[\cos\gamma\cos\chi\vec{i}_0 + \cos\gamma\sin\chi\vec{j}_0 + \sin\gamma\vec{k}_0\right] \end{aligned}$$

Donde el ángulo χ es el ángulo que forma la proyección de la velocidad sobre el x_0y_0 con el eje x_0, y constituye una incógnita junto con u y v.

Utilizando la matriz de cambio de base de ejes inerciales a ejes cuerpo que se obtuvo en el capítulo dedicado a sistemas de referencia obtenemos 3 ecuaciones. Resolviendo una de ellas:

$$\sin\beta - \sin\gamma\sin\phi\cos\theta = \cos\gamma\sin\theta\sin\phi\cos(\chi-\psi) + \cos\gamma\cos\phi\sin(\chi-\phi)$$

De donde despejamos el ángulo $\chi - \psi$. Una vez conocido este ángulo obtenemos de las otras dos ecuaciones:

$$\frac{u}{V} = \cos\theta\cos\gamma\cos(\chi-\psi) - \sin\theta\sin\gamma$$
$$\frac{w}{V} = \cos\gamma\sin\theta\cos\phi\cos(\chi-\psi) + \cos\gamma\sin\phi\sin(\chi-\psi) + \sin\gamma\cos\phi\cos\theta$$

La anterior ecuación para $\chi - \psi$ tiene dos soluciones que se corresponden a distintos ángulos de ataque para el helicóptero. Para entender esta multiplicidad de soluciones podemos visualizar la condición en γ como un requisito para que la velocidad se encuentre en el cono de semiángulo $\frac{\pi}{2} - \gamma$ que tiene como eje el eje z_0, y la condición en β como otro requisito para que la velocidad se encuentre en un cono de semiángulo $\frac{\pi}{2} - \beta$ con eje el y_B. La velocidad será entonces la intersección de los dos anteriores conos que comparten su vértice, por lo que habrá dos soluciones en general aunque puede que no haya ninguna o incluso una, si ambos conos son tangentes.

De las dos soluciones nos quedamos con la que físicamente tiene mas sentido, que es la que de el menor valor de ángulo de ataque. En el caso de no haber solución habría que reintroducir alguno de los cuatro parámetros.

5.3. Cálculo de fuerzas aerodinámicas y de inercia

Conocida la velocidad en ejes cuerpo y supuestos cálculos anteriores para la velocidad inducida y ángulo de estela podemos determinar las fuerzas aerodinámicas de fuselaje y estabilizadores, y también determinamos las fuerzas de inercia. Calculamos también los ejes viento del rotor y cola, parámetros de avance y velocidades angulares adimensionalizadas con la velocidad de giro del rotor (suponemos conocida también).

5.4. Ecuaciones longitudinales

Suponemos conocidas las fuerzas y momentos del rotor de cola. A continuación vamos a hacer cumplir las ecuaciones longitudinales del movimiento, para ello vamos a suponer que nuestros parámetros de control son la tracción del rotor c_T para hacer cumplir la ecuación de fuerzas vertical, el batimiento longitudinal β_{1c} para hacer cumplir la ecuación de momentos de cabeceo y el ángulo de cabeceo θ que inclina la tracción del rotor para vencer la resistencia aerodinámica sobre todo del fuselaje. Más detalladamente:

1. Despejamos la fuerza vertical del rotor en ejes cuerpo Z_R, normalmente un valor cercano al peso del helicóptero que se puede utilizar como primera aproximación.

$$Z_R = M(-qu + pv) - (Z_f + Z_{fn} + Z_{tp} + Z_T) - Mg\cos\theta$$

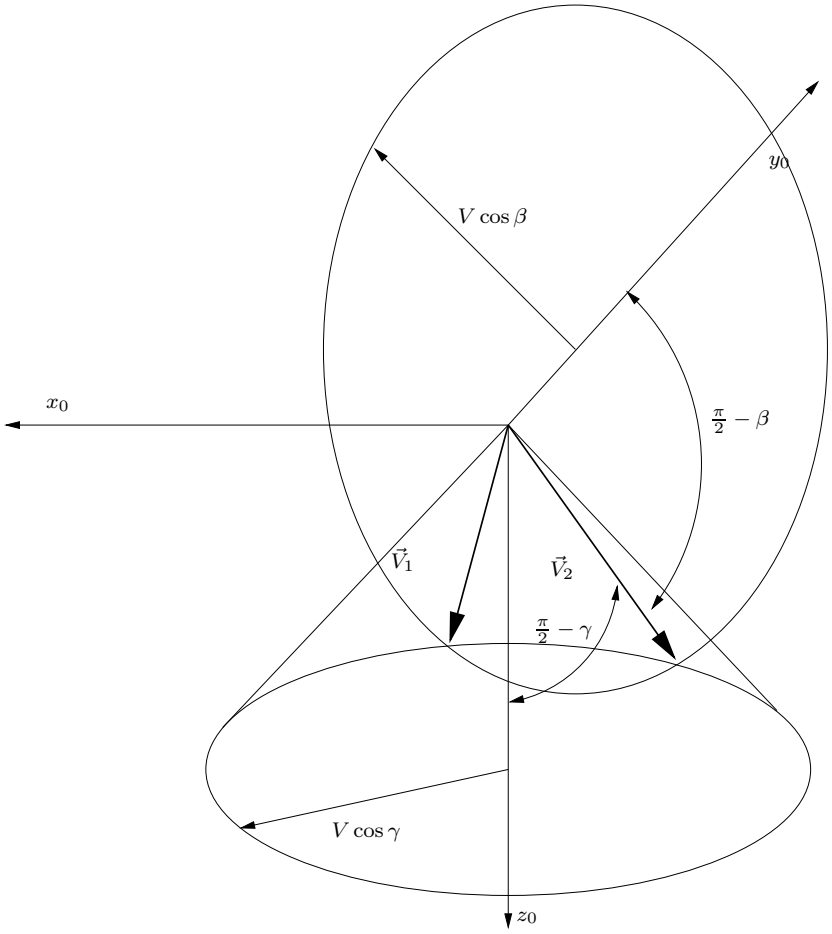


Figura 5.1: Posibles soluciones de $\vec{V}$

2. Despejamos el momento del rotor en ejes cuerpo de la ecuación de momentos de cabeceo:

$$M_R = -(I_{zz} - I_{xx})rp - I_{xz}(r^2 - p^2) - (M_f + M_{fn} + M_{tp} + M_T)$$

3. Suponiendo conocida la fuerza horizontal del rotor calculamos las fuerzas y momentos aplicados sobre la cabeza del rotor:

$$\begin{aligned} X_{Rh} &= X_R \\ Z_{Rh} &= Z_R \\ M_{Rh} &= M_R - Z_R x_{cg} + X_R h_R \end{aligned}$$

4. Obtenemos el coeficiente de sustentación del rotor a partir de las fuerzas y momentos en ejes viento:

$$c_T = -\frac{Z_{Rh_w}}{\rho(\Omega R)^2 \pi R^2}$$

5. Obtenemos el batimiento del rotor

$$\beta_{1cw} = \frac{-2M_{Rh_w}}{N_b K_\beta}$$

6. Conocido c_T y por tanto el coeficiente $\delta = \delta_0 + \delta_2 c_T^2$ y el batimiento β_{1cw} ya podemos obtener una estimación buena de la resistencia del rotor, calculamos $X_{Rh}|_w$ donde suponemos conocidos los numerosos datos que faltan, que de todas formas influyen mucho menos: λ_{1cw}, λ_{1sw}, etc... Pasamos a ejes cuerpo y tenemos X_R

7. Despejamos de las ecuaciones horizontal y vertical la fuerza de gravedad:

$$\theta = \arctan\left(-\cos\phi \frac{M(-rv + qw) - (X_f + X_{fn} + X_{tp} + X_T + X_R)}{M(-qu + pv) - (Z_f + Z_{fn} + Z_{tp} + Z_T + Z_R)}\right)$$

8. Comprobamos el error en θ, es decir, su diferencia respecto al valor anteriormente calculado. Si no cae dentro del error admisible repetimos el proceso, si es aceptable pasamos al cumplimiento de las ecuaciones laterales

5.5. Ecuaciones laterales

A continuación hacemos cumplir las ecuaciones laterales, nuestras variables de control van a ser la tracción de la cola c_{T_T} para hacer cumplir la ecuación de momentos de guiñada, el batimiento β_{1s} para cumplir el momento de balance y el ángulo de balance ϕ para contrarrestar mediante el rotor la fuerza lateral del rotor de cola.

1. Una vez conocido c_T del anterior bucle calculamos la velocidad inducida media λ_0 que nos permite junto a a c_T calcular el par del rotor, para ello resolvemos la ecuación no lineal:

$$\frac{\lambda_0}{k_i} = \frac{c_T}{2\sqrt{\left(\frac{\mu}{k_\nu}\right)^2 + \left(\frac{1}{k_\nu}^2 - \frac{1}{k_i}^2\right)\mu_z^2 + \left(\frac{\mu_z - \lambda_0}{k_i}\right)^2}}$$

   Donde ahora es c_T una constante que no depende de λ_0.

2. Calculamos el par del rotor en la cabeza del rotor en ejes viento y los pasamos al centro de masas en ejes cuerpo.

3. De la ecuación del momento de guiñada despejamos el par que realiza el rotor de cola.

$$N_T = -(I_{xx} - I_{yy})pq + I_{xz}qr - (N_f + N_{fn} + N_{tp} + N_R)$$

   Despejamos del par del rotor de cola la sustentación del rotor de cola:

$$Y_T = T_T = -\frac{N_T}{l_T + x_{cg}}$$

4. Con la tracción del rotor de cola podemos estimar bien el momento de balance provocado por el rotor de cola:

$$L_T = Y_T h_T$$

   Y despejar de la ecuación del momento de balance el momento que debe de realizar el rotor:

$$L_R = (I_{xx} - I_{yy})qr - I_{xz}pq - (L_f + L_{fn} + L_{tp} + L_T)$$

   Calculamos el momento de balance en la cabeza del rotor:

$$L_{Rh} = L_R - Y_R h_R$$

   Y despejamos el batimiento lateral en ejes viento:

$$\beta_{1sw} = -\frac{L_{Rh_w}}{N_b K_\beta}$$

5. Conocido el batimiento lateral y la tracción del rotor se puede estimar muy bien la fuerza lateral del rotor; una vez calculada despejamos ϕ de la ecuación de fuerzas lateral.

$$\phi = \arcsin \frac{1}{Mg\cos\theta}\left[M(uv - wp) - (Y_f + Y_{fn} + Y_{tp} + Y_T\right]$$

6. Comprobamos el error en ϕ; si cae dentro de un margen aceptable continuamos con el cálculo de la velocidad angular del rotor, si no, comenzamos de nuevo desde el principio calculando θ

5.6. Velocidad angular del rotor

Lo primero que necesitamos es calcular el par del rotor de cola. Para ello necesitamos su velocidad inducida. Una vez calculado el par es muy sencillo resolver las ecuaciones del motor.

1. Calculamos el coeficiente de sustentación de la cola:

$$c_{T_T} = \frac{T_T}{\rho(\Omega_T R_T)^2 \pi R_T^2 F_T}$$

   Donde recordamos que F_T es el factor de bloqueo empírico debido a la superficie del estabilizador vertical.

2. Calculamos el coeficiente de resistencia $\delta_T = \delta_{T_0} + \delta_{T_2} c_{T_T}^2$ y la velocidad inducida λ_{0_T} a partir de la ecuación no lineal:

$$\frac{\lambda_{0_T}}{k_i} = \frac{c_{T_T}}{2\sqrt{\left(\frac{\mu_T}{k_\nu}\right)^2 + \left(\frac{1}{k_\nu}^2 - \frac{1}{k_i}^2\right)\mu_{z_T}^2 + \left(\frac{\mu_{z_T} - \lambda_{0_T}}{k_i}\right)^2}}$$

3. Calculamos el par del rotor de cola Q_T y de ahí junto con el par del rotor despejamos el par de un motor:

$$Q_1 = \frac{1+P}{n}(Q_R + g_T Q_T)$$

4. Calculamos la velocidad angular:

$$\Omega = \Omega_i - \frac{Q_1}{K_3}$$

5. Calculamos los controles θ_0, θ_{1cw}, θ_{1sw}, el batimiento β_0 y las velocidades inducidas λ_{1cw} y λ_{1sw}. Para ello reorganizamos las ecuaciones de batimiento y de velocidad inducida pasando los controles al lado izquierdo y los batimientos β_{1cw} y β_{1sw} al lado derecho y añadiendo la ecuación que nos

aporta conocer c_T. Entonces nos queda un sistema lineal de 5 ecuaciones:

$$\begin{bmatrix} A^{11}_{\beta\beta} & A^{11}_{\beta\lambda} & A^{12}_{\beta\lambda} & -A^{11}_{\beta\theta} & -A^{13}_{\beta\theta} & -A^{14}_{\beta\theta} \\ A^{21}_{\beta\beta} & A^{21}_{\beta\lambda} & A^{22}_{\beta\lambda} & -A^{21}_{\beta\theta} & -A^{23}_{\beta\theta} & -A^{24}_{\beta\theta} \\ A^{31}_{\beta\beta} & A^{31}_{\beta\lambda} & A^{32}_{\beta\lambda} & -A^{31}_{\beta\theta} & -A^{33}_{\beta\theta} & -A^{34}_{\beta\theta} \\ A^{11}_{\lambda\beta} & A^{11}_{\lambda\lambda} & A^{12}_{\lambda\lambda} & -A^{11}_{\lambda\theta} & -A^{13}_{\lambda\theta} & -A^{14}_{\lambda\theta} \\ A^{21}_{\lambda\beta} & A^{21}_{\lambda\lambda} & A^{22}_{\lambda\lambda} & -A^{21}_{\lambda\theta} & -A^{23}_{\lambda\theta} & -A^{24}_{\lambda\theta} \\ 0 & 0 & 0 & \frac{1}{3}+\frac{\mu^2}{2} & \frac{\mu}{2} & 0 \end{bmatrix} \begin{Bmatrix} \beta_0 \\ \lambda_{1sw} \\ \lambda_{1cw} \\ \theta_0 \\ \theta_{1sw} \\ \theta_{1cw} \end{Bmatrix} =$$

$$- \begin{bmatrix} A^{12}_{\beta\beta} & A^{13}_{\beta\beta} \\ A^{22}_{\beta\beta} & A^{23}_{\beta\beta} \\ A^{32}_{\beta\beta} & A^{33}_{\beta\beta} \\ A^{12}_{\lambda\beta} & A^{13}_{\lambda\beta} \\ A^{22}_{\lambda\beta} & A^{23}_{\lambda\beta} \\ 0 & 0 \end{bmatrix} \begin{Bmatrix} \beta_{1cw} \\ \beta_{1sw} \end{Bmatrix} + \begin{bmatrix} A_{\beta\omega} \\ A_{\lambda\omega} \\ -\frac{\mu}{4} \end{bmatrix} \begin{Bmatrix} \bar{p}_w \\ \bar{q}_w \end{Bmatrix} +$$

$$\begin{Bmatrix} \vec{A}_{\beta\lambda_0} \\ \vec{A}_{\lambda\lambda_0} \\ -\frac{1}{2} \end{Bmatrix} (\mu_z - \lambda_0) + \begin{Bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ \frac{2c_T}{a_0 s} - \frac{1}{4}(1+\mu^2)\theta_t \end{Bmatrix}$$

Despejamos las incógnitas del anterior sistema y pasamos de ejes viento a ejes rotor el paso y el batimiento.

6. Calculamos el control de cola θ_{0_T} y los batimientos β_{0_T}, β_{1cw_T} y β_{1sw_T}. Como hicimos para el rotor principal hay que reordenar las ecuaciones y añadir el hecho de que conocemos c_{T_T}. Queda entonces el siguiente sistema lineal:

$$\begin{bmatrix} & A_{\beta\beta_T} & & -A_{\beta\theta_{0_T}} \\ k_3\left(\frac{1}{3}+\mu_T^2\right) & 0 & k_3\frac{\mu_T}{2} & \frac{1}{3}+\mu_T^2 \end{bmatrix} \begin{Bmatrix} \beta_{0_T} \\ \beta_{1cw_T} \\ \beta_{1sw_T} \\ \theta_{0_T} \end{Bmatrix} =$$

$$\begin{Bmatrix} \vec{A}_{\beta\lambda_{0_T}} \\ -\frac{1}{2} \end{Bmatrix} (\mu_{z_T} - \lambda_{0_T}) + \begin{Bmatrix} \vec{A}_{\beta\theta_{t_T}}\theta_{t_T} \\ \frac{2c_{T_T}}{a_{0_T}s_T} \end{Bmatrix}$$

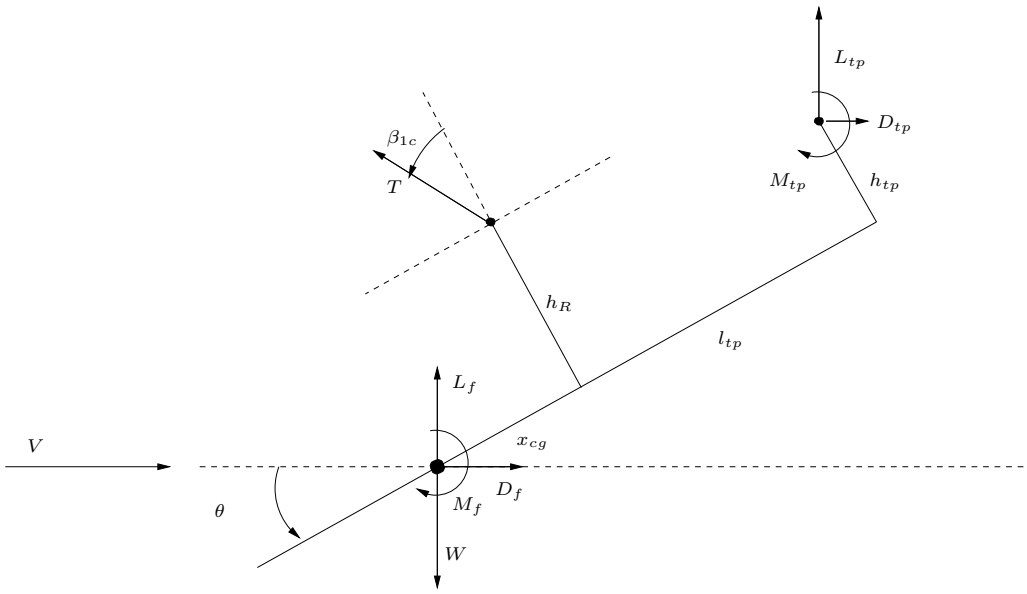
Resolvemos el sistema, pasamos de ejes viento a ejes rotor de cola y calculamos las reacciones y momentos de la cola sobre el centro de masas.

7. Comprobamos el error en Ω; si cae dentro del margen termina el proceso de trimado, si no volvemos a calcular θ

5.7. Convergencia del trimado

Es difícil asegurar la convergencia del trimado ya que estamos resolviendo un sistema de ecuaciones no lineales.

Cuando se resuelven las ecuaciones no lineales despejamos un parámetro a calcular de la ecuación. De esta forma obtenemos para dicho parámetro una iteración de punto fijo del tipo:



$$x_{n+1} = f(x_n)$$

Suponiendo la existencia del punto fijo la única forma de garantizar la convergencia es que en un entorno del punto fijo se cumpla que la derivada sea menor que uno y que el punto de partida esté dentro de dicho intervalo.

$$|f'(x)| < 1, x \in [a, b]$$
$$x_0 \in [a, b]$$

Esto quiere decir que ecuaciones tan simples como:

$$x = a + bx$$

no convergen para ningún punto de partida si $|b| >= 1$. De hecho, la solución exacta de

$$x_{n+1} = a + bx_n$$

es:

$$x_n = \frac{a}{1-b} + b^n \left(x_0 - \frac{a}{1-b} \right)$$

Esta situación de divergencia se produce en el trimado del helicóptero al ir aumentando la velocidad, en la iteración de las ecuaciones longitudinales. A continuación se realiza un examen aproximado del equilibrio longitudinal para averiguar en qué condiciones se produce la inestabilidad de la iteración.

Supongamos el helicóptero en vuelo rectilíneo horizontal y uniforme. En el análisis de fuerzas sólo consideramos el rotor, el fuselaje y el estabilizador horizontal y despreciamos el estabilizador vertical y el rotor de cola. En el diagrama se pueden observar la posición de las fuerzas y momentos del helicóptero.

Del equilibrio vertical:

$$\cos(\beta_{1c}+\theta)T + L_f + L_{tp} = W$$

y despreciando la sustentación del fuselaje y del estabilizador horizontal frente al peso y la sustentación del rotor $L_f, L_{tp} \ll T, W$, y suponiendo ángulos pequeños $\beta_{1c}, \theta \ll 1$:

$$T \approx W \tag{5.1}$$

Del equilibrio horizontal:

$$T\sin(\beta_{1c}+\theta) = D_f + D_{tp} \tag{5.2}$$

Como siempre suponemos polar parabólica para la resistencia:

$$D_f = \frac{1}{2}\rho V^2 S_f \left(c_{D_{0_f}} + c_{D_{\alpha_f}}\theta^2\right)$$
$$D_{tp} = \frac{1}{2}\rho V^2 S_{tp} \left(c_{D_{0_{tp}}} + c_{D_{\alpha_{tp}}}\theta^2\right)$$

Como los ángulos son pequeños y sustituyendo la sustentación del rotor por el peso del helicóptero, como se obtuvo del equilibrio vertical, tenemos:

$$\theta \approx \frac{D_0}{W} - \beta_{1c}$$

donde D_0 es la resistencia conjunta de fuselaje y estabilizador horizontal (sobre todo fuselaje) para ángulo de ataque nulo.

Del equilibrio de momentos, directamente para ángulos pequeños:

$$M_f + M_{tp} + D_{tp}\left[\left(l_{tp}+x_{cg}\right)\theta + h_{tp}\right] - L_{tp}\left[l_{tp}+x_{cg}-h_{tp}\theta\right] - Tx_{cg} - T\beta_{1c}h_R - \frac{N_b K_\beta}{2}\beta_{1c} = 0$$

Despejando el batimiento longitudinal:

$$\beta_{1c} = a + b\theta + c\beta_{1c}$$

Donde

$$a = \frac{\rho V^2}{N_b K_\beta}\left[S_f l_f c_{m_{0_f}} + S_{tp} l_{tp} c_{m_{0_{tp}}} + S_{tp} c_{D_{0_{tp}}} h_{tp} - S_{tp} c_{L_{0_{tp}}}(l_{tp}+x_{cg})\right] - \frac{2W}{N_b K_\beta}x_{cg}$$

$$b = \frac{\rho V^2}{N_b K_\beta}\left[S_f l_f c_{m_{\alpha_f}} + S_{tp} l_{tp} c_{m_{\alpha_{tp}}} + S_{tp} c_{D_{0_{tp}}}(l_{tp}+x_{cg}) - S_{tp}(-c_{L_{0_{tp}}} h_{tp} + c_{L_{\alpha_{tp}}}(l_{tp}+x_{cg}))\right]$$

$$c = -\frac{2Wh_R}{N_b K_\beta}$$

El esquema de iteración aproximado queda:

$$\left\{ \begin{array}{c} \theta_{n+1} \\ \beta_{1c_{n+1}} \end{array} \right\} = \left\{ \begin{array}{c} \frac{D_0}{W} \\ a \end{array} \right\} + \left[\begin{array}{cc} 0 & -1 \\ b & c \end{array} \right] \left\{ \begin{array}{c} \theta_n \\ \beta_{1c_n} \end{array} \right\}$$

La solución exacta de la anterior ecuación es análoga al caso unidimensional, es decir, si A y B son matrices:

$$\vec{x}_{n+1} = A + B\vec{x}_n$$

La solución exacta de la anterior iteración es:

$$\vec{x}_n = (I - B)^{-1}A + B^n \left(\vec{x}_0 - (I - B)^{-1}A\right)$$

Por lo que el esquema converge si la potencia de la matriz B tiende a la matriz nula. Esto es así si y sólo si los autovalores de la matriz B son todos menores que uno. En el caso del helicóptero los autovalores vienen dados por:

$$\lambda = \frac{c \pm \sqrt{c^2 - 4b}}{2}$$

Como c es negativo el mayor de ellos en valor absoluto corresponde al signo negativo y varía cuadráticamente con la velocidad. Una pequeña aplicación numérica demuestra que por ejemplo, para el BlackHawk, al llegar aproximadamente a 40 m/s el esquema diverge (como así ocurre en 48 m/s). Es por ello que necesitamos amortiguar el esquema de iteración.

5.7.1. Amortiguamiento de la iteración

Sea k una constante, entonces si x_0 es un punto fijo de $f(x)$, x_0 también es un punto fijo de $g(x) = x(1-k)+kf(x)$, por lo que podemos utilizar el siguiente esquema numérico para obtener el punto fijo de f:

$$x_{n+1} = x_n(1 - k) + kf(x_n)$$

con la ventaja de que $|g'(x)| = 1 - k(1 - f'(x))$ por lo que eligiendo el valor de k podemos hacer converger la iteración aunque sea $|f'(x)| > 1$.

Por lo tanto, en cada bucle del algoritmo de trimado aparece un factor de amortiguamiento que empieza valiendo uno, y que decrece progresivamente si se alcanza el máximo de iteraciones permitidas para conseguir la convergencia.

Alternativamente se puede especificar al método de trimado que utilice el método de Newton, en cuyo caso también se puede aplicar la fórmula de amortiguamiento. Si la fórmula para el método de Newton es:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} = h(x_n)$$

Hacemos:

$$x_{n+1} = x_n(1 - k) + kh(x_n) = x_n - k\frac{f(x_n)}{f'(x_n)}$$

Capítulo 6

Ejemplo de aplicación: Lynx

6.1. Breve descripción general e historia del Lynx

Se trata de un helicóptero militar fabricado por la compañía inglesa Westland. Voló por primera vez en 1971 y hoy en día se siguen utilizado y fabricando numerosas versiones, adaptadas según su uso, que va desde combate hasta transporte de tropas y vigilancia, siendo el helicóptero principal de la marina británica. Como características técnicas principales hay que decir que se trata de un helicóptero de pala no articulado, muy ágil y veloz (que ostenta el récord del mundo de velocidad para helicópteros) y que dispone de dos motores de aproximadamente 1000 caballos cada uno. Más adelante se concretan las características técnicas.

6.2. Fuentes de información y validez de los datos

Resulta extremadamente difícil encontrar información publicada suficientemente detallada para poder utilizar en un simulador, incluso para un modelo sencillo que sólo incluye un grado de libertad de batimiento en cada pala. Las dos fuentes primarias de información que se han utilizado han sido [12] y [9], que en realidad tratan de diferentes versiones del Lynx. En cualquier caso se ha intentado comparar los resultados que ambos documentos aportan y si alguno aportaba un dato que el otro no tenía siempre es mejor disponer de ese dato, que por lo menos aproximará al real, que ninguno.

6.3. Coeficientes aerodinámicos del fuselaje

A continuación se presentan los resultados según [9], que dan el área equivalente de las fuerzas y momentos sin ángulo de resbalamiento. Las unidades son ft^2 y ft^3 para fuerzas y momentos respectivamente y el ángulo de ataque se encuentra en grados. Las fuerzas se encuentran expresadas en componentes de ejes viento. Se disponen de dos tablas, una con la cola instalada y la otra sin la cola instalada, y ambas tablas sin el rotor. A pesar de que en el documento dice que son valores para el modelo a escala 1/5 se trata en realidad de los valores

para el helicóptero real, por lo que no hace falta escalar.

Fuerzas con la cola instalada:

α	L	D	Y	m	m	l
21	18.764	16.619	1.850	-53.045	-6.308	-25.583
18	15.694	14.945	1.505	-58.579	-9.310	-22.590
15	13.204	13.793	1.623	-74.844	-9.731	-20.858
12	9.546	12.548	1.657	-68.209	-11.909	-16.770
9	6.930	11.842	0.690	-67.183	-13.213	-9.773
6	4.062	11.447	0.622	-70.908	-14.155	-6.367
3	0.849	11.110	0.488	-58.629	-14.971	0.875
0	-1.901	11.060	0.572	-51.606	-17.124	4.373
-3	-5.265	11.749	0.774	-34.441	-21.093	10.866
-6	-9.924	12.103	0.606	0.572	-24.483	19.479
-9	-14.130	12.675	0.715	24.794	-30.278	27.023
-12	-17.511	13.768	0.606	36.468	-34.752	36.930
-15	-19.966	15.559	0.723	31.211	-31.405	38.024
-18	-21.632	17.990	0.252	12.288	-23.709	33.028
-21	-23.936	20.286	-0.782	-36.098	-12.742	32.439

Fuerzas sin la cola instalada:

α	L	D	Y	m	n	l
21	11.093	14.500	1.951	145.795	-19.386	-7.174
18	8.823	13.356	1.682	121.716	-19.193	-5.433
15	6.703	12.725	1.463	89.992	-18.495	-4.399
12	4.525	12.103	1.489	58.242	-18.966	-3.381
9	3.364	11.590	0.723	24.071	-17.847	-0.168
6	1.564	11.295	0.597	-8.688	-18.503	0.429
3	0.294	10.967	0.463	-42.817	-17.098	1.194
0	-0.833	10.934	0.488	-75.997	-18.377	2.195
-3	-2.237	11.421	0.614	-109.184	-22.809	3.886
-6	-4.188	11.783	0.412	-137.384	-25.643	6.039
-9	-6.434	12.161	0.513	-161.522	-30.942	9.108
-12	-8.368	12.893	0.681	-183.364	-39.857	16.072
-15	-10.412	14.289	1.161	-204.987	-48.209	19.529
-18	-13.095	15.526	1.489	-224.727	-56.661	27.948
-21	-16.207	17.208	1.161	-245.576	-61.606	35.029

Para obtener valores aproximados a ángulos de ataque mayores buscamos datos de algún helicóptero similar, por ejemplo se disponen de datos para el UH-60 en el rango de ángulos de ataque y de resbalamiento de -90º a 90º, excepto para el momento de guiñada, bien en forma de tablas como en [6] o en forma de funciones analíticas que ajustan a las tablas como en [5]. Mediante estos datos podemos calcular el valor de los coeficientes para ángulos de ataque de 90º y escalar los coeficientes comparando las áreas frontales y laterales de los fuselajes del Lynx y del UH60. No hay que dar demasiada importancia a la exactitud de los datos a grandes ángulos de ataque ya que normalmente se presentan a velocidades de vuelo bajas con lo que predominan las fuerzas de los rotores frente a las fuerzas aerodinámicas.

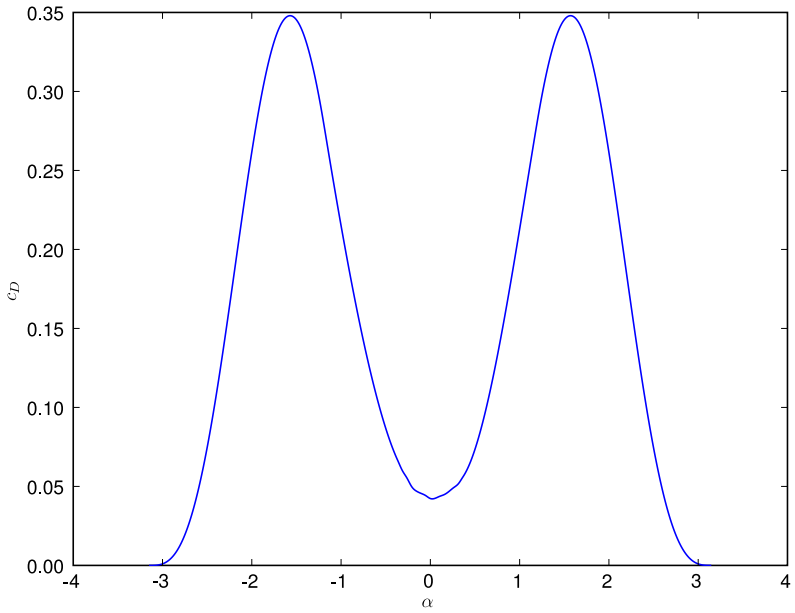


Figura 6.1: Coeficiente de resistencia

Finalmente, para el momento de guiñada utilizamos los valores genéricos suministrados por [12], dados en forma de tabla.

El simulador espera que los datos aerodinámicos del fuselaje se presenten en forma de tabla o funciones de coeficientes adimensionales referidos a ejes viento.

A continuación se muestran los coeficientes interpolados mediante splines cúbicas naturales, tal como se utilizan en el simulador, para todo el rango de posibles ángulos.

6.4. Coeficientes aerodinámicos de la cola

Para ángulos de ataque pequeño podemos obtener el comportamiento del estabilizador horizontal comparando los coeficientes de sustentación y momento para el fuselaje con cola y sin cola. Preferiblemente utilizamos los de momento ya que son más sensibles a la sustentación de la cola:

$$M_{cc} = M_{sc} - (l_{tp} + x_{cg})L_c$$

Donde M_{cc}, M_{sc} son el momento con cola y el momento sin cola respectivamente y L_c es la sustentación de la cola. Despejando y adimensionalizando con el área de la cola $S_{tp} = 1{,}197m^2$ obtenemos el valor del coeficiente de sustentación de la cola $c_{L_{tp}}$. También obtenemos la resistencia comparando la resistencia del fuselaje con cola y sin cola.

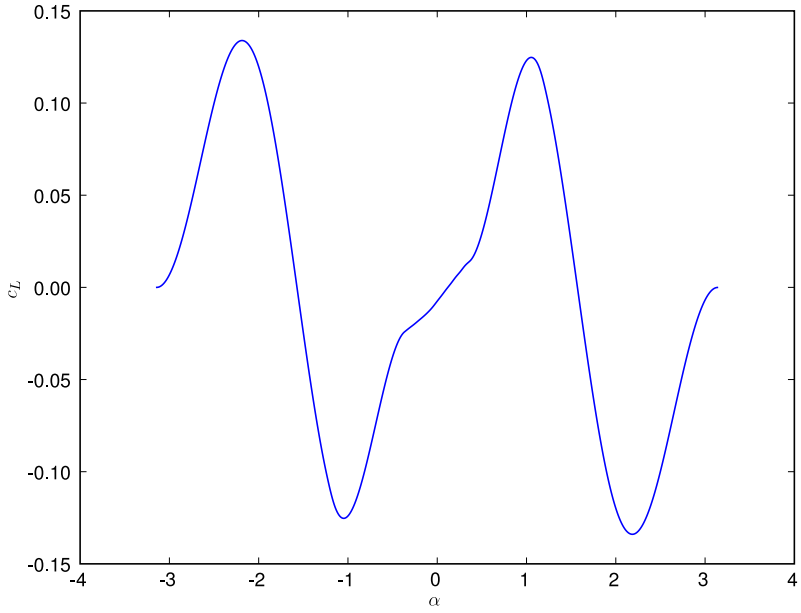


Figura 6.2: Coeficiente de sustentación

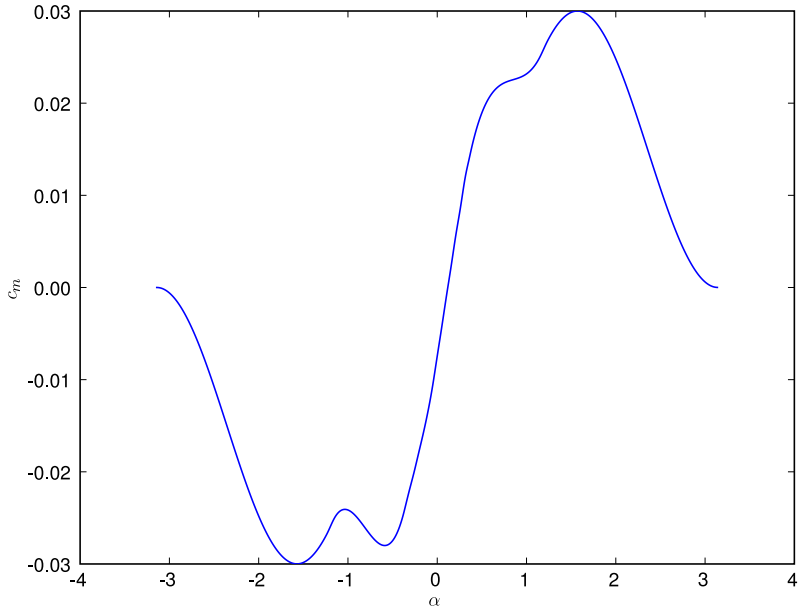


Figura 6.3: Coeficiente de cabeceo

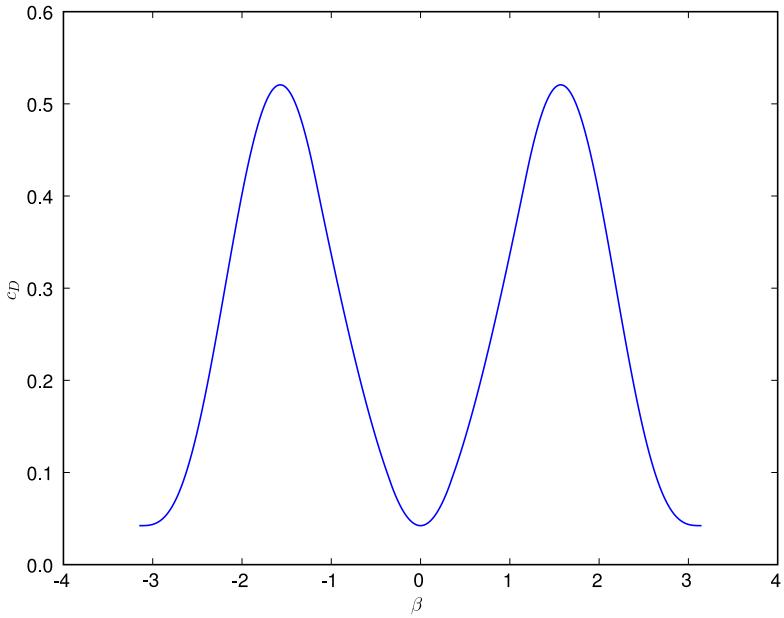


Figura 6.4: Coeficiente de resistencia

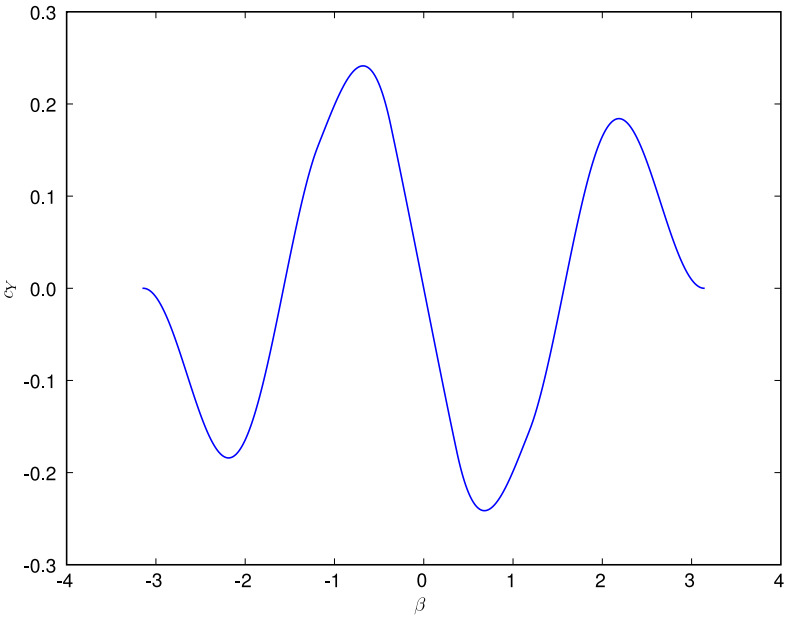


Figura 6.5: Coeficiente de fuerza lateral

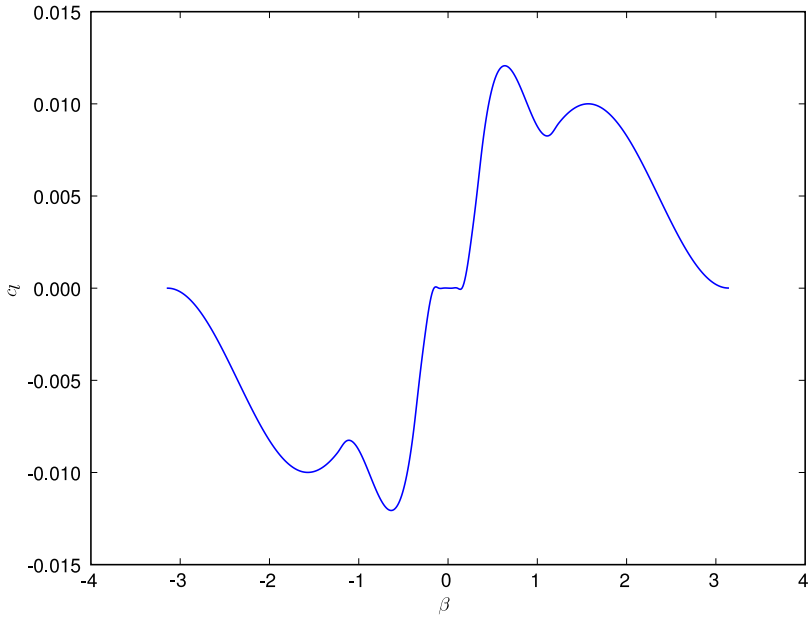


Figura 6.6: Coeficiente de balance

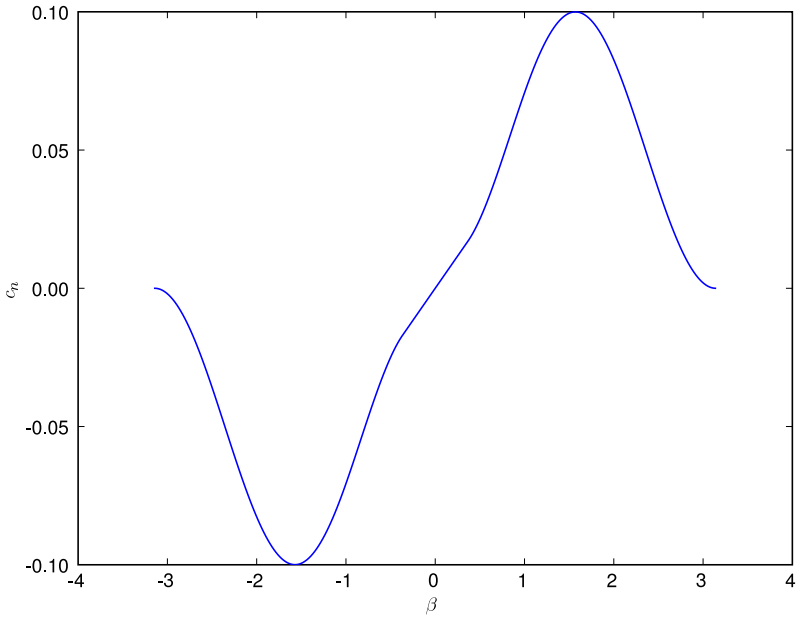


Figura 6.7: Coeficiente de guiñada

$\alpha(deg)$	$c_{L_{tp}}$	$c_{D_{tp}}$
21	0.6156	0.1645
18	0.5767	0.1233
15	0.5217	0.0829
12	0.3967	0.0345
9	0.2854	0.0196
6	0.1944	0.0118
3	0.0506	0.0111
0	-0.0740	0.0098
-3	-0.2277	0.0255
-6	-0.4234	0.0248
-9	-0.5709	0.0399
-12	-0.6705	0.0679
-15	-0.7165	0.0986
-18	-0.7052	0.1912
-21	-0.6128	0.2389

Dados estos datos de sustentación y resistencia tenemos varias opciones:

- Extendemos los datos tabulados al rango de ángulos de ataque de -180º a 180º y corregimos los valores para resbalamiento no nulo. Pasamos los resultados al simulador como una tabla para cada coeficiente que el simulador utiliza para interpolar.
- A partir de estos datos calculamos los coeficientes que definen el comportamiento de la cola para ángulos de ataque pequeños, es decir calculamos la pendiente de sustentación a_{tp}, el coeficiente de sustentación para ángulo de ataque nulo $c_{L_{tp_0}}$, los coeficientes de resistencia δ_{tp_0}, δ_{tp_1} y δ_{tp_2} y los valores de sustentación y resistencia máximos: $c_{L_{max}}$ y $c_{D_{max}}$ respectivamente. Pasamos estos datos junto con el alargamiento de la cola al simulador y éste, mediante un modelo aproximado, calcula c_L y c_D para todo ángulo de ataque y resbalamiento.

Eligiendo el segundo procedimiento, calculamos los coeficientes mediante ajustes por mínimos cuadrados (ver [16]). A continuación se presentan los resultados del ajuste:

$$\begin{aligned} a_{tp} &= 2{,}3663 \\ c_{L_{tp_0}} &= -0{,}1296 \\ c_{L_{tp_{max}}} &= 0{,}6 \\ \delta_{tp_0} &= 1{,}065 \cdot 10^{-3} \\ \delta_{tp_1} &= -8{,}470 \cdot 10^{-2} \\ \delta_{tp_2} &= 1{,}4698 \end{aligned}$$

No tenemos ningún dato del estabilizador vertical del Lynx, pero podemos deducir la pendiente de la curva de sustentación suponiendo que para ángulos de resbalamiento pequeño debe ser capaz de contrarrestar el momento del fuselaje, queda aproximadamente:

$$a_{fn} = 2{,}5$$

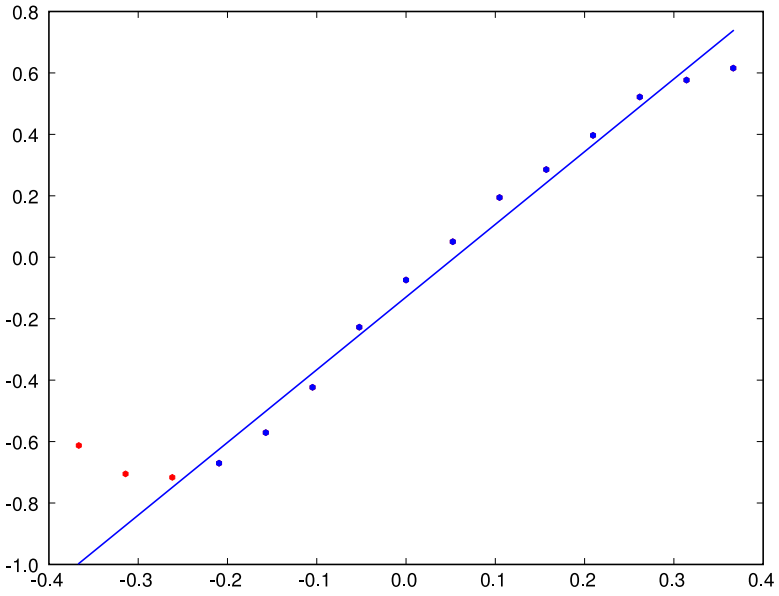


Figura 6.8: Coeficiente de sustentación del estabilizador horizontal

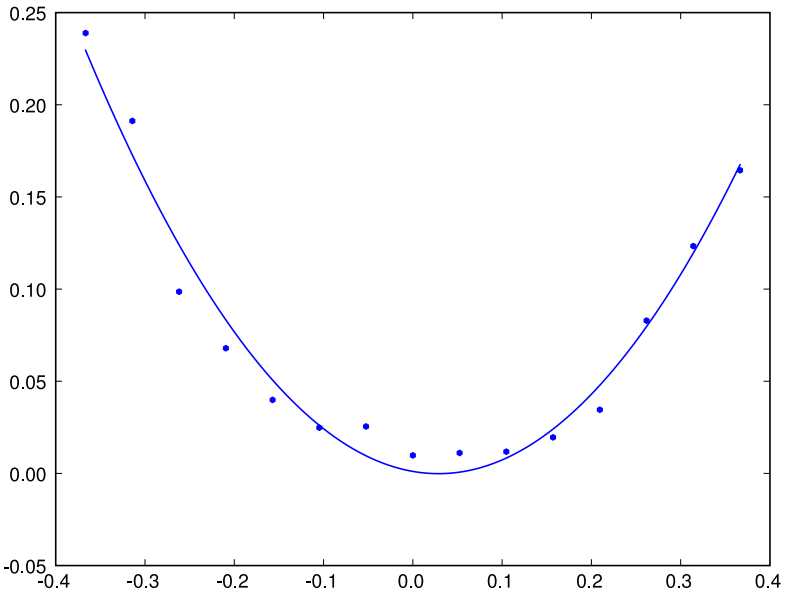


Figura 6.9: Coeficiente de resistencia del estabilizador horizontal

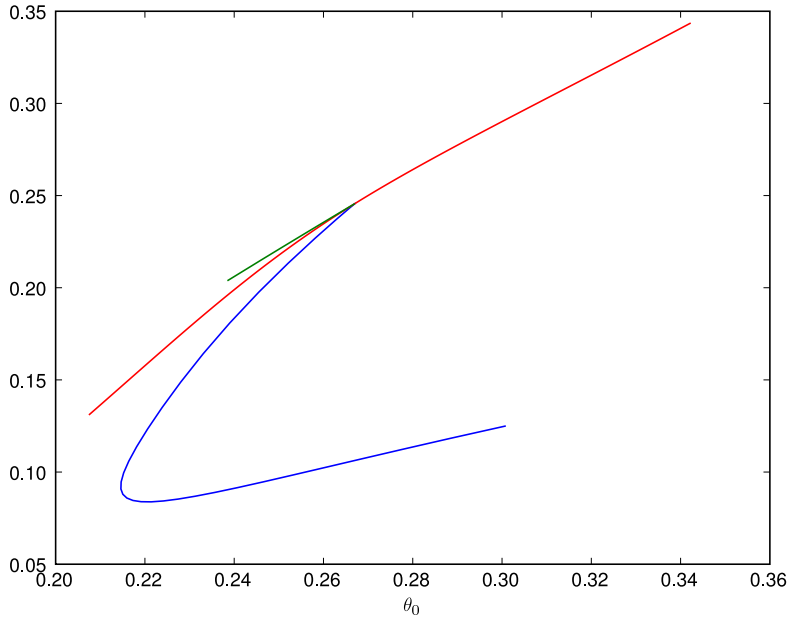


Figura 6.10: θ_{0_T} frente a θ_0

Suponiendo que debe extenderse la efectividad hasta, digamos los 18º aproximadamente de resbalamiento, podemos estimar:

$$c_{L_{fn}max} = 0{,}9$$

A falta de otro dato mejor utilizamos el mismo coeficiente de resistencia para la cola vertical que para la horizontal.

6.5. Controles

Como no se disponían de datos acerca del control del Lynx se ha intentado estimar unos valores razonables. Para ello suponemos que debemos poder tener control suficiente desde vuelo en punto fijo hasta velocidad de avance de 160 nudos y desde velocidad de descenso de 20 nudos hasta velocidad de ascenso de 20 nudos. Además intentaremos mezclar los controles longitudinales y de cola con el colectivo para intentar simplificar la respuesta del helicóptero. A continuación se muestran datos de trimado para punto fijo desde 0 a 30 metros sobre el suelo, (verde) y vuelo de avance (azul) y vertical (rojo) para las anteriores velocidades.

Como se ve, es imposible satisfacer automáticamente todas las condiciones, así que hacemos lo siguiente:

1. Ajustamos $\theta_0 = a + b\eta_{0p}$, para ello podemos exigir que por ejemplo el helicóptero se encuentre en vuelo a punto fijo para un 60 % de colectivo

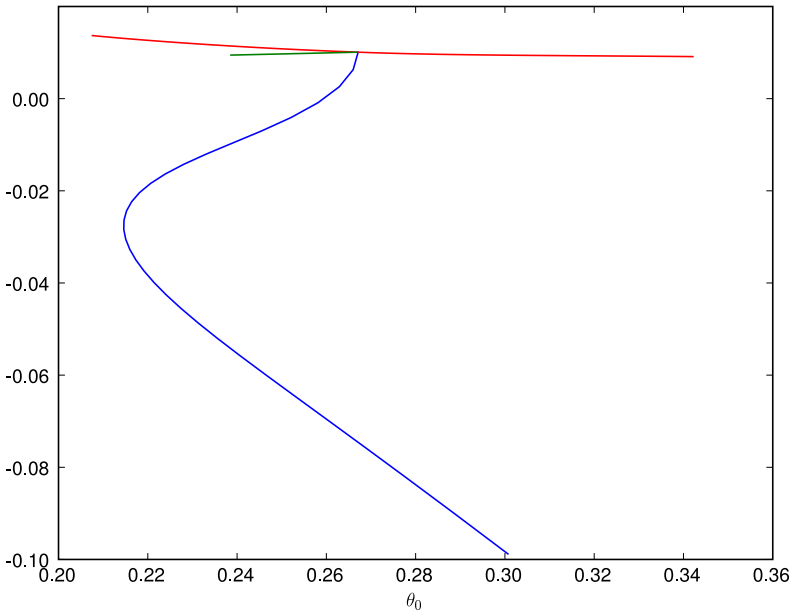


Figura 6.11: θ_{1s} frente a θ_0

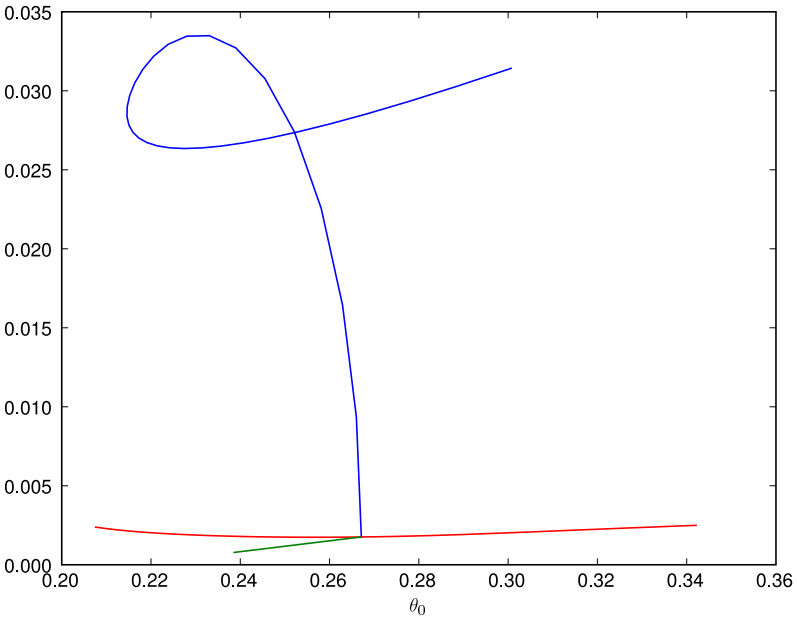


Figura 6.12: θ_{1c} frente a θ_0

del piloto, es decir:

$$\eta_{0p} = 0{,}6 \qquad \theta_0 = 0{,}239$$

y que cubramos todo el margen de operación del helicóptero:

$$\begin{aligned} \eta_{0p} &= 0 & \theta_0 &\leq 0{,}207 \\ \eta_{0p} &= 1 & \theta_0 &\geq 0{,}342 \end{aligned}$$

Las anteriores condiciones se traducen en:

$$\begin{aligned} a &= 0{,}239 - 0{,}6b \\ b &\geq 0{,}2575 \end{aligned}$$

Dando un poco de margen al final tenemos:

$$\theta_0 = 0{,}059 + 0{,}3\eta_{0p}$$

2. Ajustamos los cíclicos en la forma:

$$\begin{aligned} \theta_{1s} &= a_s + b_s\eta_{0p} + c_s\eta_{1s} \\ \theta_{1c} &= a_c + b_c\eta_{0p} + c_c\eta_{1c} \end{aligned}$$

para que sean en vuelo a punto fijo sea $\eta_{1sp} = \eta_{1cp} = 0$. Los cíclicos apenas varían con θ_0 así que hacemos:

$$a_s = b_s = a_c = b_c = 0$$

Y damos una sensibilidad razonable a los controles:

$$c_s = c_c = 0{,}2$$

3. Ajustamos el colectivo de cola de la misma forma que en los apartados anteriores y queda:

$$\theta_{0_T} = -0{,}061 + 0{,}441\eta_{0p} + 0{,}1\eta_{pp}$$

A falta de mejores datos los valores para el control automático se han colocado por prueba y error para dar un manejo más o menos sencillo del helicóptero.

6.6. Fichero de entrada

A continuación se muestra cómo queda el fichero de entrada para el helicóptero Lynx. Guardaríamos el siguiente texto en un fichero de nombre `Lynx.py` y lo cargaríamos con el parámetro `--modelo=Lynx`

6.6.1. Preámbulo

Todos los ficheros deben de incluir éste preámbulo. Hay que tener en cuenta que los ficheros de definición de los helicópteros son ficheros de código fuente en python. Esto no quiere decir que sea necesario dominar el lenguaje ya que el formato del fichero de entrada es suficientemente simple como para que no sea necesario. Basta con utilizar un fichero de ejemplo y sustituir los valores numéricos correspondientes. Los casos mas complejos como la introducción de matrices, tablas para interpolaciones, etc. . . se cubren más adelante.

```
# -*- coding: latin-1 -*-
#!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
#                          NO TOCAR
from modelo import Modelo
from matematicas import *
from numarray import *
from util import *
from integrador import *

import logging

modelo = Modelo()
#!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

6.6.2. Nombre

El nombre del helicóptero no es transcendente, sirve solo para mostrar por pantalla ciertos mensajes haciendo referencia al helicóptero. Como se ve el nombre del helicóptero es una cadena de texto, por lo que debe ir entre comillas dobles o comillas simples:

```
modelo.nombre = "Lynx"
```

6.6.3. Integrador

El integrador a utilizar. Los integradores disponibles se encuentran disponibles en el fichero `integrador.py`. Para asignar el integrador se escribe el nombre y se acompaña de dos paréntesis al final. Los integradores disponibles son:

- Euler
- AdamsBashforth2
- RungeKutta4
- AdamsBashforth3
- ABM2

```
modelo.integrador = AdamsBashforth2()
```

6.6.4. Datos másicos del helicóptero

La masa del helicóptero en kilogramos y los momentos de inercia en kilogramos por metro al cuadrado. Se supone que el helicóptero es simétrico respecto al plano xz por lo que la matriz de inercia es de la forma:

$$I = \begin{bmatrix} I_{xx} & 0 & -I_{xz} \\ 0 & I_{yy} & 0 \\ -I_{xz} & 0 & I_{zz} \end{bmatrix} \tag{6.1}$$

```
modelo.M = 4313.7
modelo.Ixx = 2767.1
modelo.Iyy = 13904.5
modelo.Izz = 12208.8
modelo.Ixz = 2034.8
```

6.6.5. Rotor principal

Posición y orientación del rotor

- `gas` (γ_s): Es la inclinación del rotor positiva hacia delante. Debe darse en radianes. En este caso se sabe que la inclinación es de cuatro grados por lo que se utiliza la función `rad` que pasa de radianes a grados. Alternativamente se podría haber calculado aparte el ángulo en radianes e introducirlo directamente.
- `hR` (h_R): Distancia vertical del rotor al centro de masas, en metros. Evidentemente positiva con el rotor por encima del centro de masas.
- `xcg` (x_{cg}): Distancia horizontal del rotor al centro de masas. Positiva cuando el rotor queda por detrás del centro de masas. Unidades en metros.
- `h0` (h_0): Distancia desde el rotor hasta el suelo en metros.

```
modelo.rotor.gas = rad(4)
modelo.rotor.hR = 1.274
modelo.rotor.xcg = -0.0198
modelo.rotor.h0 = 2.946
```

Descripción geométrica y másica del rotor

- `Nb` (N_b): Número de palas del rotor. Actualmente las ecuaciones del rotor sólo son correctas para cuatro palas.
- `R` (R): Radio del rotor en metros.
- `c` (c): Cuerda de los perfiles del rotor en metros.
- `Ibe` (I_β): Momento de inercia en kilogramos por metro cuadrado de cada una de las palas del rotor respecto al eje de batimiento.

- `Kbe` (K_β): Muelle equivalente para el batimiento de la pala, en Newtons por radián. Se puede obtener a partir de la primera frecuencia de batimiento.
- `tht` (θ_t): Torsión lineal de la pala en radianes, por tanto torsión en la punta de la pala. Es decir, se supone que el paso de los perfiles de la pala debido a la torsión es del tipo:

$$\theta = \theta_t \frac{r}{R} \tag{6.2}$$

```
modelo.rotor.Nb = 4
modelo.rotor.R = 6.4
modelo.rotor.c = 0.391
modelo.rotor.Ibe = 678.14
modelo.rotor.Kbe = 166352
modelo.rotor.tht = -0.14
```

Descripción aerodinámica de los perfiles de la pala

- `a0` (a_0): pendiente de sustentación de los perfiles. Se supone simétrico, lineal e independiente del número de Mach. Evidentemente como si depende del número de Mach se selecciona la pendiente de sustentación de una sección característica de la pala, por ejemplo la sección $\frac{3}{4}$.
- `de0` (δ_0)
- `de1` (δ_1)
- `de2` (δ_2): Términos de la polar de resistencia del perfil (adimensionales), no en función del ángulo de ataque, sino en función del coeficiente de sustentación del rotor principal. Es decir, el coeficiente de sustentación de todos los perfiles del rotor se supone de la forma:

$$\delta = \delta_0 + \delta_1 c_T + \delta_2 c_T^2 \tag{6.3}$$

  Se pueden calcular a partir de la polar clásica dada en función del ángulo de ataque de forma que el par del rotor sea el mismo.

```
modelo.rotor.a0 = 6.0
modelo.rotor.de0 = 0.009
modelo.rotor.de1 = 0.0
modelo.rotor.de2 = 37.983
```

6.6.6. Rotor de cola

Posición y orientación

- `hT` (h_T): distancia vertical en metros del rotor de cola al centro de masas. Positiva con el rotor de cola por encima del centro de masas.
- `lT` (l_T): distancia horizontal entre el rotor principal y el rotor de cola, en metros.

- `K` (K): inclinación del rotor de cola en radianes. Positivo según un giro por el eje $-x$.

```
modelo.rotor_cola.hT = 1.146
modelo.rotor_cola.lT = 7.66
modelo.rotor_cola.K = 0.0
```

Descripción geométrica y másica

- `RT` (R_T): radio del rotor de cola en metros.
- `cT` (c_T): cuerda del rotor de cola en metros.
- `IbeT` (I_{β_T}): momento de inercia en kilogramos por metro cuadrado de cada pala del rotor de cola respecto a su eje de batimiento.
- `KbeT` (K_{β_T}): muelle equivalente del rotor de cola, en newtons por radián.
- `k3` (k_3): Factor de acoplamiento entre el ángulo de batimiento y el ángulo de paso. Adimensional.

$$\theta = k_3 \beta \tag{6.4}$$
$$k_3 = \tan \delta_3 \tag{6.5}$$
$$\tag{6.6}$$

- `tht` (θ_{t_T}): torsión lineal del rotor de cola.
- `gT` (g_T): relación entre la velocidad angular del rotor de cola y del rotor principal. Adimensional.

$$g_T = \frac{\Omega_T}{\Omega} \tag{6.7}$$

```
modelo.rotor_cola.RT = 1.106
modelo.rotor_cola.cT = 0.18001
modelo.rotor_cola.IbeT = 1.08926
modelo.rotor_cola.KbeT = 2511.24
modelo.rotor_cola.k3 = -1.
modelo.rotor_cola.tht = 0.0
modelo.rotor_cola.gT = 5.8
```

Descripción aerodinámica

- `a0T` (a_{0_T}): pendiente de sustentación de los perfiles del rotor de cola, con las mismas hipótesis del rotor principal.
- `de0T`
- `de1T`

- `de2T`: coeficientes de la polar de resistencia de los perfiles del rotor de cola, en función del coeficiente de sustentación del rotor de cola:

$$\delta = \delta_{0_T} + \delta_{1_T} c_{T_T} + \delta_{2_T} c_{T_T}^2 \tag{6.8}$$

```
modelo.rotor_cola.a0T = 6.0
modelo.rotor_cola.de0T = 0.008
modelo.rotor_cola.de1T = 0.0
modelo.rotor_cola.de2T = 5.334
```

6.6.7. Fuselaje

Descripción geométrica

- `Ss` (S_s): Área lateral de fuselaje en metros cuadrados.
- `Sp` (S_p): Área frontal de fuselaje en metros cuadrados. Tanto S_s como S_p no necesitan ser los valores exactos, sino tan sólo los valores de superficie que se han utilizado para adimensionalizar los coeficientes c_Y, c_l, c_m y c_D, c_L, c_m respectivamente.
- `lf` (l_f): longitud del fuselaje en metros. De nuevo basta con que sea la longitud utilizada en la adimensionalización de los coeficientes de momento.
- `xca` (x_{ca}): posición horizontal del centro aerodinámico del fuselaje, positivo por delante del centro de masas. En metros.
- `zca` (z_{ca}): posición vertical del centro aerodinámico, positivo debajo del centro de masas. En metros.
- `ht` (h_t): distancia del centro de masas al suelo.

```
modelo.fuselaje.Ss = 32.
modelo.fuselaje.Sp = 24.
modelo.fuselaje.lf = 12.
modelo.fuselaje.xca = 0.139
modelo.fuselaje.zca = 0.190
modelo.fuselaje.ht = 1.67
```

Coeficientes aerodinámicos del fuselaje

La descripción aerodinámica del fuselaje es la que presenta la estructura más compleja de todo el archivo del modelo. Para empezar existen dos métodos de describir el fuselaje, denominados FuselajeA y FuselajeB. Para utilizar cada uno de ellos se utiliza una de las dos opciones siguientes:

```
modelo.fuselaje.aero = FuselajeA(
        ...
)
```

o bien:

```
modelo.fuselaje.aero = FuselajeB
        ...
)
```

FuselajeB es el método con la sintaxis más sencilla y el que presenta mayor generalidad ya que sólo requiere que se le pase una función válida de python que acepte como argumentos el ángulo de ataque y el ángulo de resbalamiento del fuselaje y devuelve una tupla con los 6 coeficientes aerodinámicos: c_D, c_L, c_Y, c_l, c_m, c_n (en este orden). Evidentemente es un método que requiere un conocimiento más profundo de programación en python. Por ello, y porque a veces se tiene un conjunto reducido de datos sobre el fuselaje, se proporciona el modelo de fuselaje A, que requiere únicamente la sustitución de los valores numéricos de un fichero de ejemplo. Aunque para el helicóptero Lynx se utiliza el modelo de fuselaje A aquí se muestra también el ejemplo del BlackHawk:

```
def aero(al, be):
        if al<-pi/2:
                al += pi
        elif al>pi/2:
                al -= pi
        if be<-pi/2:
                be += pi
        elif be>pi/2:
                be -= pi

        Sal, S2al, Sbe, S2be, S4be =\
                map(sin, [al, 2*al, be, 2*be, 4*be])
        Cal, C2al, Cbe, C2be, C4be =\
                map(cos, [al, 2*al, be, 2*be, 4*be])

        cD = 1.3944*(Sal)**2 - 0.6435*Cal + 2.4806 +\
             0.0456*C4be - 1.597*C2be -\
             8.2893e-9*(be**4)
        cL = 0.4546*Sal + 0.6730*S2al -\
             1.2680*(Sal)**2 - 1.3029*Cal + 1.3215 +\
             0.00127*be - 0.0465*S4be +\
             0.0005*(be**2)
        cY = -0.0802*Sbe -0.1627*S2be + 0.0182*S4be

        if be<=rad(10):
                cl = 0.0
        elif rad(10)<be and be<=rad(25):
                cl = sign(be)*0.02098*(Cbe**4) - 0.0197
        else:
                cl = 0.0283*Sbe +\
                     sign(be)*(0.0022*C4be -\
                     0.0134*(Cbe)**3 +\
```

```
                    0.0338*(Cbe)**4 - 0.0308)

        cm = 0.0007 + 0.2291*S2al + 0.1343*(Sal)**2 +\
             0.1095*Cal - 0.1606*(Cbe)**3 + 0.0176

        if be<=rad(20):
                cn = 0.0128*S2be - 0.0195*S4be -\
                     8.4339e-5
        else:
                cn = -0.0101*S2be +\
                     sign(be)*(0.0309*(C4be**4) - 0.0197)

        return cD, cL, cY, cl, cm, cn

modelo.fuselaje.aero = FuselajeB(aero)
```

A continuación se muestra el modelo de fuselaje tal como se usa en el modelo Lynx. La sintaxis es un poco más complicada, pero como ventajas presenta que no requiere saber programar en python y además a partir de pocos datos aerodinámicos calcula (de forma muy aproximada) los coeficientes aerodinámicos para todo ángulo de ataque. La sintaxis es de la forma:

```
modelo.fuselaje,aero = FuselajeA(
        nombre1 = valor1,
        nombre2 = valor2,
        ...
        nombreN = valorN
)
```

Y el listado tal como aparece en el fichero del Lynx es, parte a parte:

Declaración del modelo de fuselaje

```
modelo.fuselaje.aero = FuselajeA(
```

Unidades

Puede ser que los datos que se poseen del fuselaje den los coeficientes en función del ángulo de ataque en grados. Como pasar dichos valores a radianes puede ser muy pesado, se da la opción de introducir esos valores en grados o radianes, con tal de que se indique en el fichero del modelo. En este caso se utilizan grados, por lo que el valor que se le asigna `unidades` es una cadena de texto llamada `deg`. Hay que notar que es una cadena de texto, por lo que debe ir entre comillas dobles o simples.

```
        unidades = 'deg',
```

Si se hubiesen utilizado radianes se habría indicado:

```
unidades = 'rad',
```

Valores de los coeficientes para ángulos de 90 grados

```
cDa90 = 0.3480,
cDb90 = 0.4784,
cm90 =  0.03,
cl90 =  0.01,
cn90 =  0.1,
```

Rango de los ángulos de ataque y resbalamiento para los cuales se utilizan valores precisos

Fuera de este rango se aproxima el valor de los coeficientes utilizando los valores que anteriormente se han introducido para 90 grados. Como utilizamos ángulos en grados, estos valores van en grados.

```
al1 = -21,
al2 =  21,

be1 = -21,
be2 =  21,
```

Valores de los coeficientes para ángulos pequeños

La sintaxis es:

```
cDa = Lista(
        angulo1, valor1,
        angulo2, valor2,
        ...
),
cDb = Lista(
        ...
),
cm = Lista(
        ...
),
cL = Lista(
        ...
),
cl = Lista(
        ...
),
cY = Lista(
        ...
```

```
        ),
        cn = Lista(
                ...
        ),
```

donde:

- `cDa` (c_{D_α}): coeficiente de resistencia para ángulo de resbalamiento nulo, en función de ángulo de ataque.
- `cDa` (c_{D_β}): coeficiente de resistencia para ángulo de ataque nulo, en función de ángulo de resbalamiento.
- `cm` (c_m): coeficiente de momento de cabeceo en función de ángulo de ataque.
- `cL` (c_L): coeficiente de sustentación en función de ángulo de ataque.
- `cl` (c_l): coeficiente de momento de balance, en función de ángulo de resbalamiento.
- `cY` (c_Y): coeficiente de fuerza lateral, en función de ángulo de resbalamiento.
- `cn` (c_n): coeficiente de momento de guiñada, en función de ángulo de resbalamiento.

Y a continuación el listado tal como se encuentra en el fichero del modelo del Lynx. De nuevo expresamos los ángulos en grados tal como se especificó en `unidades`. También notamos la presencia de un paréntesis al final que cierra el paréntesis que se abrió con `FuselajeA(`

```
        cDa = Lista(
                -21, 0.06661,
                -18, 0.06010,
                -15, 0.05531,
                -12, 0.04991,
                 -9, 0.04707,
                 -6, 0.04561,
                 -3, 0.04421,
                  0, 0.04233,
                  3, 0.04245,
                  6, 0.04372,
                  9, 0.04486,
                 12, 0.04685,
                 15, 0.04926,
                 18, 0.05170,
                 21, 0.05613
        ),

        cDb = Lista(
                -21, 0.05532,
```

```
                -18, 0.04096,
                -15, 0.02863,
                -12, 0.01842,
                 -9, 0.01040,
                 -6, 0.00464,
                 -3, 0.00116,
                  0, 0.00000,
                  3, 0.00116,
                  6, 0.00464,
                  9, 0.01040,
                 12, 0.01842,
                 15, 0.02863,
                 18, 0.04096,
                 21, 0.05532
        ),

        cm = Lista(
                -21, -0.02415,
                -18, -0.02210,
                -15, -0.02015,
                -12, -0.01803,
                 -9, -0.01588,
                 -6, -0.01351,
                 -3, -0.01074,
                  0, -0.00747,
                  3, -0.00421,
                  6, -0.00085,
                  9,  0.00237,
                 12,  0.00573,
                 15,  0.00875,
                 18,  0.01197,
                 21,  0.01433
        ),

        cL = Lista(
                -21, -0.02415,
                -18, -0.02210,
                -15, -0.02015,
                -12, -0.01803,
                 -9, -0.01588,
                 -6, -0.01351,
                 -3, -0.01074,
                  0, -0.00747,
                  3, -0.00421,
                  6, -0.00085,
                  9,  0.00237,
                 12,  0.00573,
                 15,  0.00875,
                 18,  0.01197,
                 21,  0.01433
```

```
        ),

        cY = Lista(
                -21,  0.17547,
                -18,  0.15138,
                -15,  0.12677,
                -12,  0.10178,
                 -9,  0.07653,
                 -6,  0.05110,
                 -3,  0.02557,
                  0,  0.00000,
                  3, -0.02557,
                  6, -0.05110,
                  9, -0.07653,
                 12, -0.10178,
                 15, -0.12677,
                 18, -0.15138,
                 21, -0.17547
        ),

        cl = Lista(
                -21, -0.00697,
                -18, -0.00472,
                -15, -0.00270,
                -12, -0.00097,
                 -9,  0.00000,
                 -6,  0.00000,
                 -3,  0.00000,
                  0,  0.00000,
                  3,  0.00000,
                  6,  0.00000,
                  9,  0.00000,
                 12,  0.00097,
                 15,  0.00270,
                 18,  0.00472,
                 21,  0.00697
        ),

        cn = Lista(
                -21, -0.01704,
                -18, -0.01461,
                -15, -0.01217,
                -12, -0.00974,
                 -9, -0.00730,
                 -6, -0.00487,
                 -3, -0.00243,
                  0,  0.00000,
                  3,  0.00243,
                  6,  0.00487,
                  9,  0.00730,
```

```
                12,  0.00974,
                15,  0.01217,
                18,  0.01461,
                21,  0.01704
        )
)
```

6.6.8. Estabilizador horizontal

Posición y orientación

- `ltp` (l_{tp}): distancia horizontal desde el rotor al estabilizador horizontal en metros.
- `htp` (h_{tp}): distancia vertical desde el centro de masas al estabilizador horizontal, positiva con el estabilizador horizontal por encima del centro de masas, en metros.
- `S_{tp}` (S_{tp}): Área del estabilizador horizontal en metros cuadrados.
- `altp0` (α_{tp_0}): Ángulo de ataque del estabilizador horizontal, con helicóptero sin ángulo de ataque. En radianes.

```
modelo.estabilizador_horizontal.ltp = 7.66
modelo.estabilizador_horizontal.htp = 1.146
modelo.estabilizador_horizontal.Stp = 1.197
modelo.estabilizador_horizontal.altp0 = rad(-1.0)
```

Propiedades aerodinámicas

De la misma forma que con el fuselaje tenemos varias posibilidades para elegir: perfilA, perfilB, perfilC. La más general es el modelo de perfil C, ya que permite especificar las dos funciones que para todo ángulo de ataque devuelven los coeficientes de resistencia y sustentación. Su sintaxis es:

```
        modelo.estabilizador_horizontal.aero = perfilC(
                        cL = funcion_cL,
                        cD = funcion_cD
        )
```

Donde `funcion_cL` y `funcion_cD` son funciones escritas en python que aceptan un único valor, el ángulo de ataque y devuelven un único valor, el coeficiente que corresponda.

Para el fichero del modelo Lynx utilizamos el modelo de perfil A, que es el recomendado. El significado de los parámetros que intervienen es:

- `AR` (A): alargamiento del estabilizador.
- `a` (a): pendiente de sustentación.

- `cLmax` ($c_{L_{max}}$): máximo coeficiente de sustentación.
- `de0` (δ_0)
- `de1` (δ_1)
- `de2` (δ_2): coeficientes de la polar del perfil, para obtener el coeficiente de resistencia del perfil en función del ángulo de ataque:

$$c_D = \delta_0 + \delta_1\alpha + \delta_2\alpha^2 \tag{6.9}$$

```
modelo.estabilizador_horizontal.aero = perfilA(
                AR  =  2.7,
                a   =  2.3663,
                cLmax = 0.6,
                de0 =  1.065e-3,
                de1 = -8.4703e-2,
                de2 =  1.46981
)
```

Si no se obtienen datos suficientes del perfil se puede indicar únicamente el alargamiento y el programa calculará automáticamente el resto de parámetros. Por ejemplo, si hubiésemos hecho:

```
modelo.estabilizador_horizontal.aero = perfilA(
                AR  =  2.7,
```

los otros parámetros habrían sido calculados automáticamente.

6.6.9. Controles

- `c0` ($\vec{c}_0$): pasos del rotor para controles nulos.
- `c1` (c_1): sensibilidad y mezcla de los controles.
- `S0` (S_0)
- `S1` (S_1)
- `S2` (S_2): matrices para el sistema de control automático.
- `th_0` (θ_0): cabeceo marcado por el sistema de control. Se puede especificar como una constante o como una función del mando longitudinal.
- `fi_0` (ϕ_0): balance marcado por el sistema de control. Se puede especificar como una constante o como una función del mando lateral.
- `ch_0` (ψ_0): guiñada marcada por el sistema de control. Se puede especificar como una constante o como una función de los pedales.
- `L` (L): límite del sistema automático de control.

```
modelo.controles.c0 = array([
              0.059,
              0.000,
              0.000,
             -0.061,
            ], type='Float64')


modelo.controles.c1 = array([
            [ 0.300, 0.000,  0.000,  0.000 ],
            [ 0.000, 0.200,  0.000,  0.000 ],
            [ 0.000, 0.000,  0.200,  0.000 ],
            [ 0.441, 0.000,  0.000,  0.100 ],
            ], type='Float64')

modelo.controles.S0 = array([
    [ 1.0000, 0.0000, 0.0000, 0.0000 ],
    [ 0.0000, 0.0000, 0.0000, 0.0000 ],
    [ 0.0000, 0.0000, 0.0000, 0.0000 ],
    [ 0.0000, 0.0000, 0.0000, 1.0000 ]], type='Float64')

modelo.controles.S1 = array([
    [ 0.0000,  0.0000, 0.0000 ],
    [ 0.0000, -2.5000, 0.0000 ],
    [ 2.5000,  0.0000, 0.0000 ],
    [ 0.0000,  0.0000, 0.6000 ]], type='Float64')

modelo.controles.S2 = array([
    [  0.0000,  0.0000, 0.0000 ],
    [ -2.5000,  0.0000, 0.0000 ],
    [  0.0000,  2.5000, 0.0000 ],
    [  0.0000,  0.0000, 0.0000 ]], type='Float64')

modelo.controles.th_0 =  lambda et1sp:  0.0603 + 1.4*et1sp
modelo.controles.fi_0 =  lambda et1cp: -0.0476 - 1.4*et1cp
modelo.controles.ch_0 =  0.0

modelo.controles.L = 0.2
```

Para introducir funciones se recuerda que puede ser cualquier tipo de codigo en python válido, y también se pueden especificar mediante listas, de forma que se podría haber escrito (y hubiese sido equivalente):

```
modelo.controles.th_0 = Lista( 0,  0.0603,
                               1,  1.4603 )
modelo.controles.fi_0 = Lista( 0, -0.0476,
                               1, -1.4476 )
```

6.6.10. Motor

- `n` (n): Número de motores.
- `Omi` (Ω_i): Velocidad nominal de rotación del rotor.
- `Wto` (W_{to_0}): Máxima potencia de despegue a nivel del mar.
- `Wmc` (W_{mc_0}): Máxima potencia continua a nivel del mar.
- `x` (x): Coeficiente de variación de potencia disponible con la altura.
- `Wid` (W_{id}): Mínima potencia de operación del motor.
- `K_3` (K_3): rigidez del motor.
- `P` (P): Pérdidas del sistema de transmisión del rotor y rotor de cola.
- `Irt` (I_R): Inercia del sistema de transmisión en función del número de motores activos.
- `tae1` (τ_1): Primera constante de tiempos del motor.
- `a2` (a_2)
- `b2` (b_2)
- `c2` (c_2): Parámetros para la segunda constante de tiempos del motor τ_2.
- `a3` (a_3)
- `b3` (b_3)
- `c3` (c_3): Parámetros para la tercera constante de tiempos del motor τ_3.

```
modelo.motor.Omi = 35.63

modelo.motor.Wto_0 = 746000
modelo.motor.Wmc_0 = 664000
modelo.motor.x     = 0.85
modelo.motor.Wid   = 1000

modelo.motor.K3 = 65e3

modelo.motor.n = 2
modelo.motor.P  = 0.1

modelo.motor.Irt = [
                3085.069,
                3344.716,
                3344.716]

modelo.motor.tae1 = 0.1
```

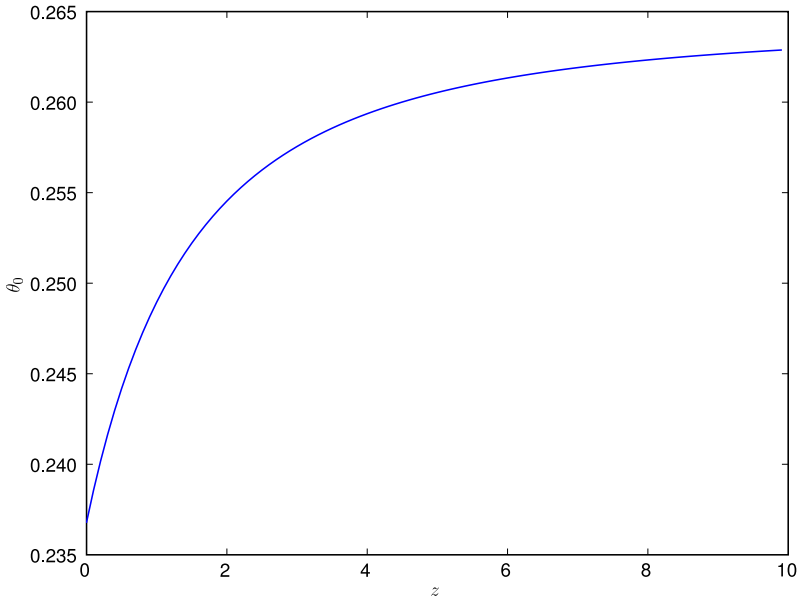


Figura 6.13: Colectivo en función de la altura

```
modelo.motor.a2 = 0.1
modelo.motor.b2 = 0.1
modelo.motor.c2 = 0.1

modelo.motor.a3 = 0.1
modelo.motor.b3 = 0.1
modelo.motor.c3 = 0.1
```

6.7. Trimado

A continuación se comentan resultados de trimado del Lynx, a partir de los cuales se comenta sobre la validez del modelo.

6.7.1. Vuelo a punto fijo

A continuación se presenta la posición de colectivo necesaria para mantener vuelo a punto fijo a diferentes alturas sobre el suelo y la variación del resto de los controles con el el colectivo.

Como se puede ver en la gráfica 6.13, el colectivo (en radianes) varía con la altura (en metros) debido al efecto suelo. Para valores mayores al radio del rotor el efecto suelo apenas se nota. La verdadera distancia del rotor al suelo es la altura de la figura (z) más la altura del fuselaje ($h_0 = 2{,}496m$)

Para determinar los valores de las ganancias de los mandos calculamos la

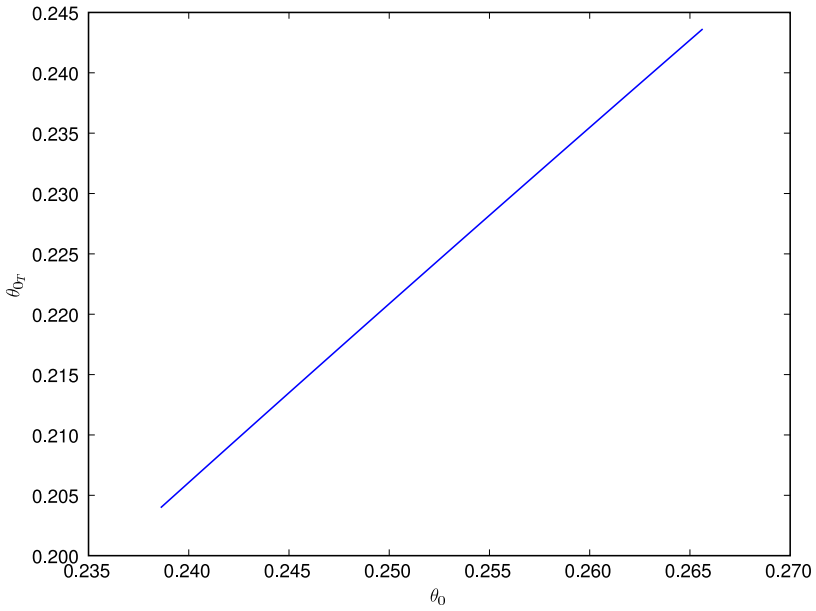


Figura 6.14: Colectivo de cola en función de colectivo

variación de θ_{0_T}, θ_{1c} y θ_{1s} con el colectivo. Como podemos ver θ_{1s} no es exactamente cero debido al pequeño desplazamiento del centro de masas respecto al centro del rotor, por lo que necesita compensar el ligero momento que se produce. El aún más pequeño valor de θ_{1c} es debido al acoplamiento lateral-longitudinal de la respuesta de del helicóptero, o dicho de otro modo, debido a que la frecuencia de batimiento de las palas no es exactamente la de giro del rotor, el desfase entre el paso y el batimiento no es exactamente de 90^o, por lo que se produce ese pequeño acoplamiento.

6.7.2. Efecto suelo

A continuación se presenta la variación del efecto suelo para diferentes velocidades, de 0 a 25m/s, y alturas, de 0 a 10m. Como se puede ver, al aumentar la velocidad disminuye en general el requerimiento de colectivo pero también los beneficios del efecto suelo. Para los primeros 10 m/s ambos efectos se cancelan, lo cual es afortunado porque permite realizar una transición sencilla desde punto fijo a vuelo en avance en las cercanías del suelo.

6.7.3. Vuelo vertical

A continuación (figura 6.17) se muestran resultados para diferentes velocidades ascensionales $V_a(m/s)$, en ausencia de efecto suelo. Se puede apreciar en la gráfica de la velocidad inducida frente a velocidad ascensional como al haber aplicado la TCMM tenemos una transición suave desde el régimen de vuelo a punto fijo hasta molinete frenante, pasando por autorrotación.

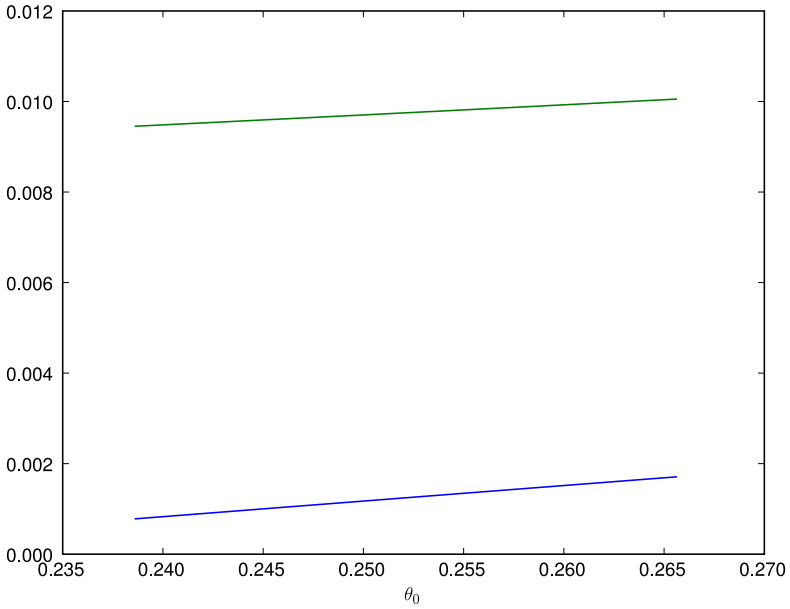


Figura 6.15: Cíclico en función de colectivo

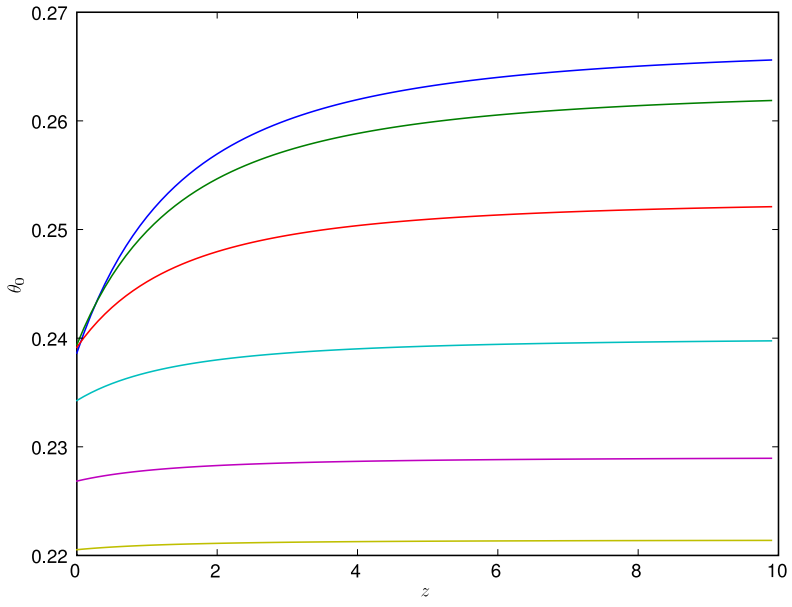


Figura 6.16: Colectivo en función de velocidad y altura

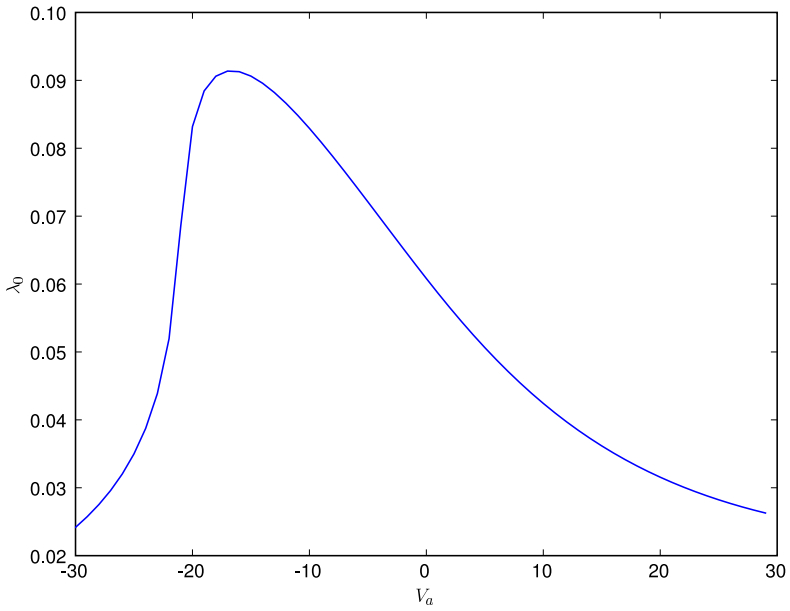


Figura 6.17: Velocidad inducida frente a velocidad ascensional

Para el vuelo vertical la expresión del coeficiente de sustentación del rotor es:

$$c_T = \frac{a_0 s}{2}\left[\frac{\theta_0}{3} + \frac{\mu_z - \lambda_0}{2} + \frac{\theta_t}{4}\right]$$

Aproximadamente permanece c_T constante, siendo la pequeña variación debida a la resistencia del fuselaje, por lo que θ_0 varía proporcionalmente a $\mu_z - \lambda_0$, como se ve en la figura.

Se puede despejar también θ_{0_T} en función de θ_0, pero la relación es bastante más complicada, lo importante es que como muestra la figura la relación no es lineal.

6.7.4. Vuelo en avance

Por último, el régimen mas común en la operación del helicóptero. Si no se especifica nada se asume que los ángulos se encuentran en radianes y la velocidad en nudos. Si todavía había esperanzas de encontrar unas ganancias de controles sencillas que nos mantuviesen el helicóptero cercano al trimado se desvanecen ahora, ya que observando la posición del colectivo frente a la velocidad (figura 6.20) vemos que hay el mismo colectivo para diferentes velocidades. El motivo es que según aumenta la velocidad del helicóptero aumenta la eficiencia del rotor al empezar a comportarse éste de forma similar a un ala, por lo que para un mismo c_T disminuye la necesidad de θ_0, por lo que hay una tendencia inicial de θ_0 a bajar con la velocidad. Sin embargo, la resistencia del fuselaje y del rotor aumenta también con la velocidad, de forma que esta tendencia domina para

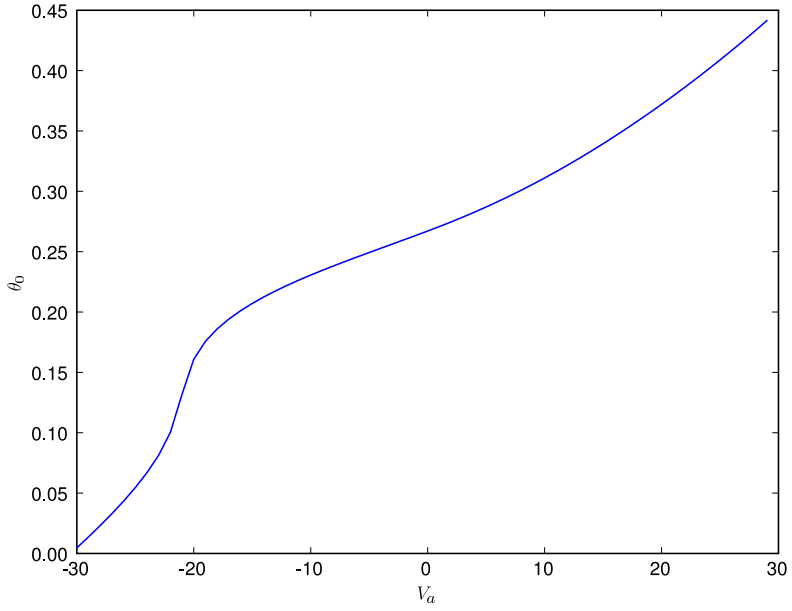


Figura 6.18: Colectivo frente a velocidad ascensional

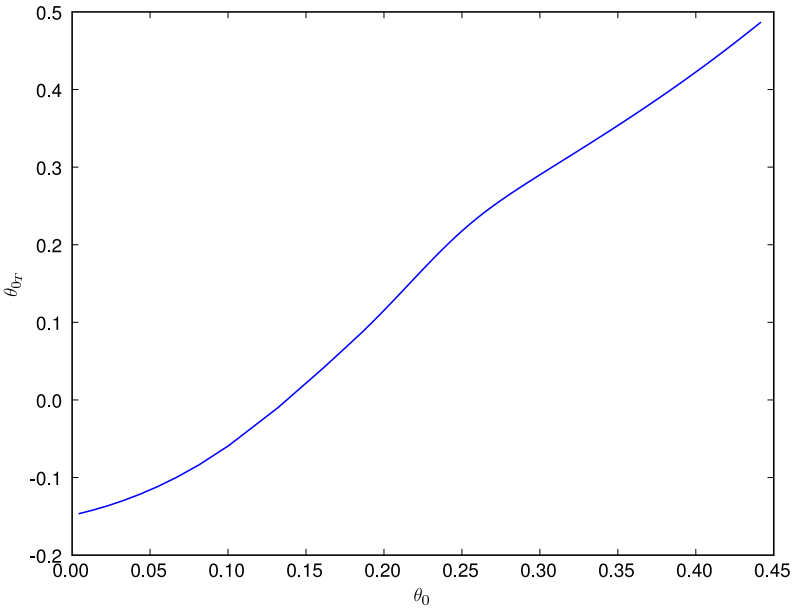


Figura 6.19: Colectivo de cola frente a colectivo para diferentes velocidades ascensionales

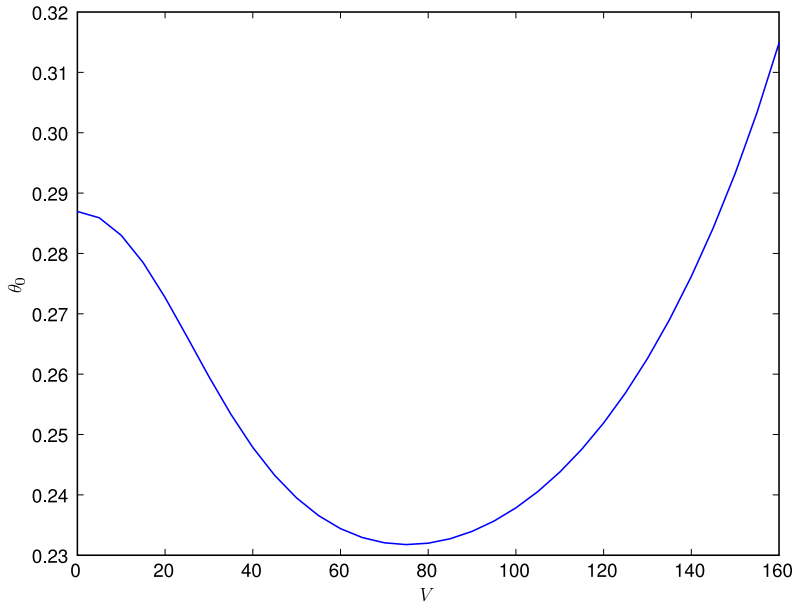


Figura 6.20: Colectivo para diferentes velocidades de avance

velocidades altas, pero en algún punto entre velocidad nula y velocidad alta se alcanza el mínimo.

Siempre es conveniente comprobar que la velocidad inducida adopta la típica forma. En este caso vemos también como gracias al modelo de estela se predice correctamente la velocidad no uniforme, ya que es aproximadamente:

$$\lambda_{1c} = \lambda_0 \tan \frac{|\chi|}{2}$$

Donde χ es el ángulo de la estela. Ver figura 6.21.

El otro control que más influye sobre el vuelo en avance es el cíclico longitudinal, debido a que es necesario mantener inclinado el helicóptero hacia adelante para que la tracción del rotor compense la resistencia del fuselaje. El cabeceo θ necesario queda muy bien aproximado entonces por $\theta = -\frac{D}{Mg}$, por lo que, como se ve en la figura 6.22, tiene una forma parabólica.

Debido a la inestabilidad del momento del fuselaje, al aumentar el cabeceo aumenta el momento aerodinámico y por tanto tiene que aumentar el batimiento longitudinal β_{1c} para compensarlo; el aumento de este batimiento debido al desfase cercano a 90º de paso y batimiento se advierte sobre todo en el cíclico θ_{1s}. θ_{1c} no varía demasiado y permanece pequeño gracias a que a pesar de que el incremento de tracción del rotor de cola produce un aumento de momento de balance, éste se encuentra aproximadamente equilibrado por el momento que se produce en el rotor de forma natural al aumentar la diferencia de velocidades entre la pala que avanza y la pala que retrocede (figura 6.23).

Vemos cómo el ángulo de balance ϕ sirve para compensar la fuerza lateral de la cola, siendo su variación exactamente opuesta (figura 6.24).

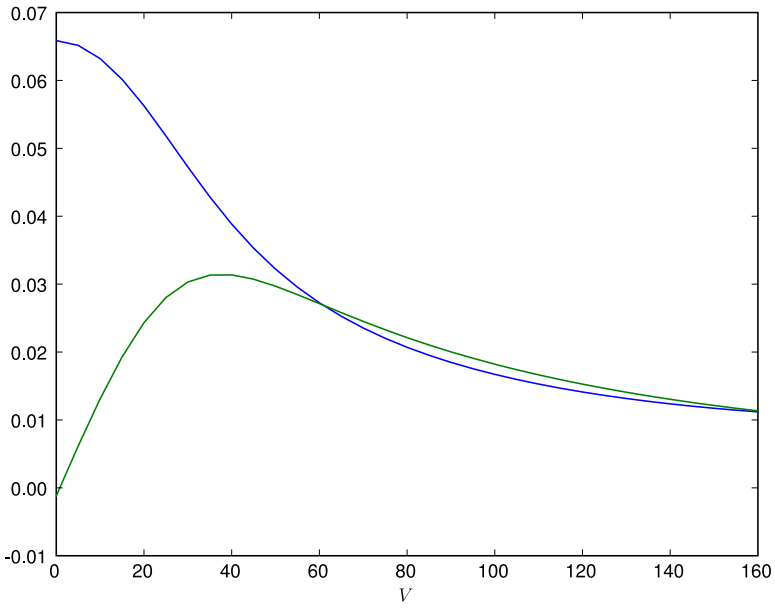


Figura 6.21: λ_0 y λ_{1c} en función de velocidad de avance

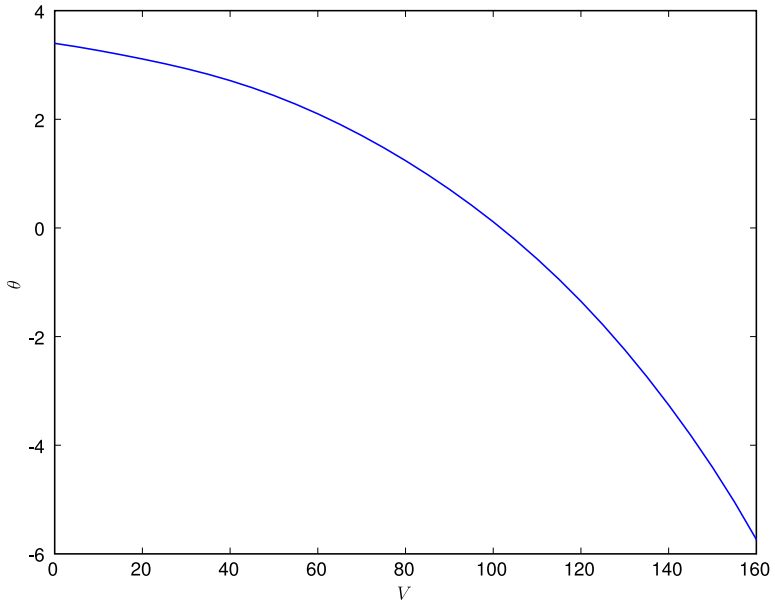


Figura 6.22: Cabeceo en grados para diferentes velocidades de avance

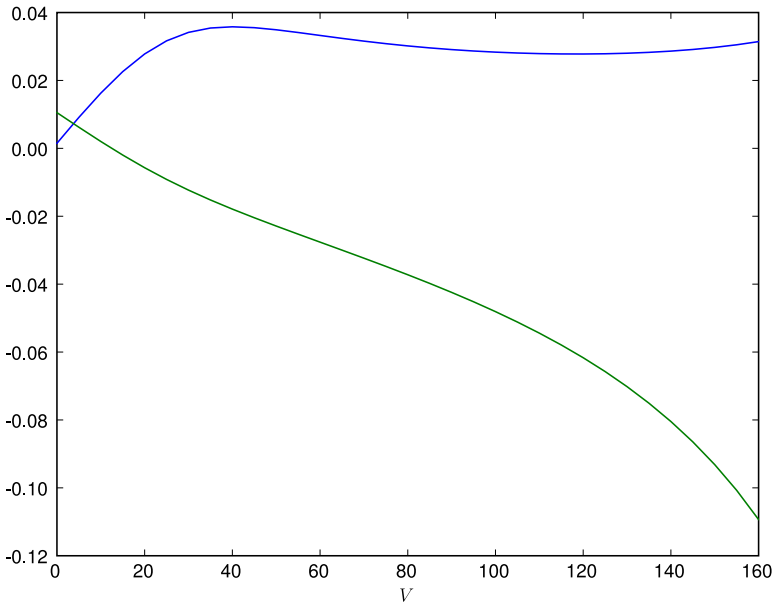


Figura 6.23: Cíclicos para diferentes velocidades de avance

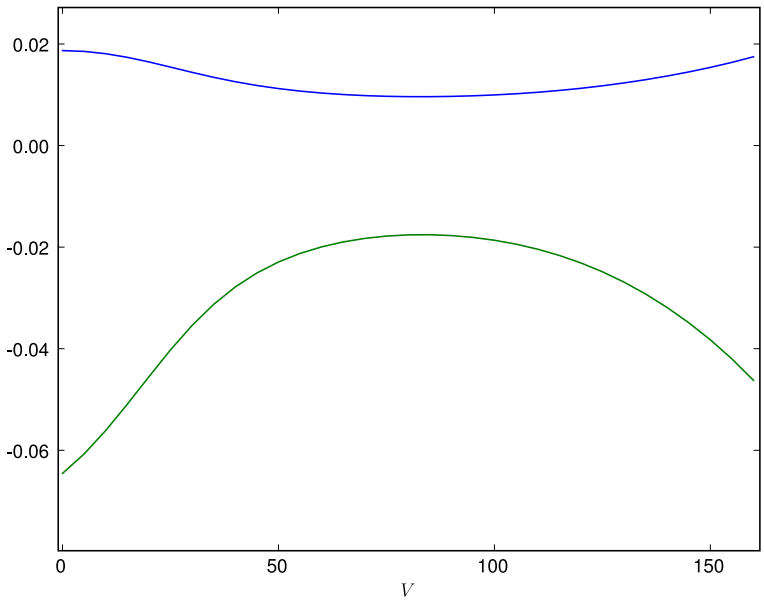


Figura 6.24: Tracción de cola y ángulo de balance frente a velocidad de avance

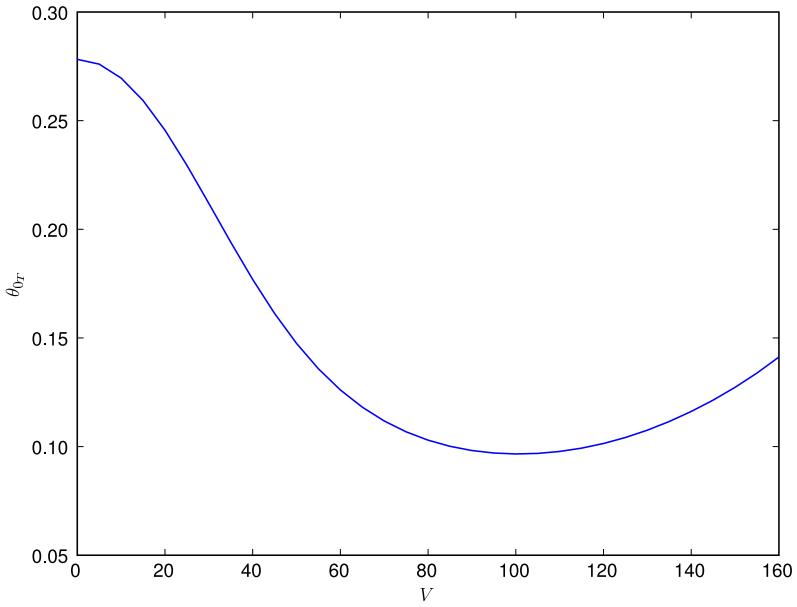


Figura 6.25: Colectivo de cola frente a velocidad de avance

Por último, se muestra el colectivo de cola frente a la velocidad de avance. Esta curva está determinada por dos factores: el comportamiento similar al colectivo del rotor principal frente a la velocidad de avance y la necesidad de variar su tracción como muestra la gráfica de c_{T_T} para compensar el par del rotor principal (figura 6.25).

6.7.5. Giro a nivel sin resbalamiento

Para que no haya resbalamiento debe compensar la aceleración centrífuga a la gravedad por lo que, como vemos en la figura 6.26, es aproximadamente:

$$\phi = \arctan \frac{\Omega_a V}{g}$$

El colectivo tiene que compensar por la pérdida de tracción vertical al inclinar el helicóptero y el colectivo de cola a su vez compensa el par de rotor debido al aumento de colectivo.

6.8. Derivadas de estabilidad

A continuación se presenta de forma gráfica el cálculo de las derivadas de estabilidad a partir de cada situación de trimado. En concreto se ha perturbado en la velocidad, la velocidad angular y los controles un valor de ±0,001 y se ha calculado la perturbación correspondiente en cada una de las tres fuerzas y tres momentos que actúan sobre el helicóptero. Las fuerzas se encuentran divididas

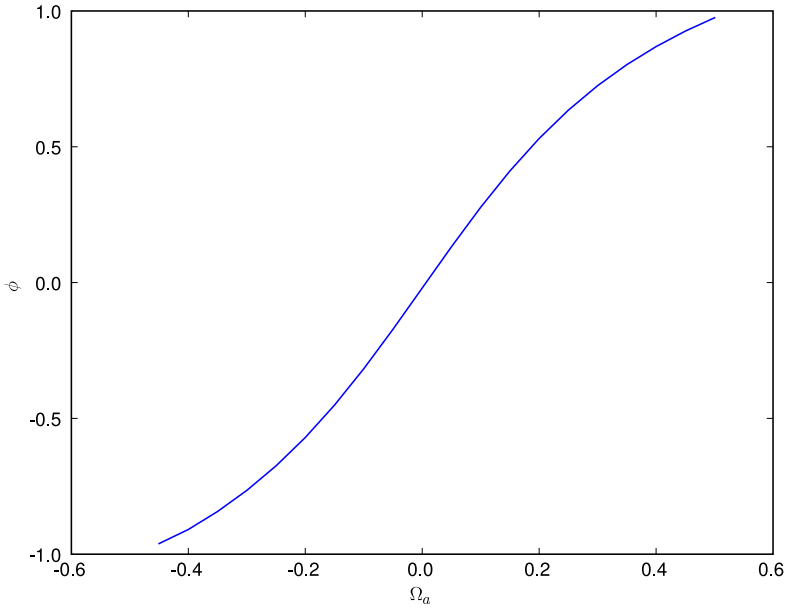


Figura 6.26: Balance en función de velocidad de giro

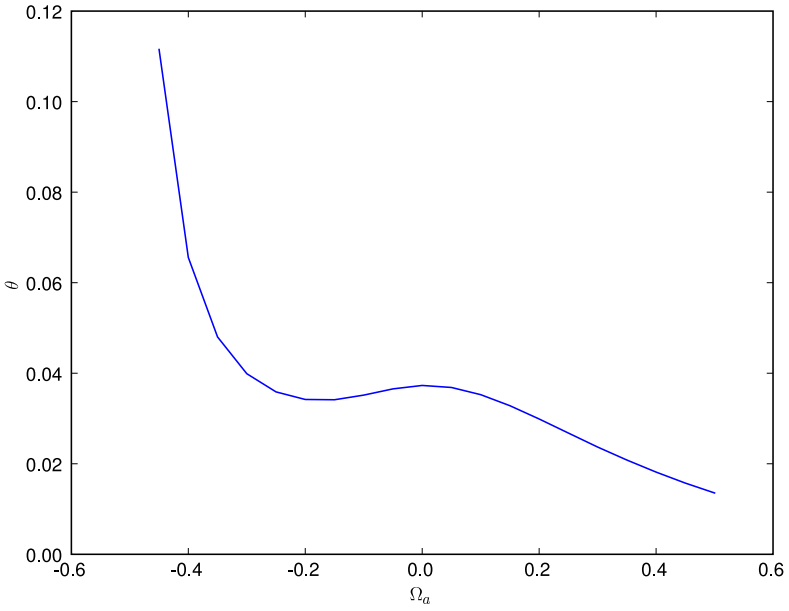


Figura 6.27: Cabeceo en función de velocidad de giro

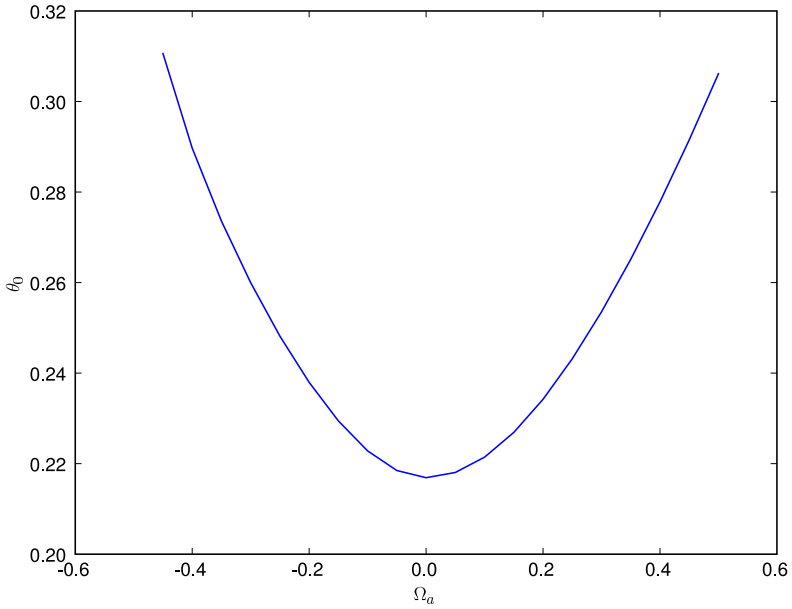


Figura 6.28: Colectivo en función de velocidad de giro

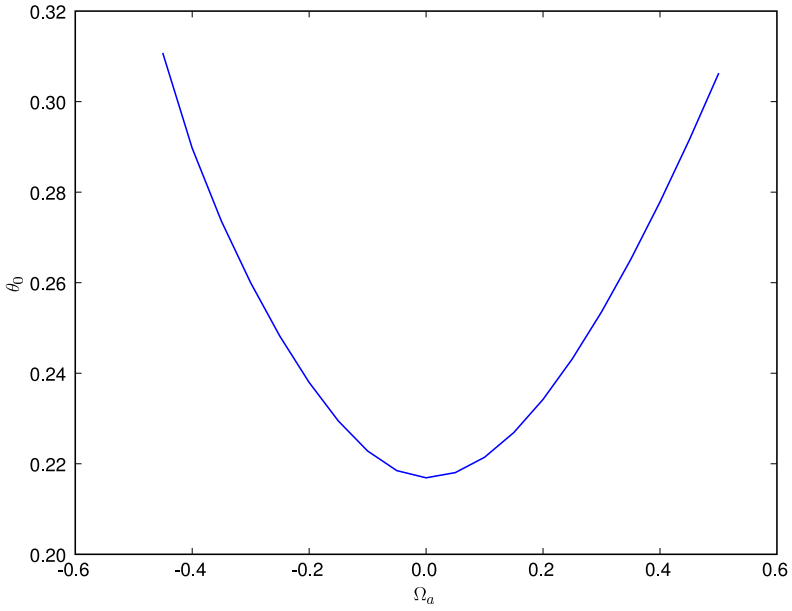


Figura 6.29: Colectivo de cola en función de velocidad de giro

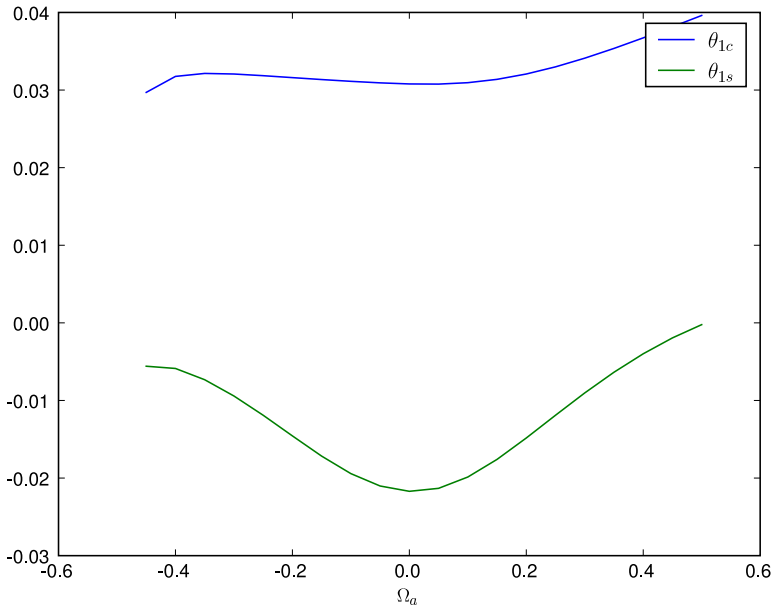


Figura 6.30: Cíclicos en función de velocidad de giro

por la masa del helicóptero y los momentos por la inercia, teniendo en cuenta el efecto de los momentos de inercia cruzados. De formas más exacta, si la prima indica la magnitud antes de casi-adimensionalizar:

$$X = \frac{X'}{M} \left[\frac{m}{s^2}\right] \tag{6.10}$$

$$Y = \frac{Y'}{M} \left[\frac{m}{s^2}\right] \tag{6.11}$$

$$Z = \frac{Z'}{M} \left[\frac{m}{s^2}\right] \tag{6.12}$$

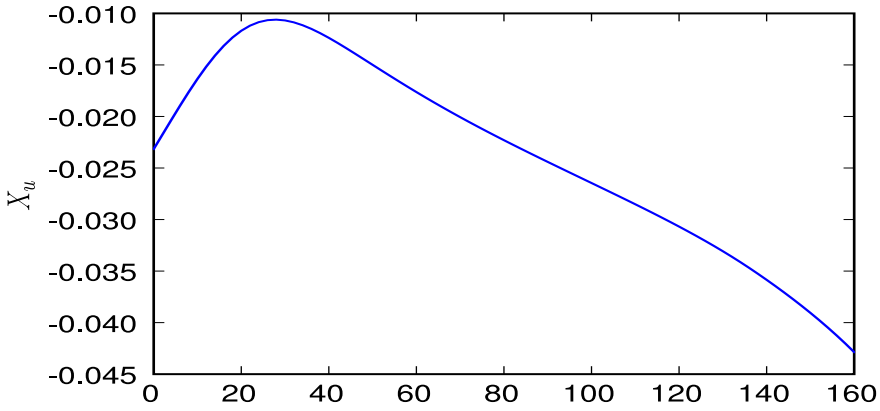
$$L = \frac{I_{zz}}{I_{xx}I_{zz} - I_{xz}^2} L' + \frac{I_{xz}}{I_{xx}I_{zz} - I_{xz}^2} N' \left[\frac{1}{s^2}\right] \tag{6.13}$$

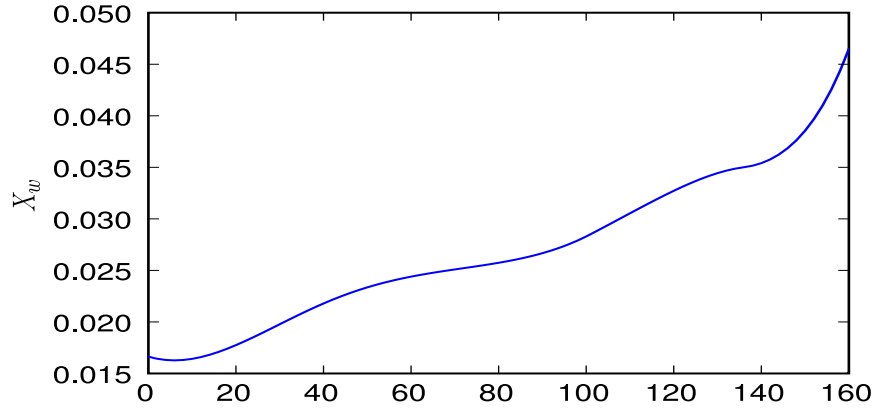
$$M = \frac{M'}{I_{yy}} \left[\frac{1}{s^2}\right] \tag{6.14}$$

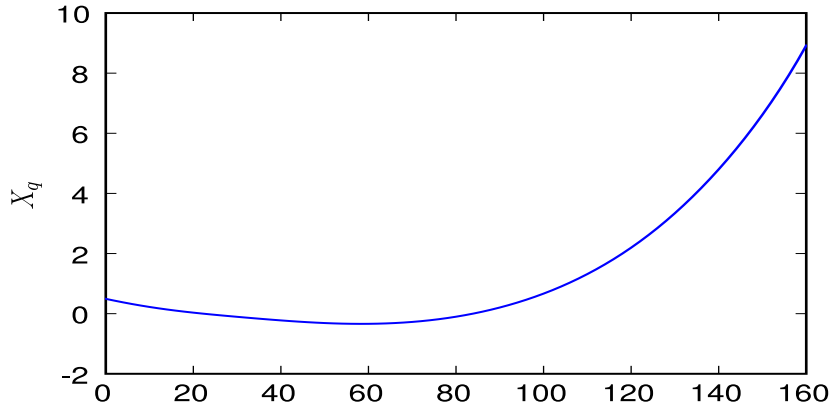
$$N = \frac{I_{xz}}{I_{xx}I_{zz} - I_{xz}^2} L' + \frac{I_{xx}}{I_{xx}I_{zz} - I_{xz}^2} N' \left[\frac{1}{s^2}\right] \tag{6.15}$$

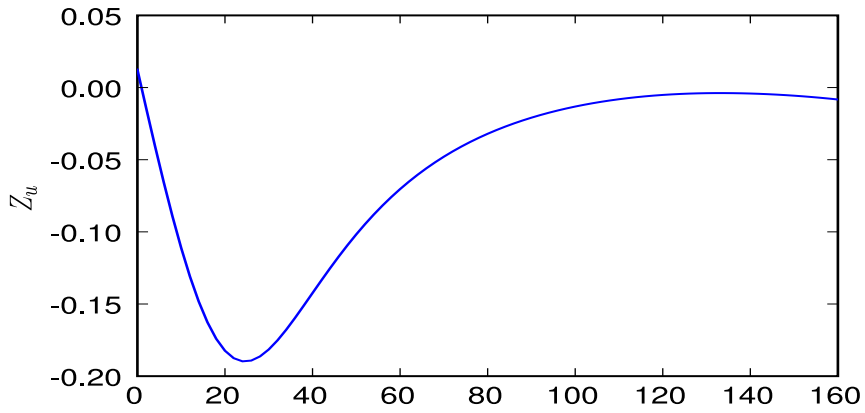
$$\tag{6.16}$$

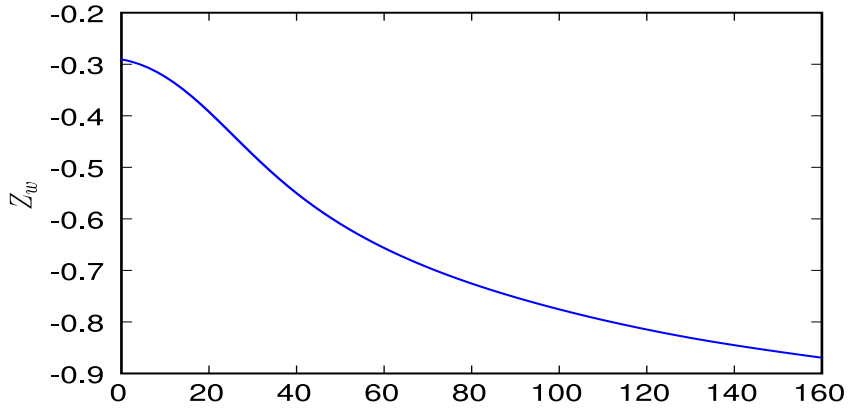
Las derivadas se encuentran representadas frente a la velocidad, dada en nudos. De 0 a 2 nudos se han calculado cada 0.2 nudos, y de 2 nudos hasta 160 cada 2 nudos.

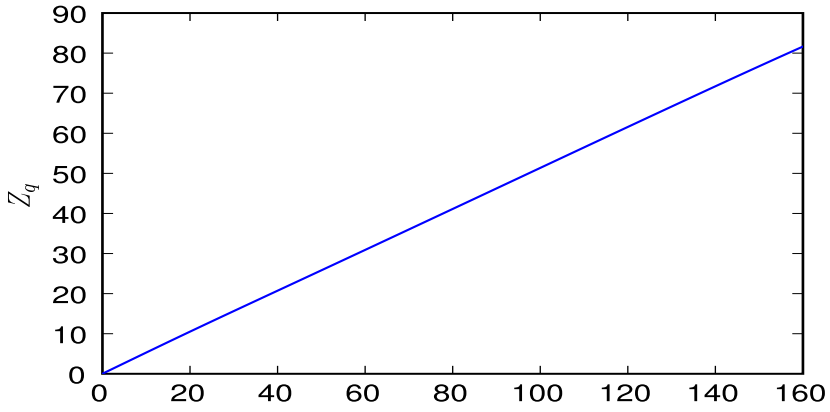

-0.010
-0.015
-0.020
-0.025
-0.030
-0.035
-0.040
-0.045
X_u
0
20
40
60
80
100
120
140
160

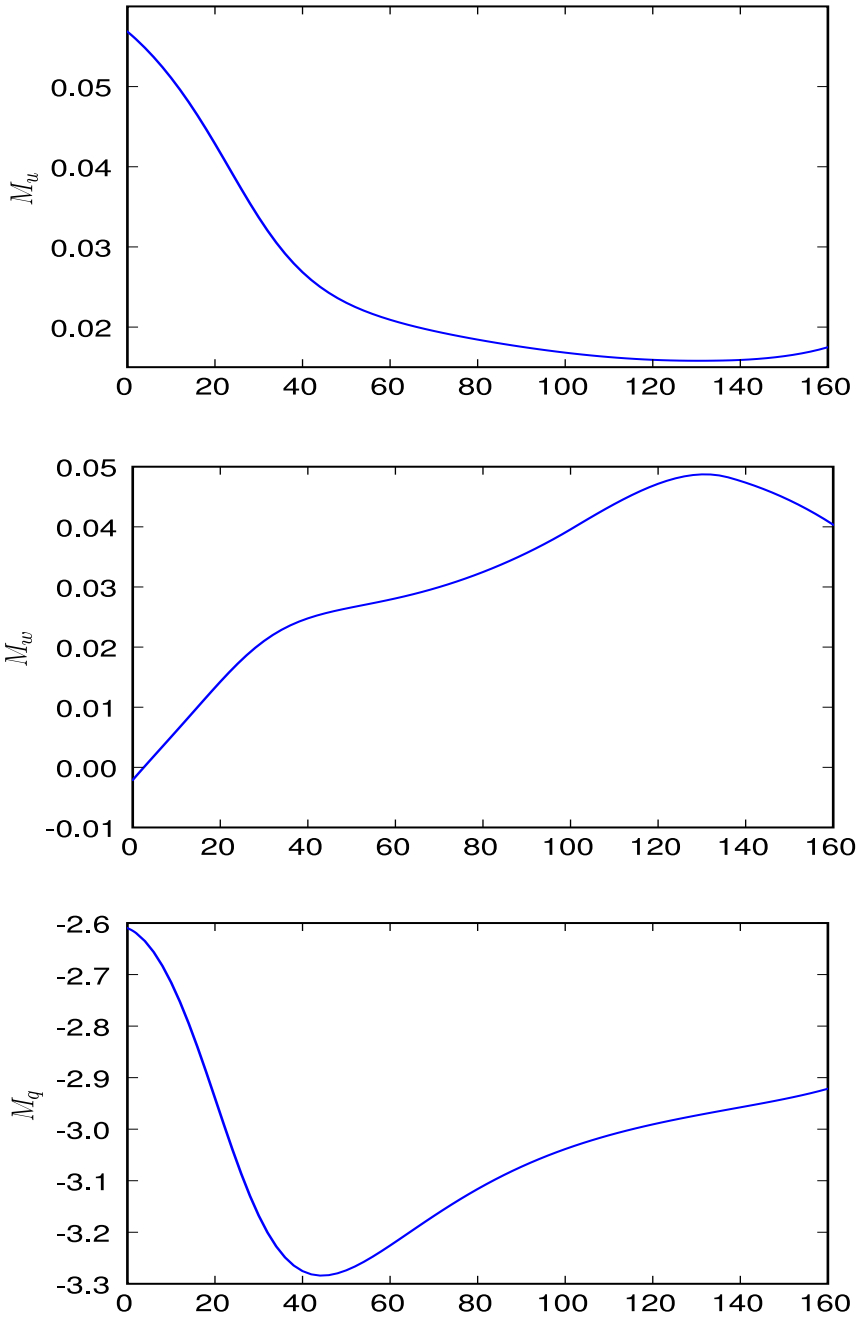

0.050
0.045
0.040
0.035
0.030
0.025
0.020
0.015
X_w
0
20
40
60
80
100
120
140
160

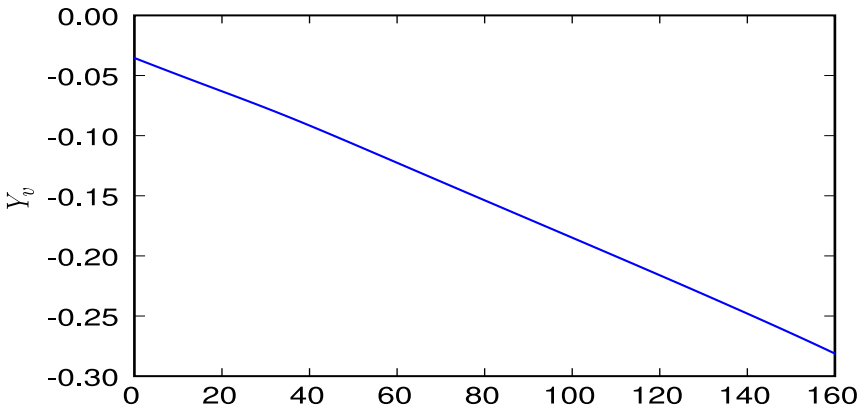

10
8
6
4
2
0
-2
X_q
0
20
40
60
80
100
120
140
160

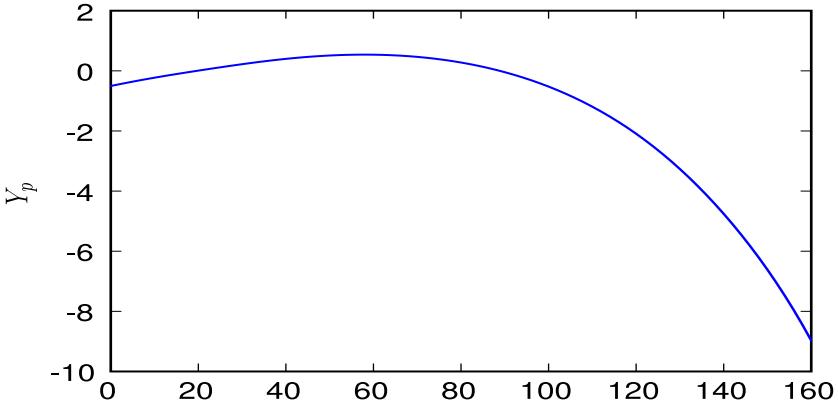

0.05
0.00
-0.05
-0.10
-0.15
-0.20
Z_u
0
20
40
60
80
100
120
140
160

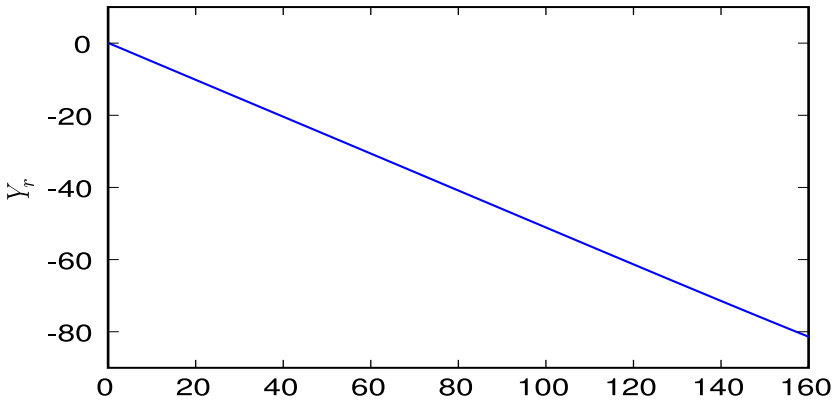

-0.2
-0.3
-0.4
-0.5
-0.6
-0.7
-0.8
-0.9
Z_w
0
20
40
60
80
100
120
140
160

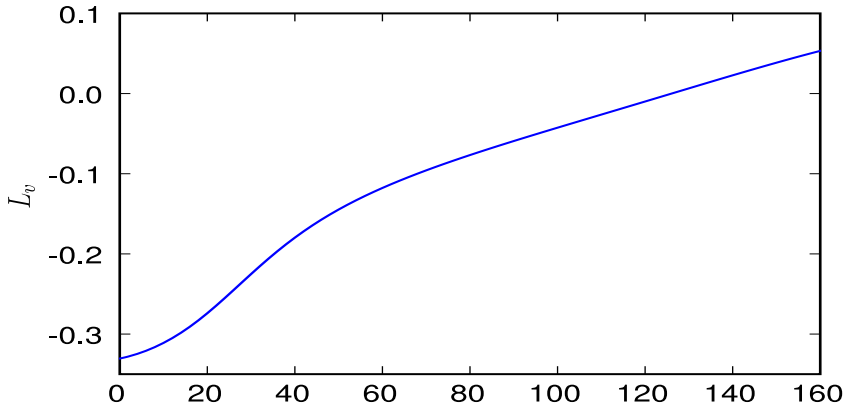

90
80
70
60
50
40
30
20
10
0
Z_q
0
20
40
60
80
100
120
140
160

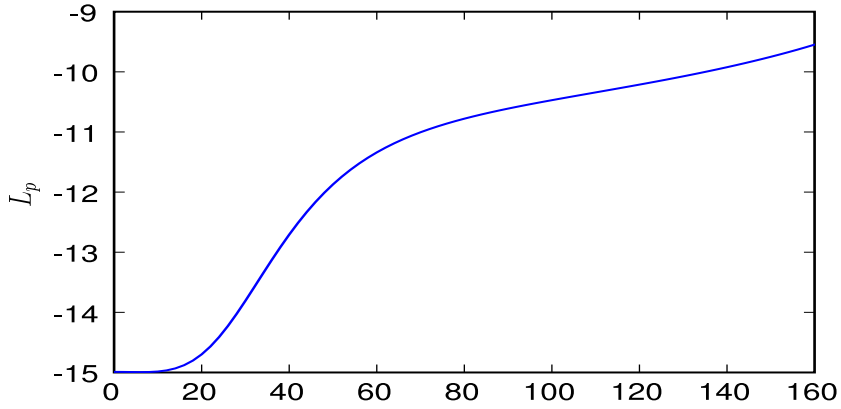

M_u
0.05
0.04
0.03
0.02
0
20
40
60
80
100
120
140
160
M_w
0.05
0.04
0.03
0.02
0.01
0.00
-0.01
0
20
40
60
80
100
120
140
160
M_q
-2.6
-2.7
-2.8
-2.9
-3.0
-3.1
-3.2
-3.3
0
20
40
60
80
100
120
140
160

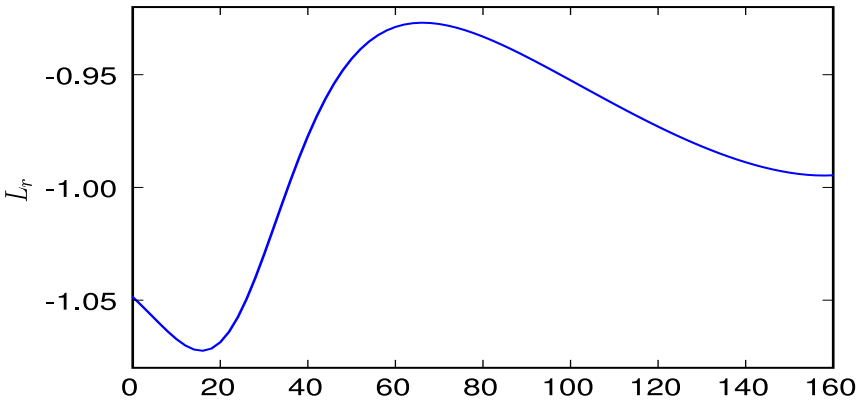

0.00
-0.05
-0.10
-0.15
-0.20
-0.25
-0.30
Y_v
0
20
40
60
80
100
120
140
160

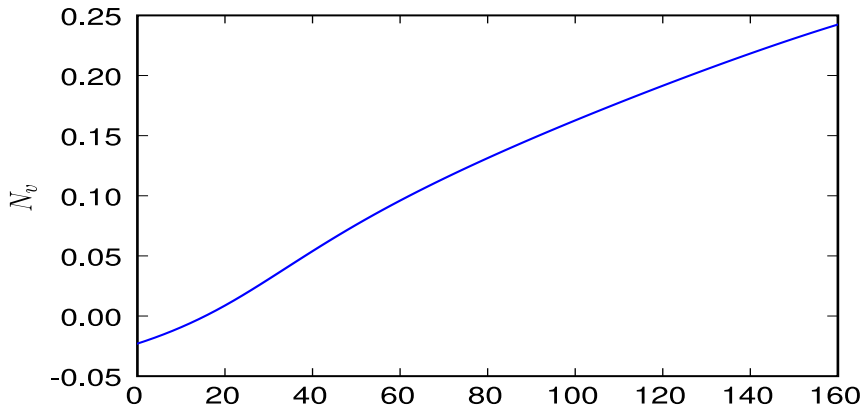

2
0
-2
-4
-6
-8
-10
Y_p
0
20
40
60
80
100
120
140
160

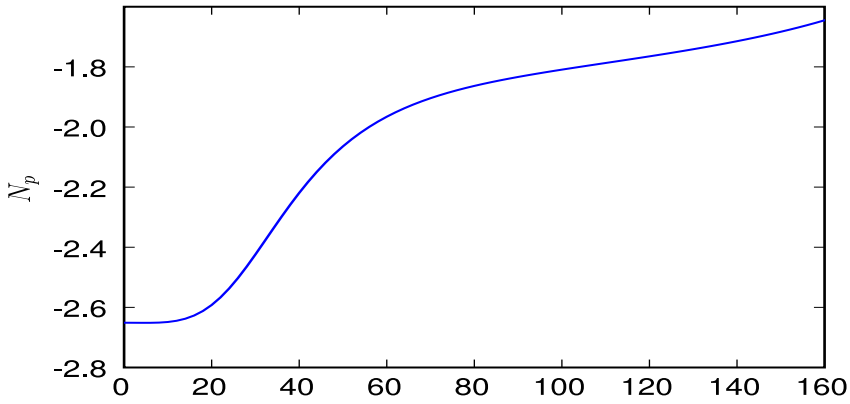

0
-20
-40
-60
-80
Y_r
0
20
40
60
80
100
120
140
160

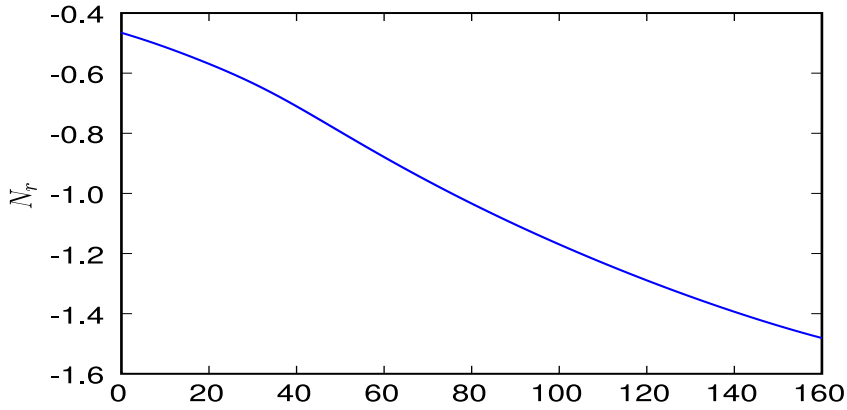

0.1
0.0
-0.1
-0.2
-0.3
L_v
0
20
40
60
80
100
120
140
160

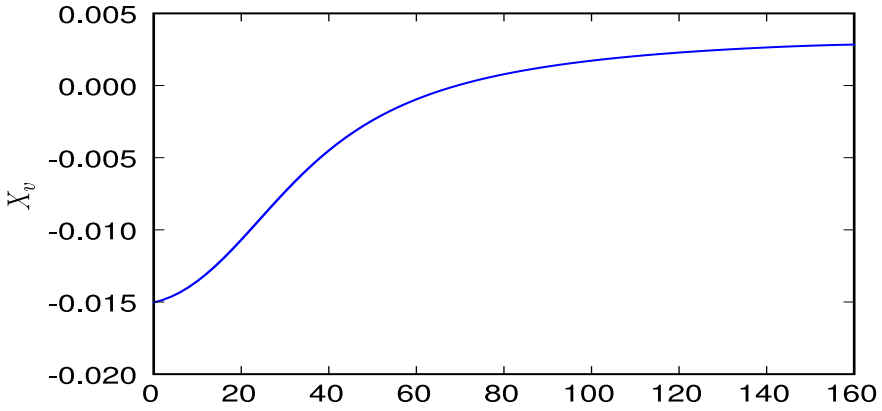

-9
-10
-11
-12
-13
-14
-15
L_p
0
20
40
60
80
100
120
140
160

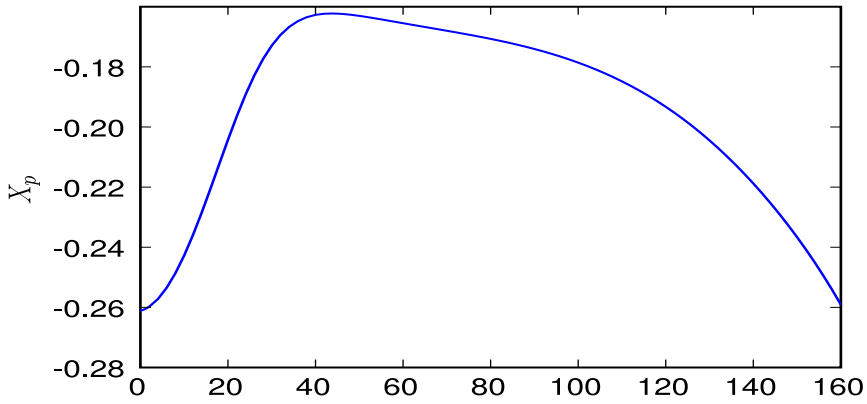

-0.95
-1.00
-1.05
L_r
0
20
40
60
80
100
120
140
160

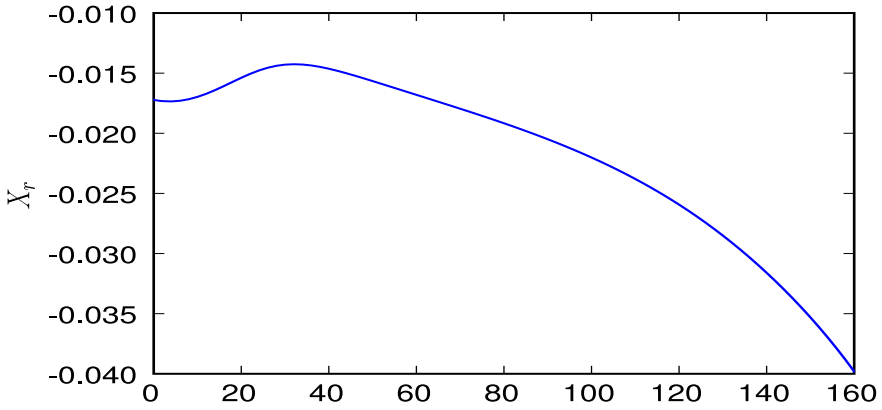

0.25
0.20
0.15
0.10
0.05
0.00
-0.05
N_v
0
20
40
60
80
100
120
140
160

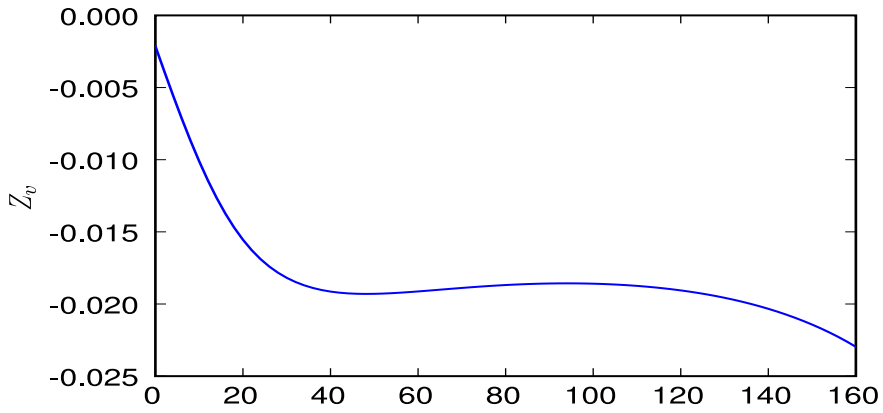

-1.8
-2.0
-2.2
-2.4
-2.6
-2.8
N_p
0
20
40
60
80
100
120
140
160

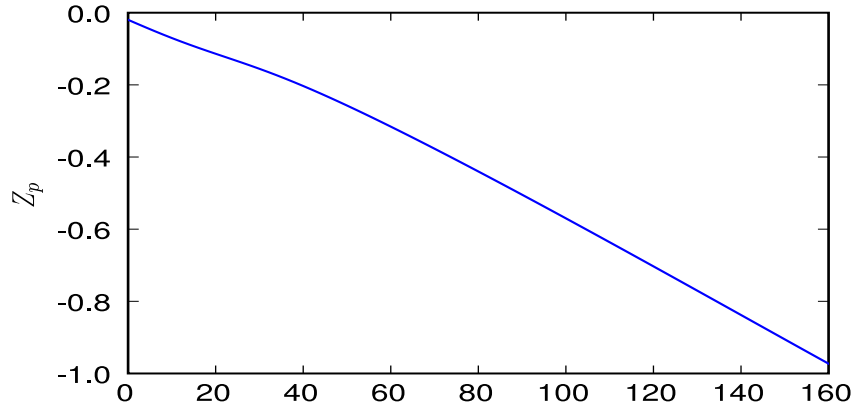

-0.4
-0.6
-0.8
-1.0
-1.2
-1.4
-1.6
N_r
0
20
40
60
80
100
120
140
160

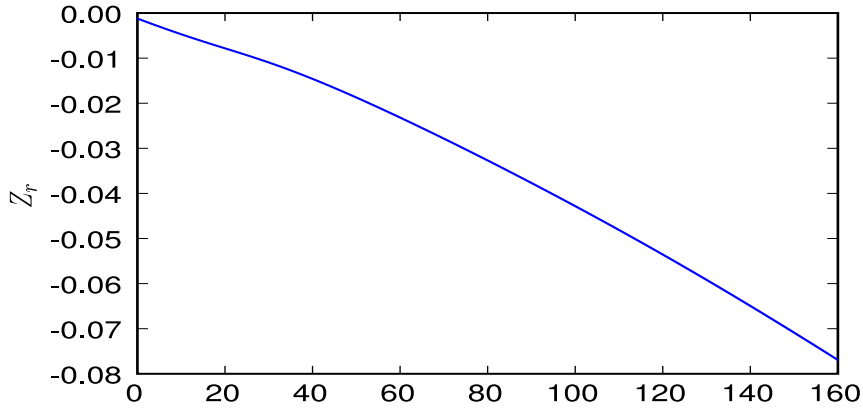

0.005
0.000
-0.005
-0.010
-0.015
-0.020
X_v
0
20
40
60
80
100
120
140
160

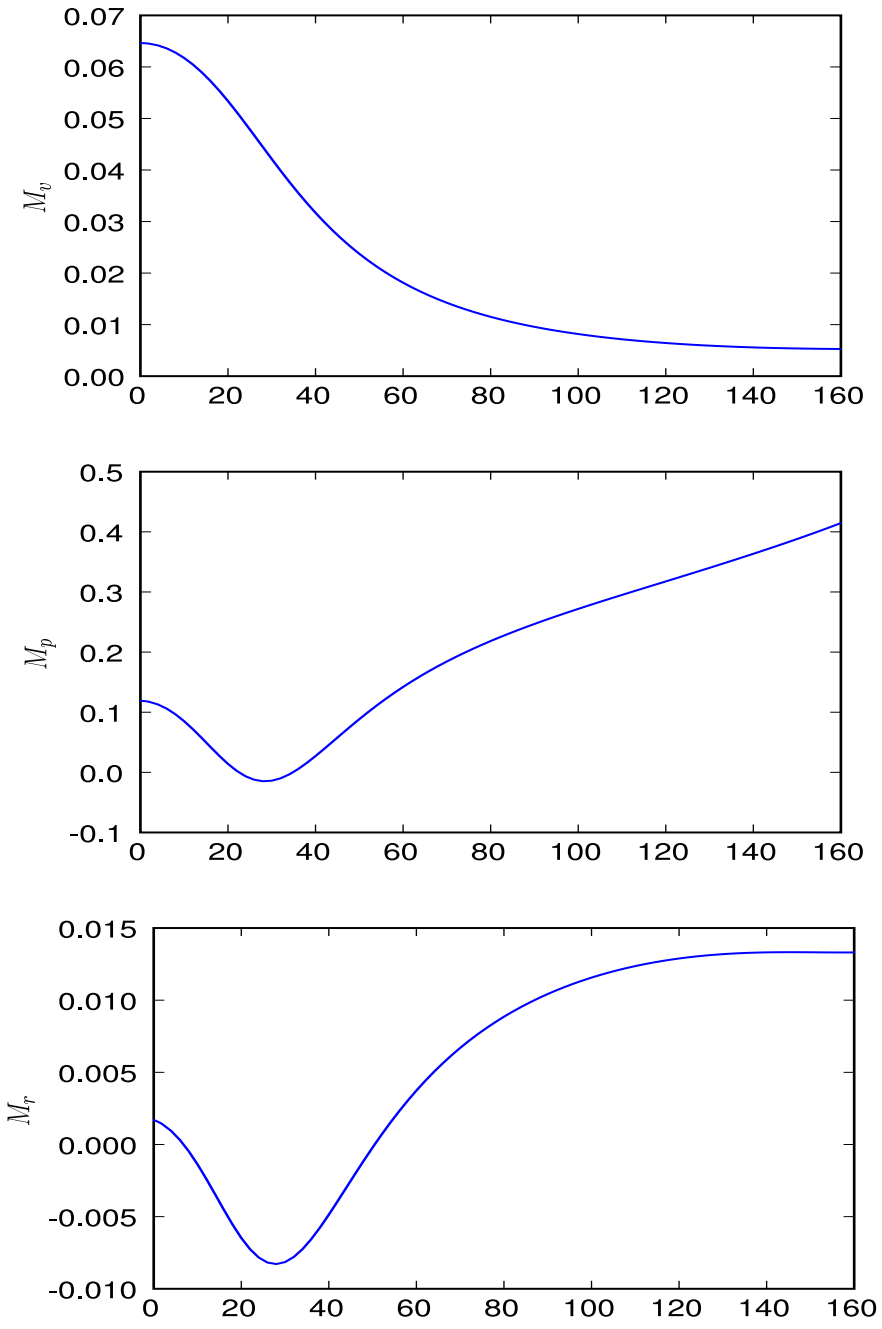

-0.18
-0.20
-0.22
-0.24
-0.26
-0.28
X_p
0
20
40
60
80
100
120
140
160

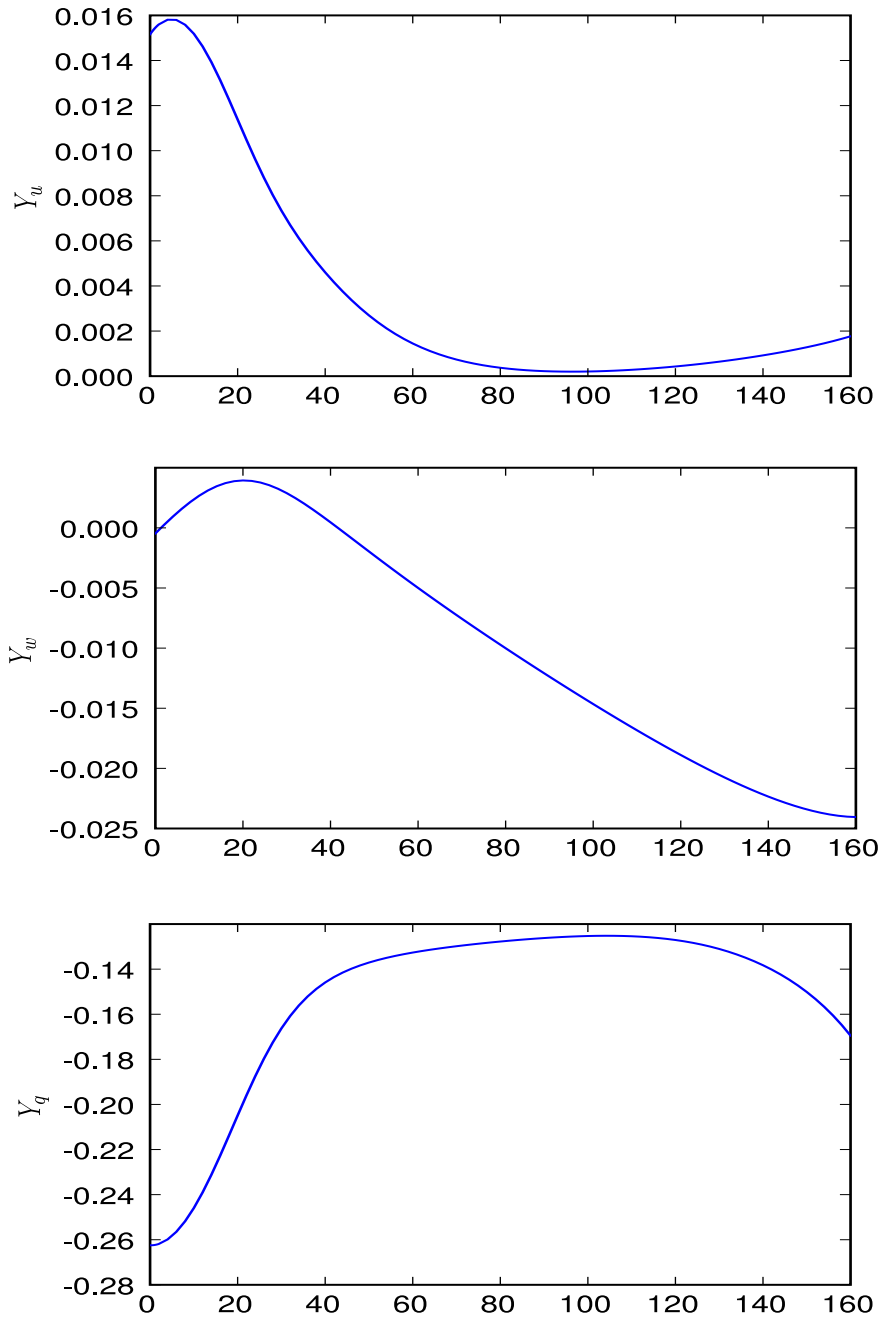

-0.010
-0.015
-0.020
-0.025
-0.030
-0.035
-0.040
X_r
0
20
40
60
80
100
120
140
160

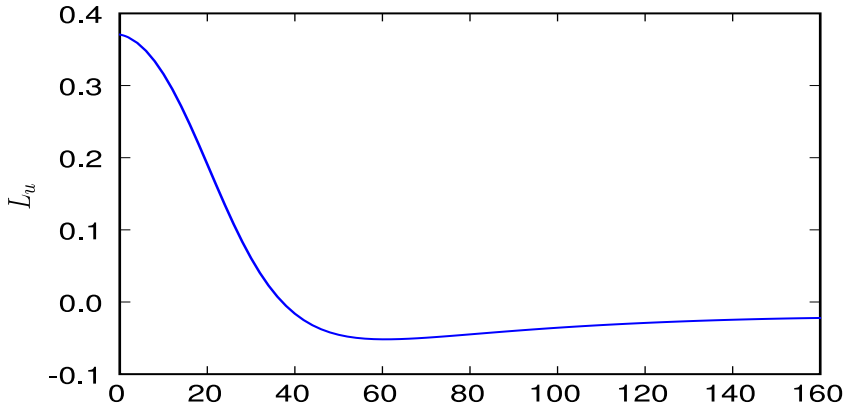

0.000
-0.005
-0.010
-0.015
-0.020
-0.025
Z_v
0
20
40
60
80
100
120
140
160

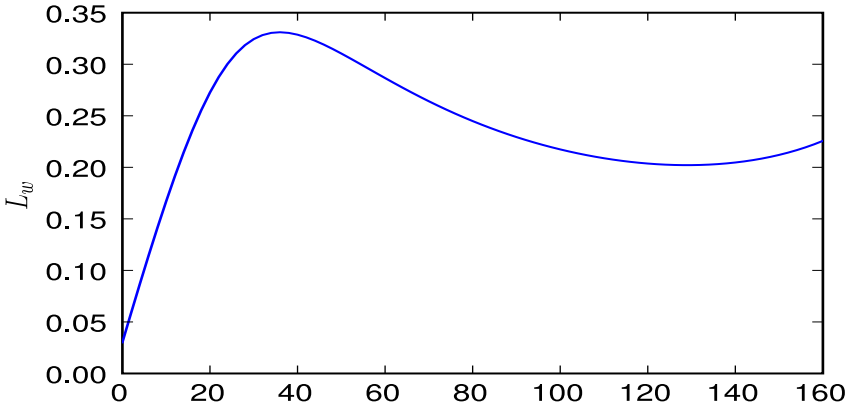

0.0
-0.2
-0.4
-0.6
-0.8
-1.0
Z_p
0
20
40
60
80
100
120
140
160

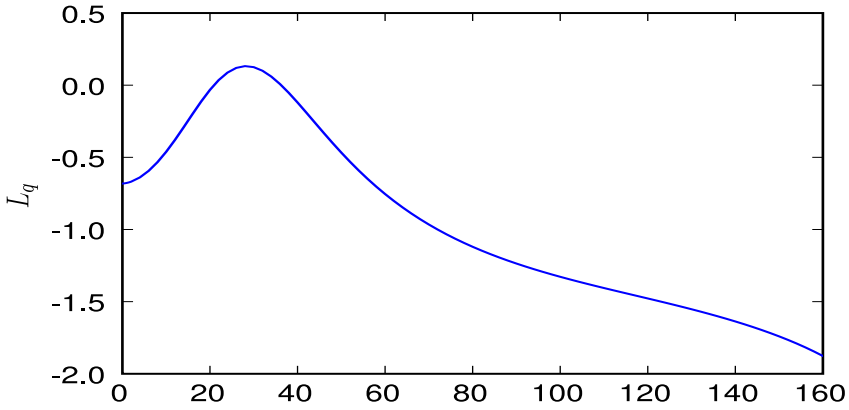

0.00
-0.01
-0.02
-0.03
-0.04
-0.05
-0.06
-0.07
-0.08
Z_r
0
20
40
60
80
100
120
140
160

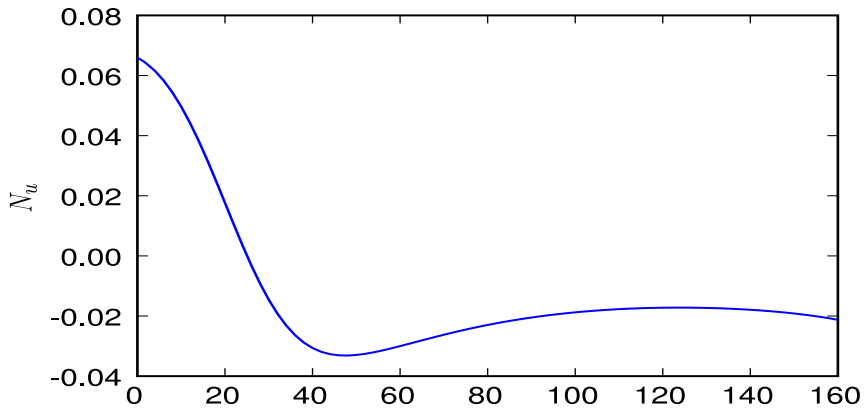

0.07
0.06
0.05
0.04
0.03
0.02
0.01
0.00
M_v
0
20
40
60
80
100
120
140
160
0.5
0.4
0.3
0.2
0.1
0.0
-0.1
M_p
0
20
40
60
80
100
120
140
160
0.015
0.010
0.005
0.000
-0.005
-0.010
M_r
0
20
40
60
80
100
120
140
160

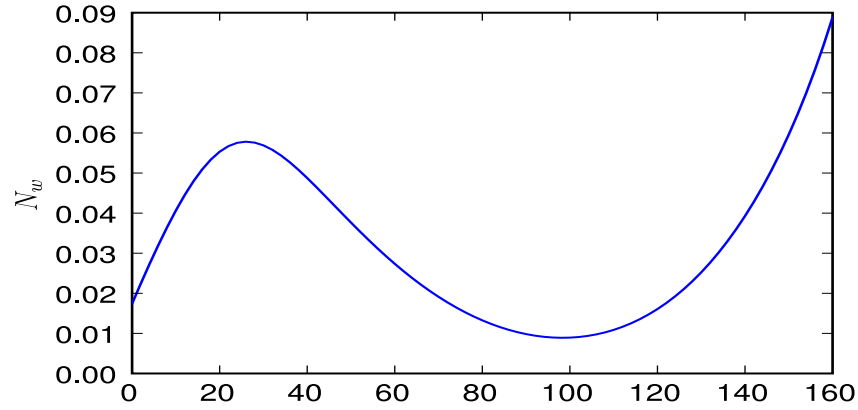

Y_u
0.016
0.014
0.012
0.010
0.008
0.006
0.004
0.002
0.000
0 20 40 60 80 100 120 140 160
Y_w
0.000
-0.005
-0.010
-0.015
-0.020
-0.025
0 20 40 60 80 100 120 140 160
Y_q
-0.14
-0.16
-0.18
-0.20
-0.22
-0.24
-0.26
-0.28
0 20 40 60 80 100 120 140 160

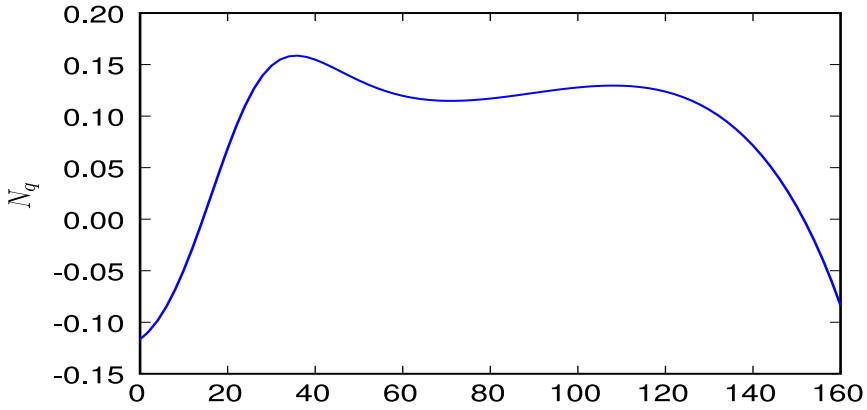

0.4
0.3
0.2
0.1
0.0
-0.1
L_u
0
20
40
60
80
100
120
140
160


0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
L_w
0
20
40
60
80
100
120
140
160


0.5
0.0
-0.5
-1.0
-1.5
-2.0
L_q
0
20
40
60
80
100
120
140
160

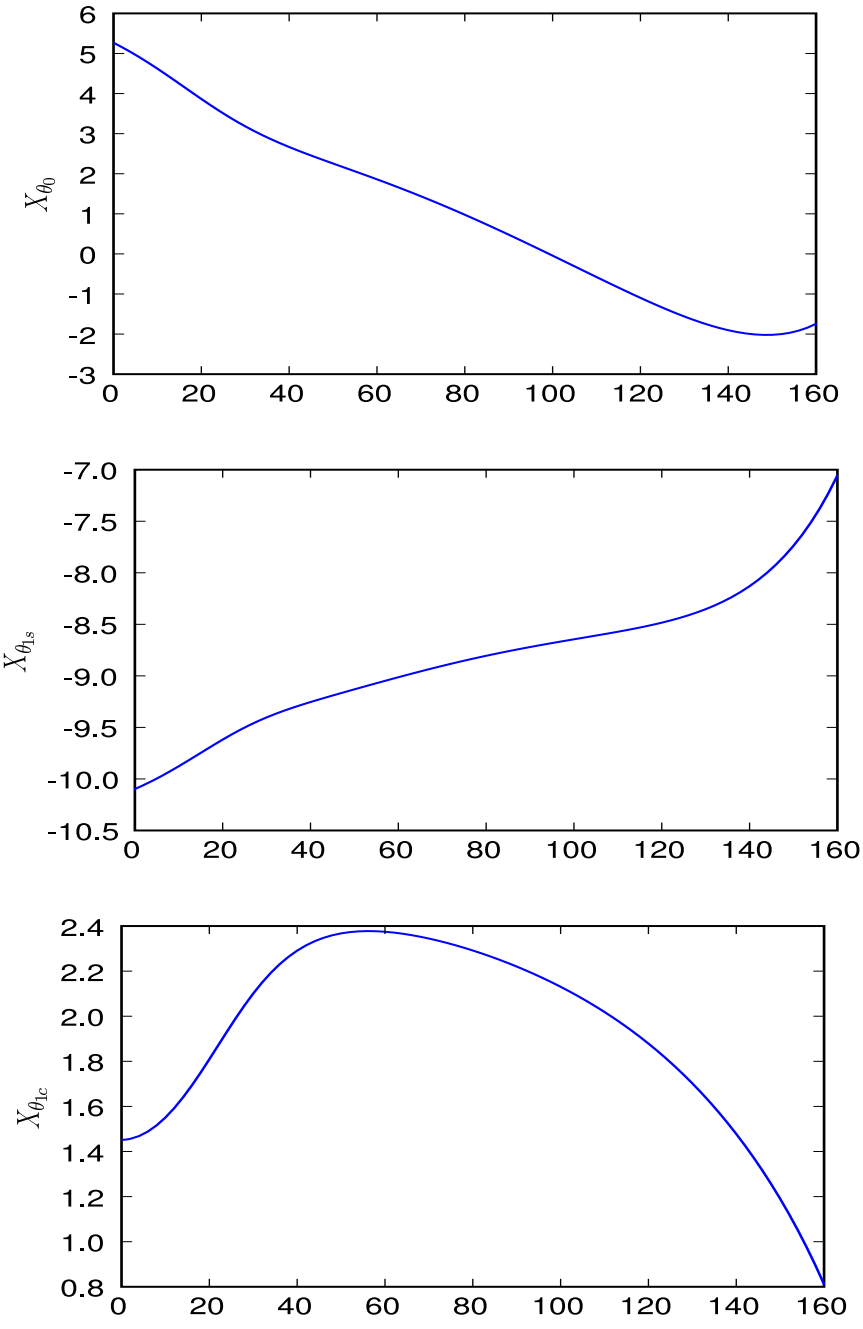

0.08
0.06
0.04
0.02
0.00
-0.02
-0.04
N_u
0
20
40
60
80
100
120
140
160

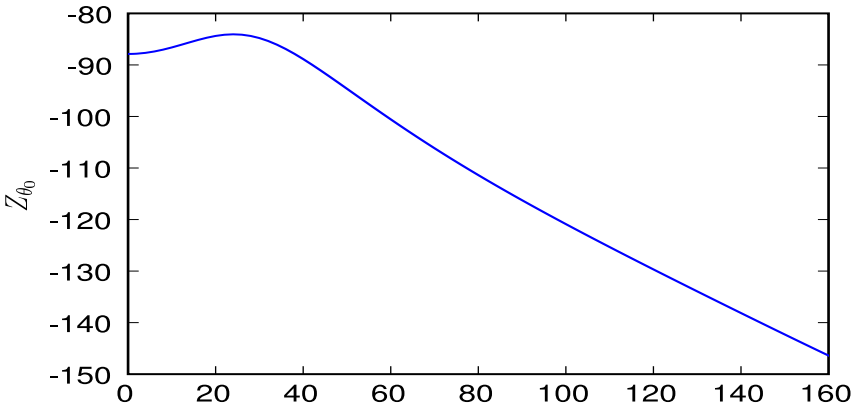

0.09
0.08
0.07
0.06
0.05
0.04
0.03
0.02
0.01
0.00
N_w
0
20
40
60
80
100
120
140
160

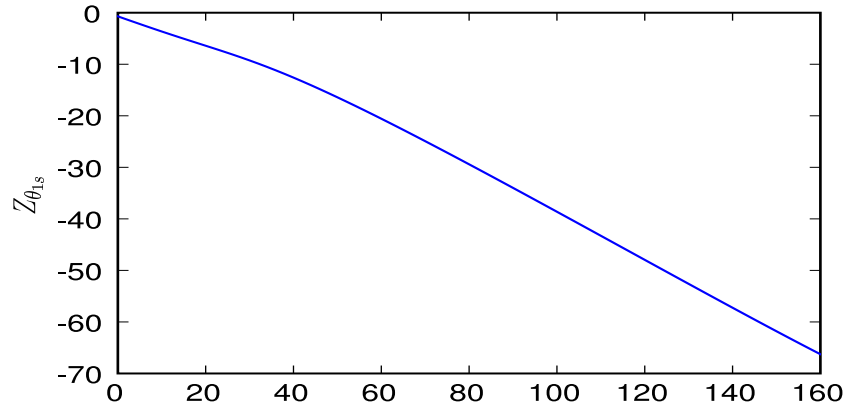

0.20
0.15
0.10
0.05
0.00
-0.05
-0.10
-0.15
N_q
0
20
40
60
80
100
120
140
160

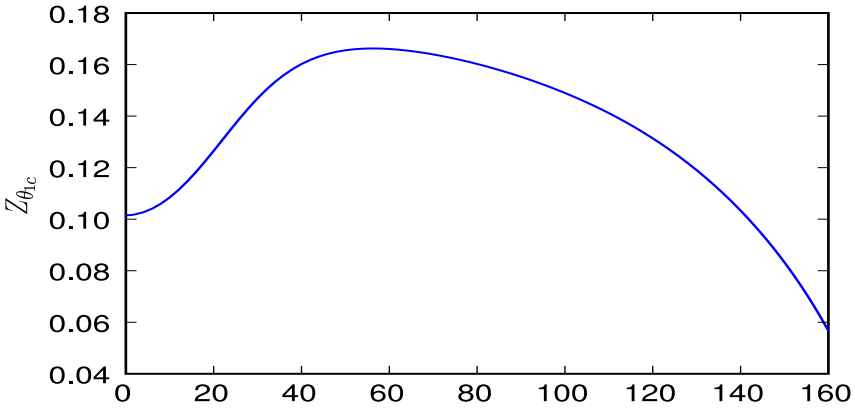
6.9. Cálculo de la respuesta

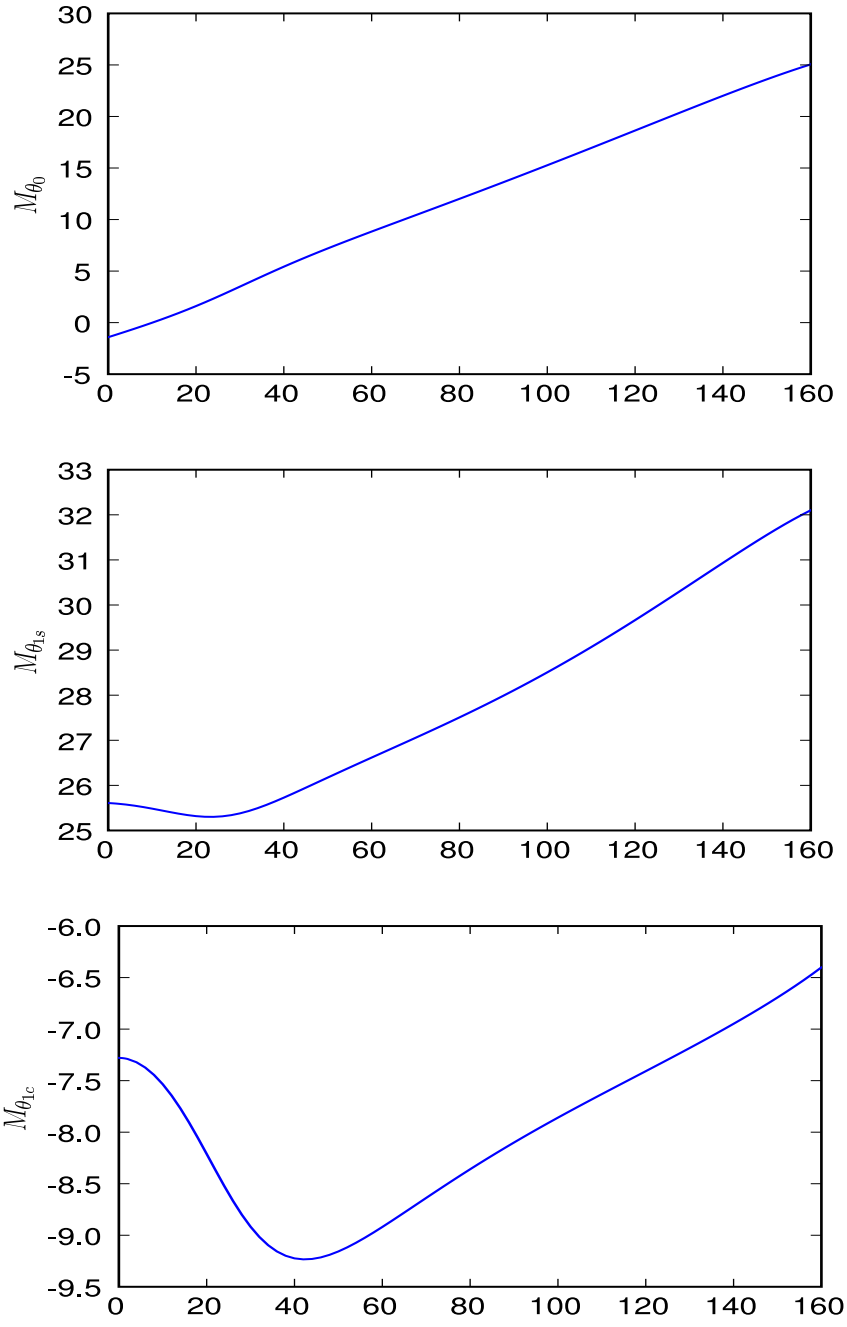
Se incluye un pequeño fichero de ejemplo `ensayos.py` que permite calcular la respuesta del helicóptero y muestra los resultados que se incluyen a continuación para entradas escalón a partir de vuelo trimado.

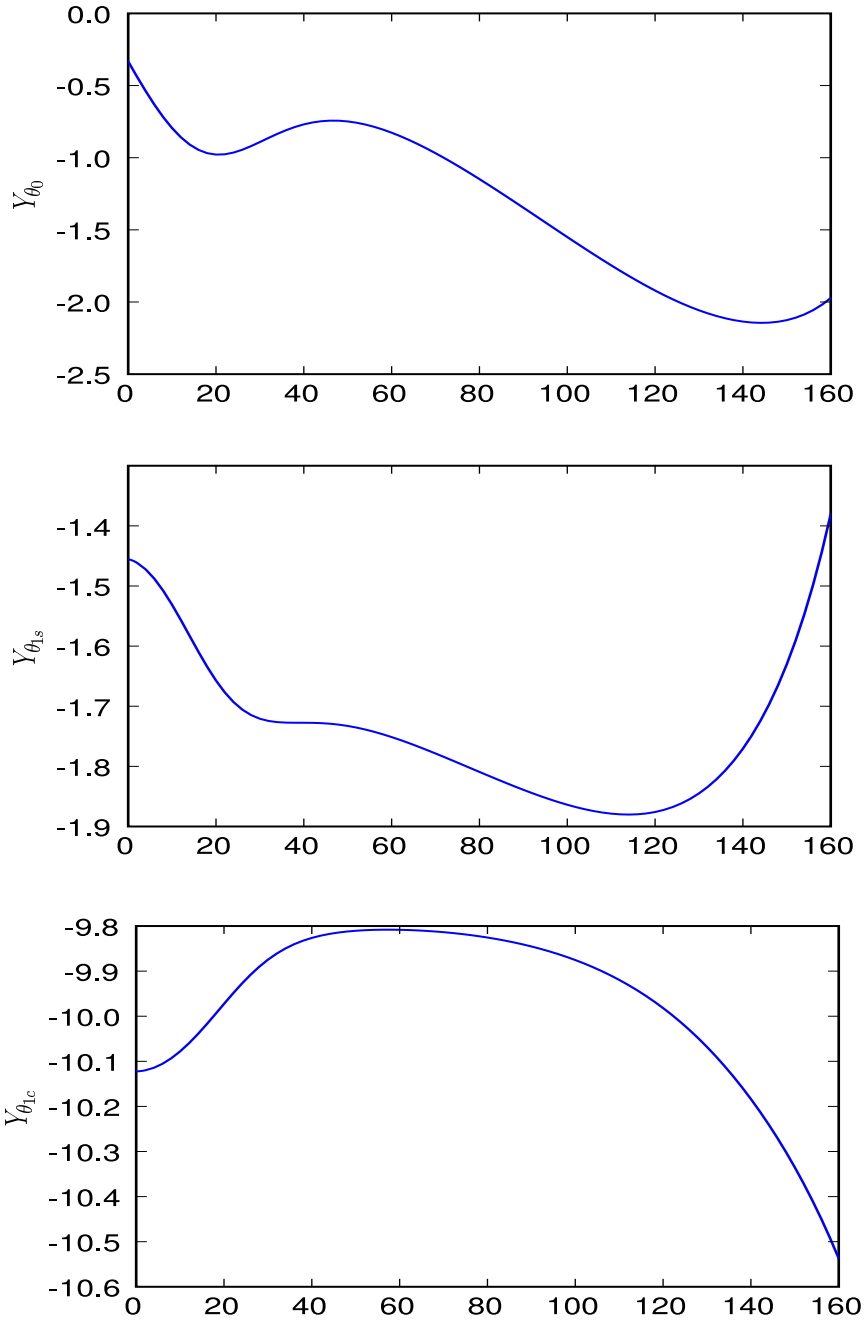

X_{θ_0}
6
5
4
3
2
1
0
-1
-2
-3
0
20
40
60
80
100
120
140
160
$X_{\theta_{1s}}$
-7.0
-7.5
-8.0
-8.5
-9.0
-9.5
-10.0
-10.5
$X_{\theta_{1c}}$
2.4
2.2
2.0
1.8
1.6
1.4
1.2
1.0
0.8

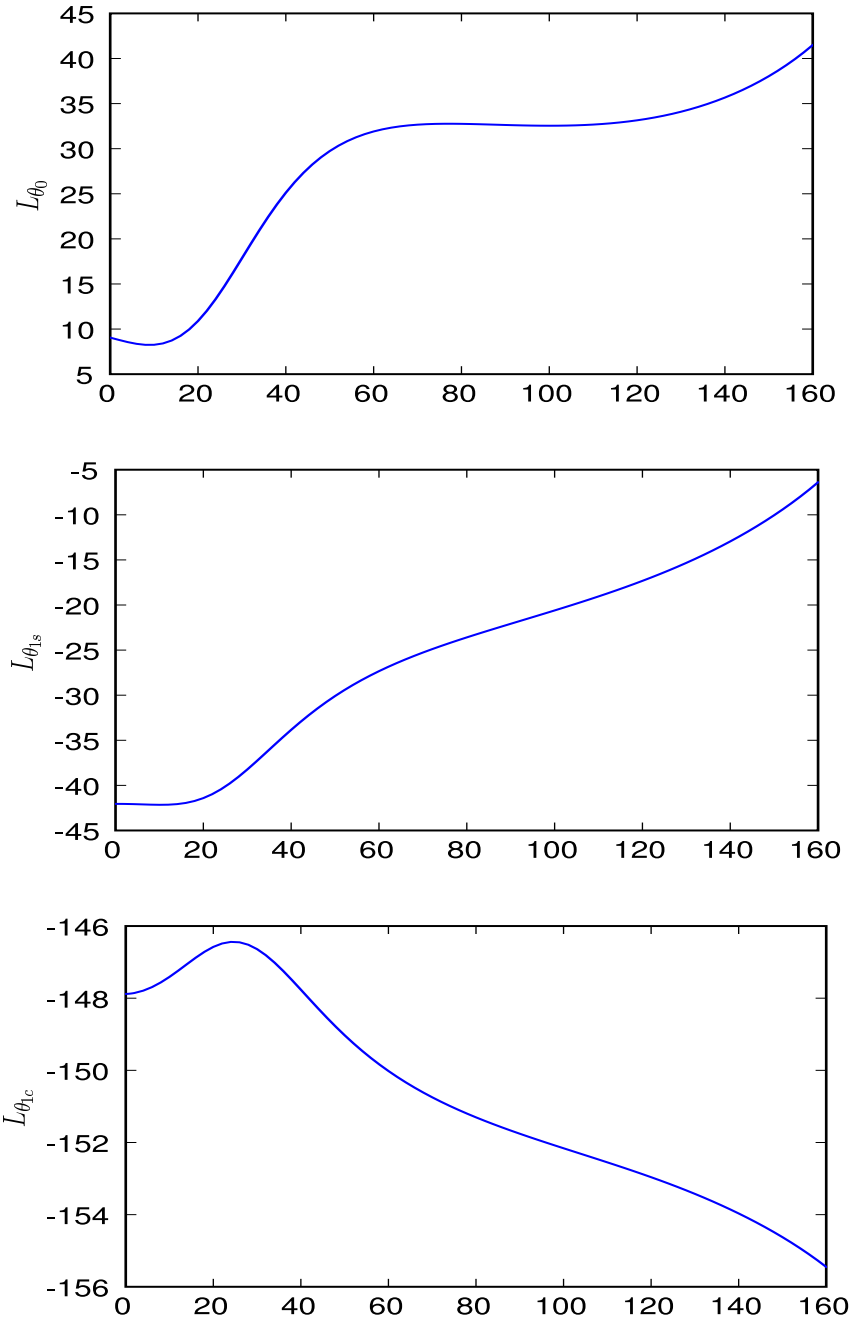

-80
-90
-100
-110
-120
-130
-140
-150
Z_{θ_0}
0
20
40
60
80
100
120
140
160

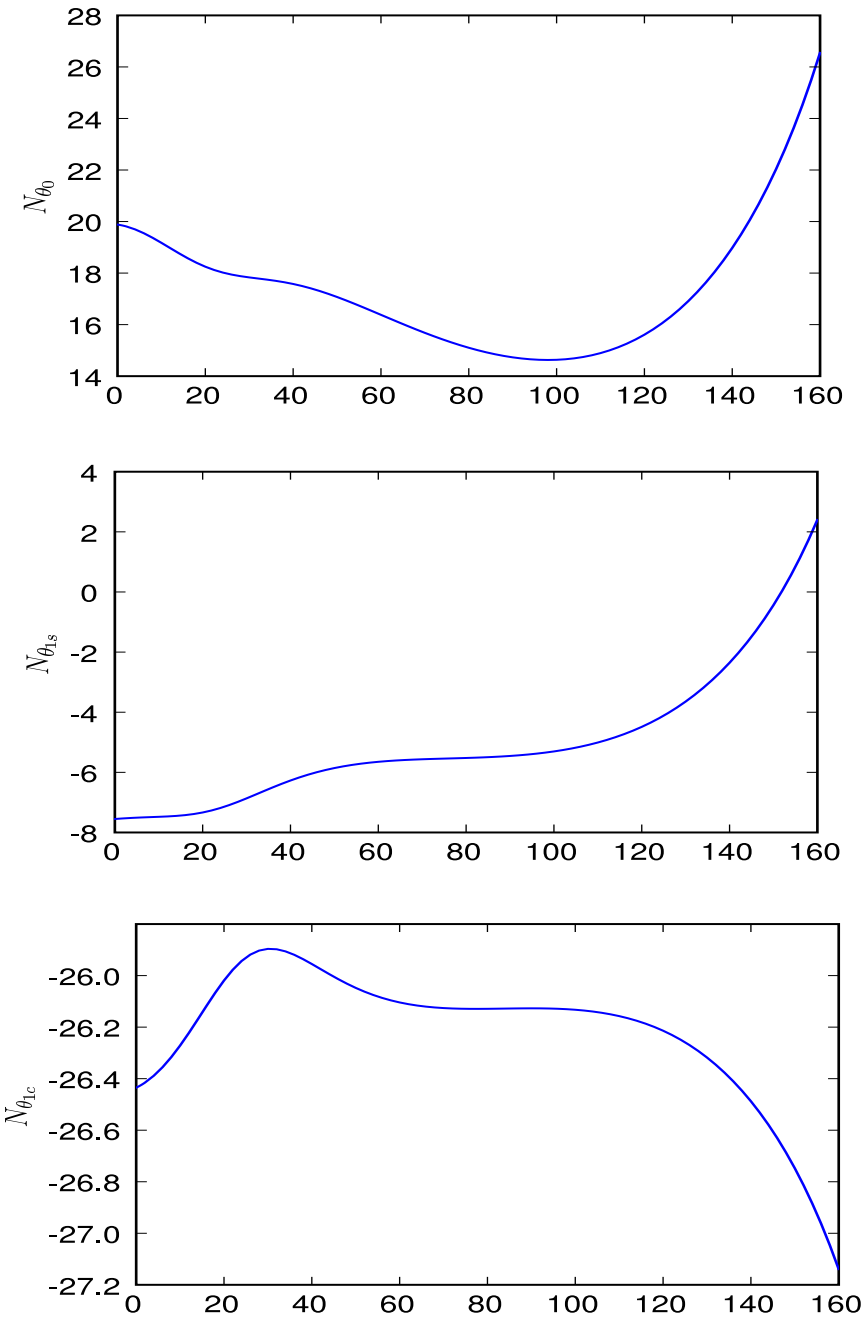

0
-10
-20
-30
-40
-50
-60
-70
$Z_{\theta_{1s}}$
0
20
40
60
80
100
120
140
160

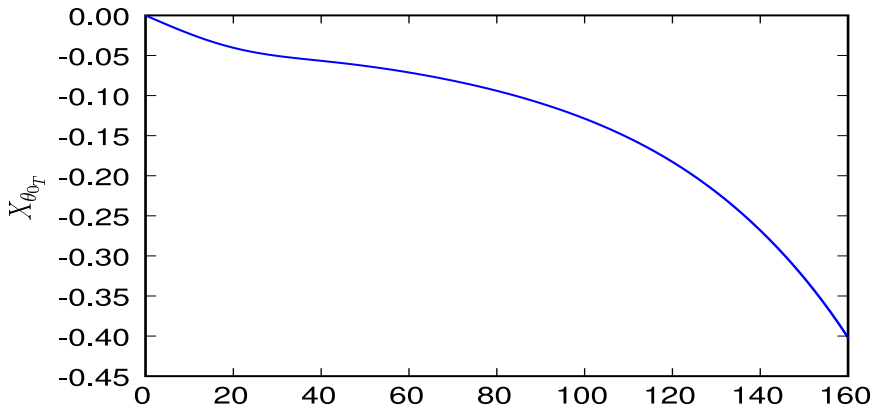

0.18
0.16
0.14
0.12
0.10
0.08
0.06
0.04
$Z_{\theta_{1c}}$
0
20
40
60
80
100
120
140
160

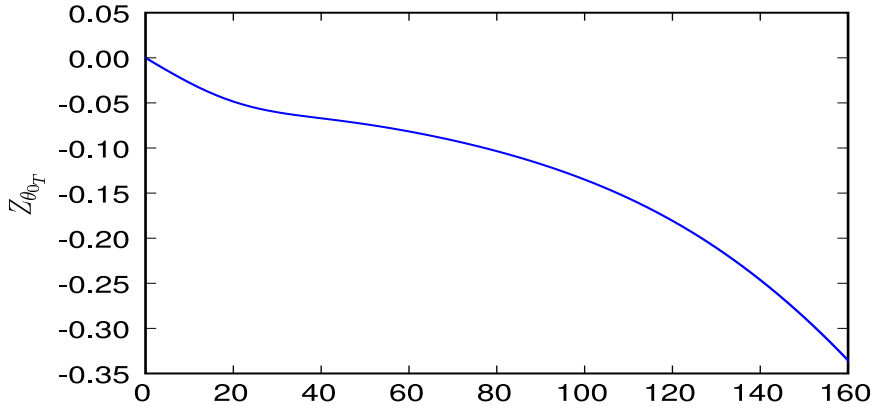

M_{θ_0}
30
25
20
15
10
5
0
-5
0
20
40
60
80
100
120
140
160
$M_{\theta_{1s}}$
33
32
31
30
29
28
27
26
25
0
20
40
60
80
100
120
140
160
$M_{\theta_{1c}}$
-6.0
-6.5
-7.0
-7.5
-8.0
-8.5
-9.0
-9.5
0
20
40
60
80
100
120
140
160

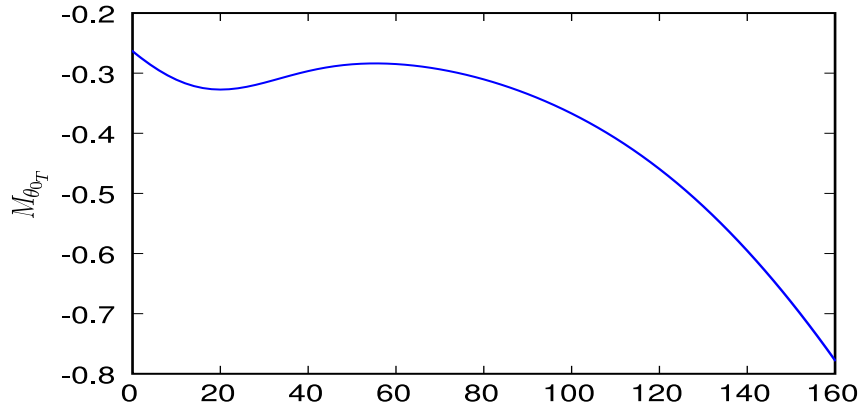

Y_{θ_0}
0.0
-0.5
-1.0
-1.5
-2.0
-2.5
0
20
40
60
80
100
120
140
160
$Y_{\theta_{1s}}$
-1.4
-1.5
-1.6
-1.7
-1.8
-1.9
$Y_{\theta_{1c}}$
-9.8
-9.9
-10.0
-10.1
-10.2
-10.3
-10.4
-10.5
-10.6

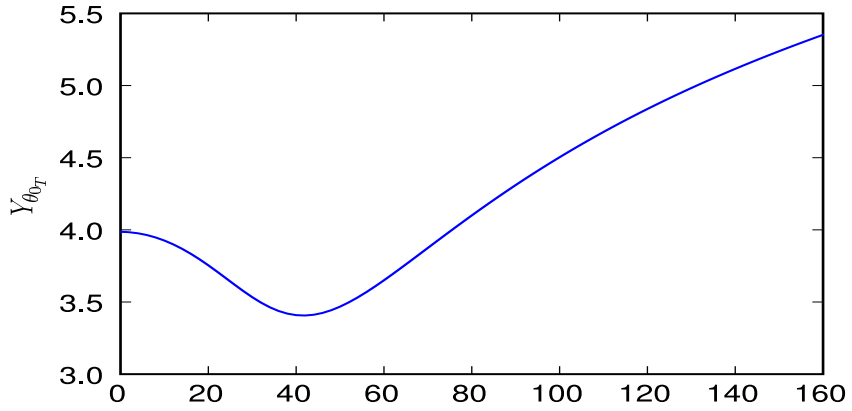

45
40
35
30
25
20
15
10
5
L_{θ_0}
0 20 40 60 80 100 120 140 160
-5
-10
-15
-20
-25
-30
-35
-40
-45
$L_{\theta_{1s}}$
0 20 40 60 80 100 120 140 160
-146
-148
-150
-152
-154
-156
$L_{\theta_{1c}}$
0 20 40 60 80 100 120 140 160

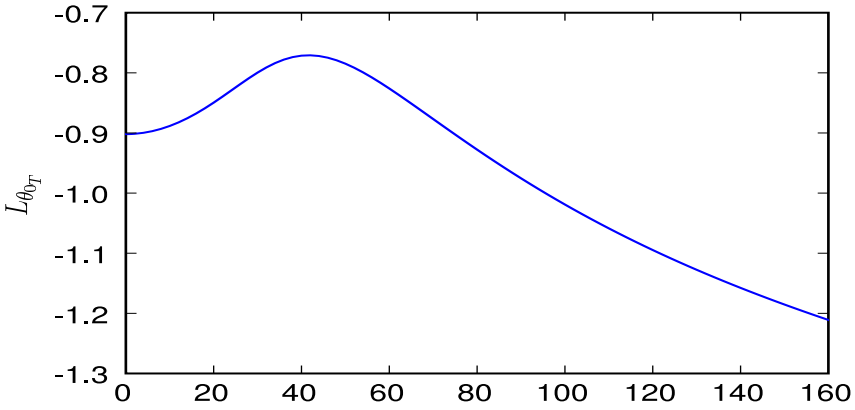

N_{θ_0}
28
26
24
22
20
18
16
14
0
20
40
60
80
100
120
140
160
$N_{\theta_{1s}}$
4
2
0
-2
-4
-6
-8
$N_{\theta_{1c}}$
-26.0
-26.2
-26.4
-26.6
-26.8
-27.0
-27.2

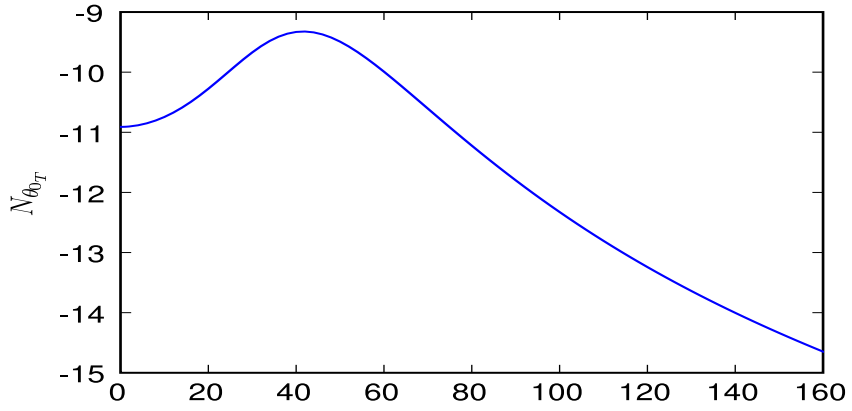

0.00
-0.05
-0.10
-0.15
-0.20
-0.25
-0.30
-0.35
-0.40
-0.45
$X_{\theta_{0_T}}$
0
20
40
60
80
100
120
140
160


0.05
0.00
-0.05
-0.10
-0.15
-0.20
-0.25
-0.30
-0.35
$Z_{\theta_{0_T}}$
0
20
40
60
80
100
120
140
160


-0.2
-0.3
-0.4
-0.5
-0.6
-0.7
-0.8
$M_{\theta_{0_T}}$
0
20
40
60
80
100
120
140
160


5.5
5.0
4.5
4.0
3.5
3.0
$Y_{\theta_{0_T}}$
0
20
40
60
80
100
120
140
160


-0.7
-0.8
-0.9
-1.0
-1.1
-1.2
-1.3
$L_{\theta_{0_T}}$
0
20
40
60
80
100
120
140
160


-9
-10
-11
-12
-13
-14
-15
$N_{\theta_{0_T}}$
0
20
40
60
80
100
120
140
160

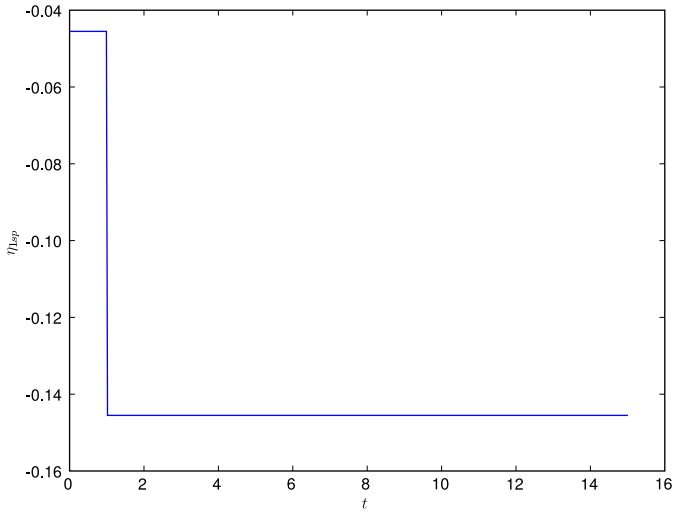


Figura 6.31: Entrada escalón para mando longitudinal a partir de vuelo trimado horizontal a 30 m/s: piloto

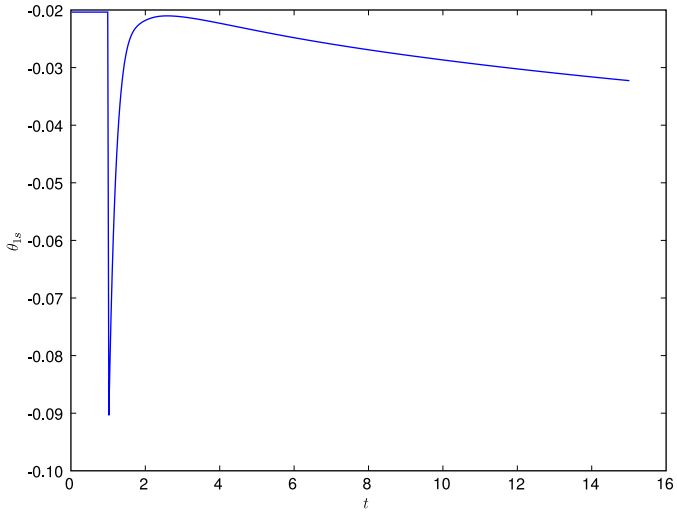


Figura 6.32: Entrada escalon para mando longitudinal a partir de vuelo trimado horizontal a 30 m/s: paso longitudinal

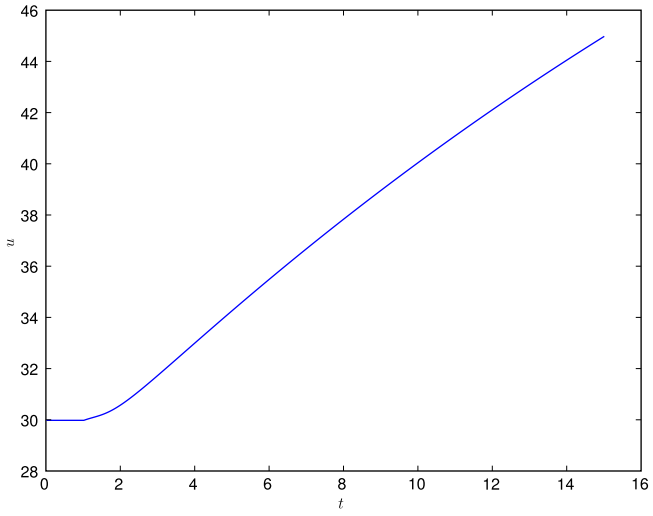


Figura 6.33: Entrada escalon para mando longitudinal a partir de vuelo trimado horizontal a 30 m/s: velocidad horizontal

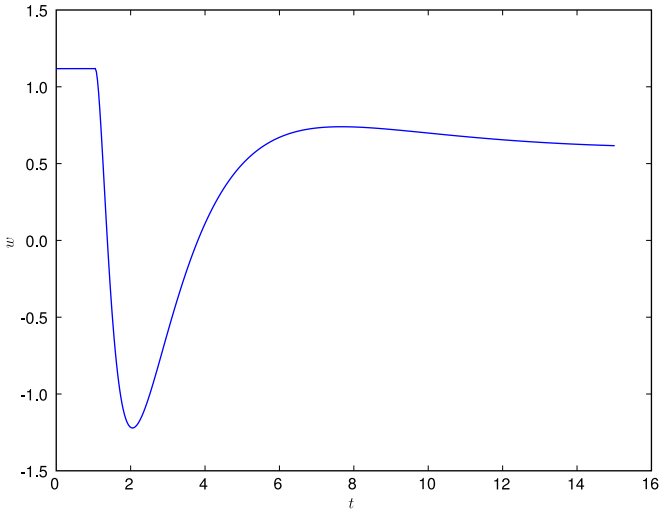


Figura 6.34: Entrada escalon para mando longitudinal a partir de vuelo trimado horizontal a 30 m/s: velocidad vertical

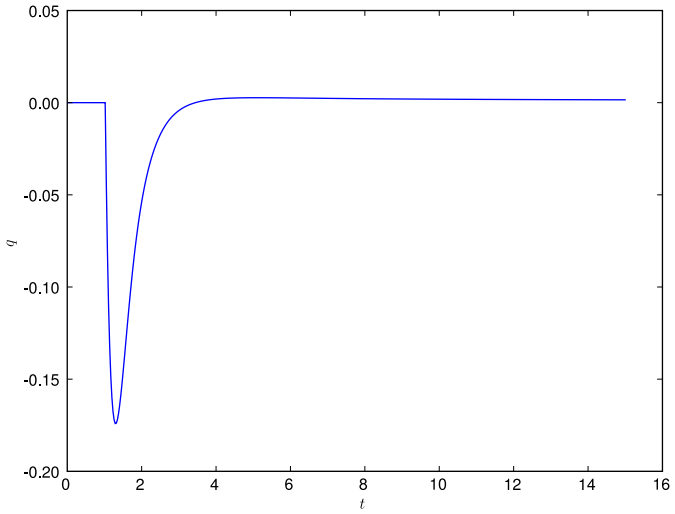


Figura 6.35: Entrada escalon para mando longitudinal a partir de vuelo trimado horizontal a 30 m/s: velocidad angular de cabeceo

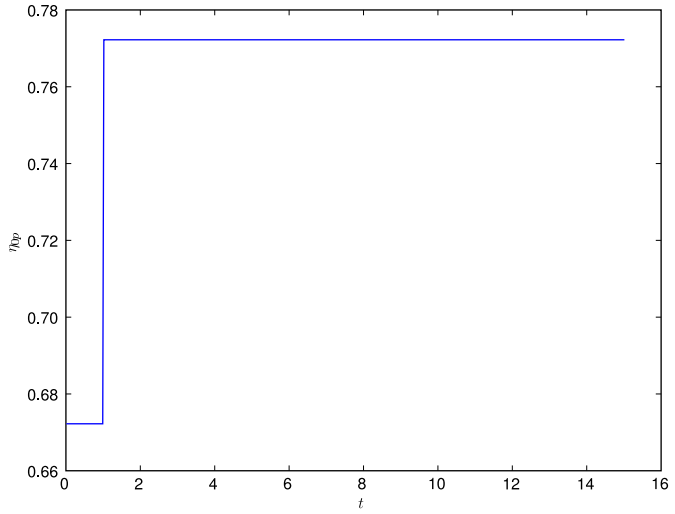


Figura 6.36: Entrada escalon para colectivo a partir de vuelo a punto fijo: piloto

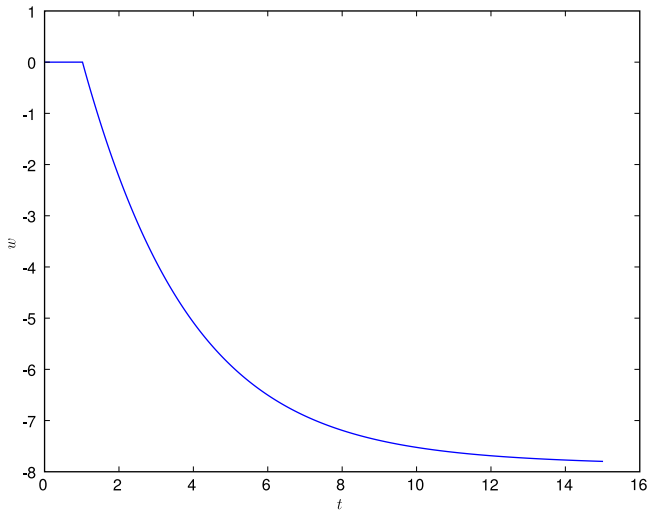


Figura 6.37: Entrada escalon para colectivo a partir de vuelo a punto fijo: velocidad vertical

Apéndice A

Manual de Usuario

A.1. Requerimientos

Para el cálculo de trimado y derivadas de estabilidad se requieren los siguientes programas y librerías:

- python 2.4
- numarray

Para representar en tiempo real el helicóptero y pilotarlo:

- FlightGear 0.9.10

A.2. Contenidos del CD

En la distribución se incluyen al menos los siguientes ficheros:

- `bin`
  - `icaro`
  - `c_colision.so`
  - `serializa`
  - `trimado`
  - `derivadas`

  `icaro` es el programa más importante y se incluye una descripción a continuación. `serializa`, `trimado` y `derivadas` son utilidades para el cálculo de diversos resultados, que se han utilizado en la realización del proyecto y `c_colision.so` es simplemente una librería de C que se necesita (su código se encuentra disponible en `src`).
- `src` Contiene el código fuente necesario para el funcionamiento del simulador
- `txt` Contiene el código LaTeXde la memoria, y todas las figuras en formato postscript y en el formato original necesarias.

- `dat`
  - `runfgfs.sh`: un script con las opciones típicas para invocar a FlightGear.
  - `X52.xml`: fichero con la descripción del joystick de la marca Saitek.
  - `prop-pedals-usb.xml`.: fichero con la descripción de los pedales de la marca CH.
  - `Lynx`: Contiene la descripción del helicóptero Lynx necesaria para el funcionamiento de FlightGear.

A.3. icaro

Para realizar un vuelo el primer paso es ejecutar `icaro`, como mínimo especificando el modelo a utilizar. Por ejemplo, si en el directorio actual se encuentra el fichero `Lynx.py` que contiene una descripción completa del modelo, el arranque del programa tendría el siguiente aspecto:

```
$icaro --modelo=Lynx
# INFO: Leyendo modelo de helicoptero Lynx
# INFO: Iniciando conexion de entrada de controles
# INFO: Entrando en el bucle de la simulacion.
# INFO: Presione Ctrl-C para cancelar la simulacion.
```

En este punto el simulador se queda esperando una conexión a FlightGear, por lo que arrancaríamos este último con, por lo menos, las siguientes opciones:

```
$fgfs --fdm=network,localhost,5001,5002,5003 --aircraft=Lynx
--enable-hud --model-hz=35
```

Es importante especificar un valor correcto para la opción `--modelo-hz` ya que es la frecuencia con que se ejecuta el modelo, por lo que tiene que ser suficientemente elevada para garantizar la convergencia del integrador y suficientemente baja para garantizar que de tiempo a realizar el paso de integración. Para el esquema Predictor-Corrector 35 es un valor aceptable.

Una descripción más detallada de las opciones se encuentra en la sección dedicada a FlightGear.

Una vez termine de arrancar FlightGear manda a nuestro programa la información de posición y orientación iniciales, icaro informa de ello:

```
# INFO: Posicion inicial recibida:
# INFO:         longitud = -2.135844 rad
# INFO:         latitud  = 0.656575 rad
# INFO:         altitud  = 1.290497 m
# INFO:         azimuth  = 4.712389 rad
# INFO:         altura del suelo  = 1.290497 m
```

Una vez hayamos terminado de volar, cancelamos la simulación pulsando Ctrl-C:

```
# Deteniendo la simulacion...
# INFO: Cerrando conexiones...
# INFO: Cerrando archivos...
```

A la descripción completa de las opciones que acepta icaro se puede acceder desde la línea de comandos:

```
$icaro --help

usage: icaro --modelo=M, [--logfile=F, --logprop=a, [b,c...]] [--host=H],
[--puertos=p0,p1,p2]

options:
  --version           show program's version number and exit
  -h, --help          show this help message and exit
  --modelo=MODELO     Nombre del archivo que contiene el modelo, sin la
                      extension .py
  --logfile=LOGFILE   Fichero opcional de salida de datos para logging
  --logprop=LOGPROP   Propiedades a escribir en el fichero de logging.
                      Consultar documentacion para ver la lista de
                      posibilidades
  --host=HOST         Direccion de la maquina ejecutando FlightGear, si no se
                      especifica se asume localhost
  --puertos=PUERTOS   Puertos de conexion con FlightGear, respectivamente son
                      de controles, del FDM y de comandos. Por defecto son los
                      puertos 5001, 5002 y 5003 respectivamente
```

A.3.1. Logging

Es posible almacenar en un fichero la historia temporal de ciertas variables; esto permite consultar más tarde los resultados en el fichero o incluso mostrarlos en pantalla en tiempo real.

Las variables que se quieran ir guardando se especifican en la línea de comandos como una lista separada por comas. Por ejemplo:

```
$icaro --modelo=Lynx --logfile=vuelo.dat --logprop=u,v,w
```

El simulador en este caso crearía el fichero `vuelo.dat` si no existe, y si existe, sobreescribiría encima y guardaría para cada instante de tiempo que calcule 4 columnas de datos. La primera columna indica el instante de tiempo y las restantes columnas las variables que se han especificado mediante `--logprop=u,v,w`, en este caso la velocidad en ejes cuerpo, en el mismo orden en que se han especificado.

En caso de que se pida una variable que no se encuentre calculada, bien porque se haya cometido un error tipográfico o porque no se pueda realizar el cálculo, el programa le asigna un valor de 0 y continúa.

Si se quiere visualizar interactivamente los resultados durante la simulación, el fichero de salida tiene ese formato precisamente para poder utilizarlo directamente con el programa `kst` (`http://kst.kde.org`). Para ello, mientras corre la simulación, deberíamos haber ejecutado:

```
$kst -x1 -y2 -y3 -y4 -m1 vuelo.dat
```

Si se quiere manipular la salida se puede pasar el fichero de logging a código en python. Se incluye una pequeña utilidad `serializa` que crea un interpolador lineal en función del tiempo para cada columna del fichero. Por ejemplo,

para utilizar en una sesión de python el fichero `vuelo.dat` del ejemplo anterior ejecutaríamos:

```
$serializa -i vuelo.dat -o vuelo.pickle
```

A continuación cargaríamos las variables en las sesión de python:

```
import pickle
f = open('vuelo.pickle', 'r')
u = pickle.load(f)
v = pickle.load(f)
w = pickle.load(f)
```

Ahora ya podemos manipular tranquilamente `u`, `v` y `w` con nuestro código. Por ejemplo, mostrándolo por pantalla:

```
import pylab
pylab.plot(u.x, u.y)
pylab.plot(v.x, v.y)
pylab.plot(w.x, w.y)
pylab.show()
```

Los objetos creados son interpoladores lineales tal como se encuentran definidos en `matematicas.py`. Disponen de dos atributos que contienen las coordenadas de los puntos que interpolan: `x` e `y` y pueden ser llamados como función para que calculen al valor en un punto arbitrario.

A continuación se da la lista de variables que se pueden pasar como argumentos, a `--logprop`, una descripción de su significado y el símbolo matemático asociado.

General

- `u` (u), `v` (v), `w` (w): componentes de la velocidad del helicóptero en ejes cuerpo, en metros por segundo.
- `p` (p), `q` (q), `r` (r): componentes de la velocidad angular del helicóptero en ejes cuerpo, en radianes por segundo.
- `q0` (q_0), `q1` (q_1), `q2` (q_2), `q3` (q_3): componentes del cuaternio de rotación.
- `th` (θ): ángulo de cabeceo, en radianes.
- `fi` (ϕ): ángulo de balance, en radianes.
- `ch` (ψ): ángulo de guiñada, en radianes.
- `alt` (h): altitud, en metros.
- `lon` (λ): longitud, en radianes.
- `lat` (ϕ): latitud, en radianes.
- `x_i` (x_i), `y_i` (y_i), `z_i` (z_i): posición en ejes inerciales

Rotor principal

- `la0` (λ_0), `la1cw` (λ_{1cw}), `la1sw` (λ_{1sw}): componentes de la velocidad inducida, en ejes rotor-viento.
- `be0` (β_0), `be1cw` (β_{1cw}), `be1sw` (β_{1sw}): componentes del batimiento, en ejes rotor-viento.
- `cT` (c_T): coeficiente de sustentación del rotor.
- `XR_b` (X_{R_b}), `YR_b` (Y_{R_b}), `ZR_b` (Z_{R_b}): componentes de las fuerzas del rotor sobre el centro de masas en ejes cuerpo, en Newtons.
- `LR_b` (L_{R_b}), `MR_b` (M_{R_b}), `NR_b` (N_{R_b}): componentes de los momentos del rotor sobre el centro de masas en ejes cuerpo, en Newtons por metro.

Fuselaje

- `alf` (α_f): ángulo de ataque del fuselaje, en radianes.
- `bef` (β_f): ángulo de resbalamiento del fuselaje, en radianes.
- `Xf_b` (X_{f_b}), `Yf_b` (Y_{f_b}), `Zf_b` (Z_{f_b}): componentes de las fuerzas del fuselaje sobre el centro de masas en ejes cuerpo.
- `Lf_b` (L_{f_b}), `Mf_b` (M_{f_b}), `Nf_b` (N_{f_b}): componentes de los momentos del fuselaje sobre el centro de masas en ejes cuerpo.

Controles

- `th0` (θ_0), `th1c` (θ_{1c}), `th1s` (θ_{1s}): componentes del paso del rotor en ejes rotor, en radianes.
- `th0T` (θ_{0_T}): paso del rotor de cola en radianes.
- `eta0p` (η_{0p}): control colectivo del piloto. Normalizado de 0 a 1.
- `eta1sp` (η_{1s}): control longitudinal del piloto. Normalizado de -1 a +1.
- `eta1cp` (η_{1c}): control lateral del piloto. Normalizado de -1 a +1.
- `etapp` (η_{pp}): control de pedales del piloto. Normalizado de -1 a +1.

Motor

- `Om` (Ω): velocidad de giro del rotor en rad/s.
- `Q1` (Q_1): par de un motor.
- `DTQ1` ($\dot{Q}_1$): derivada del par de un motor.

Rotor de cola

- `cTT` (c_{T_T}): coeficiente de sustentación del rotor de cola.
- `la0T` ($la0T$): velocidad inducida en el rotor de cola.
- `XT_b` (X_{T_b}), `YT_b` (Y_{T_b}), `ZT_b` (Z_{T_b}): componentes de las fuerzas del rotor de cola sobre el centro de masas, en ejes cuerpo.
- `LT_b` (L_{T_b}), `MT_b` (M_{T_b}), `NT_b` (N_{T_b}): componentes de los momentos del rotor de cola sobre el centro de masas, en ejes cuerpo.

Estabilizador vertical

- `alfn` (α_{fn}): ángulo de ataque del estabilizador vertical, en radianes.
- `befn` (β_{fn}): ángulo de resbalamiento del estabilizador vertical, en radianes.
- `Xfn_b` (X_{fn_b}), `Yfn_b` (Y_{fn_b}), `Zfn_b` (Z_{fn_b}): componentes de las fuerzas del estabilizador vertical sobre el centro de masas, en ejes cuerpo.
- `Lfn_b` (L_{fn_b}), `Mfn_b` (M_{fn_b}), `Nfn_b` (N_{fn_b}): componentes de los momentos del estabilizador vertical sobre el centro de masas, en ejes cuerpo.

Estabilizador horizontal

- `altp` (α_{tp}): ángulo de ataque del estabilizador horizontal, en radianes.
- `betp` (β_{tp}): ángulo de resbalamiento del estabilizador horizontal, en radianes.
- `Xtp_b` (X_{tp_b}), `Ytp_b` (Y_{tp_b}), `Ztp_b` (Z_{tp_b}): componentes de las fuerzas del estabilizador horizontal sobre el centro de masas, en ejes cuerpo.
- `Lfn_b` (L_{tp_b}), `Mtp_b` (M_{tp_b}), `Nfn_b` (N_{tp_b}): componentes de los momentos del estabilizador horizontal sobre el centro de masas, en ejes cuerpo.

A.4. FlightGear

A.4.1. Introducción

FlightGear (en adelante referido como FGFS, FlightGear Flight Simulator), es un simulador de vuelo distribuido bajo licencia GPL (GNU General Public License) totalmente gratis. La página oficial del proyecto se encuentra en `http://www.flightgear.org` y se puede leer una buena introducción a la estructura del proyecto en [13]. FGFS presenta las siguientes características interesantes para el desarrollo de este proyecto fin de carrera:

- Al ser licencia GPL se tiene acceso al código fuente, para leerlo y para modificarlo. Hubiese sido posible por ejemplo integrar el código del simulador de helicópteros en C++ dentro del propio programa FlightGear (cosa que no se ha hecho como se explica más adelante). Sí se ha utilizado dicha libertad de acceso al código fuente para estudiar el protocolo de transmisión entre FlightGear y programas externos a traves de sockets.

- Ha sido diseñado pensando en la adaptación a las necesidades de cada usuario. Resulta interesante, en particular, para proyectos de carácter académico y otros que se salgan de los intereses de la mayoría del mercado. Para ver una lista de proyectos que utilizan FGFS consultar `http://www.flightgear.org/Projects/`

- Está pensado para interaccionar con él mediante otros programas. Todo el estado del simulador, desde el modelo 3D del helicóptero, su posición, etc.. hasta la indicación de cualquier instrumento y el estado meteorológico se encuentra organizado en forma de árbol, simulando un sistema de archivos, al que se puede acceder desde diversos medios: telnet, http desde cualquier navegador web, sockets y tuberías.

- Formatos abiertos: los ficheros de configuración se encuentran en XML, esto permite trabajar con ellos desde cualquier editor de textos, además resultan bastante claros y sencillos. Hay una gran cantidad de aeronaves ya preparadas para, a partir de las cuales, crear tu propio fichero de configuración.

- Multiplataforma: Funciona tanto en Linux como en Windows o Mac. De hecho, es posible tener un ordenador con Linux ejecutando el FDM y otro con Windows encargandose de la entrada/salida con el usuario.

- Modularidad: En FGFS se encuentran separados claramente las partes encargadas de la entrada/salida de datos del FDM. De hecho, FGFS viene con 3 FDM ya incluidos: UUIC, YaSim y JSBSim. Esta separación permite que le podamos añadir un cuarto FDM especializado para helicópteros, que es el objeto de este proyecto fin de carrera.

A.4.2. Integración de FlightGear con el FDM de helicópteros

El simulador consta, por lo tanto de varios programas, operando simultáneos en uno o varios ordenadores y que se comunican entre ellos utilizando diversos protocolos, como se puede ver en la figura.

FGFS y el FDM se comunican a traves de 3 canales como se puede ver en la figura: controles, estado y comandos. Cada uno de ellos requiere de un puerto distinto por lo que al iniciar FGFS es necesario indicar en la línea de comandos que el FDM va a ser un programa aparte, el ordenador donde se encuentra y los puertos de comunicación. Por ejemplo:

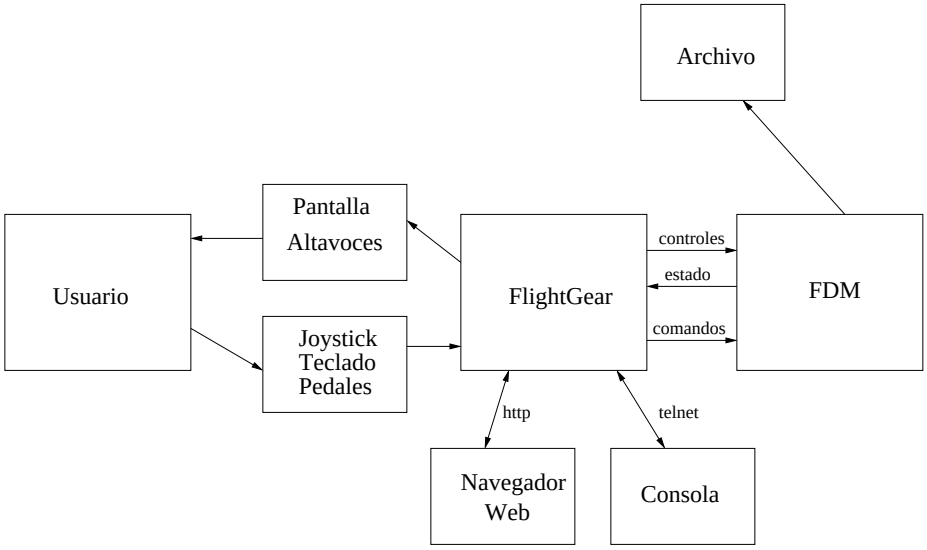
```
$fgfs --fdm=network,localhost,5001,5002,5003
```

arranca FGFS y especifica que el FDM correrá en el mismo ordenador que FGFS y utilizará el puerto 5001 para los controles, el puerto 5002 para el estado del FDM y el 5003 para los comandos.

Generalmente es necesario pasar más opciones, por ejemplo:

```
$fgfs --aircraft=Lynx --airport=LEVS  --httpd=5080
--fdm=network,localhost,5001,5002,5003
```

Arrancaría FGFS utilizando el helicóptero Lynx, situandolo inicialmente en el aeropuerto de Cuatro Vientos. Además especifica como antes los puertos de



comunicación con el FDM y añade un canal de comunicación en el puerto 5080 para poder inspeccionar y modificar el estado del simulador en tiempo real mediante un navegador web. En la imagen se puede observar como es posible acceder por ejemplo a todas las variables meteorológicas y modificarlas mediante el navegador web. Recordemos que no es necesario acceder desde el mismo ordenador. En el anterior caso un instructor podría estar modificando las variables atmosféricas desde cualquier otro ordenador, sea en una LAN o incluso Internet.

A.4.3. Canales de comunicación con el FDM

No existe documentación del protocolo de transmisión, aparte del código fuente de FGFS. Básicamente FGFS crea una estructura, la rellena con los datos a transmitir y la manda tal cual a través de un socket UDP. El FDM debe crear una estructura espejo a la anterior y copiar tal cual en memoria los datos recibidos. La estructura de los controles que FGFS manda al FDM y la estructura de estado que el FDM manda a FGFS se encuentra definida en

```
$FGSRC/src/Network/net_ctrls.hxx
$FGSRC/src/Network/net_fdm.hxx
```

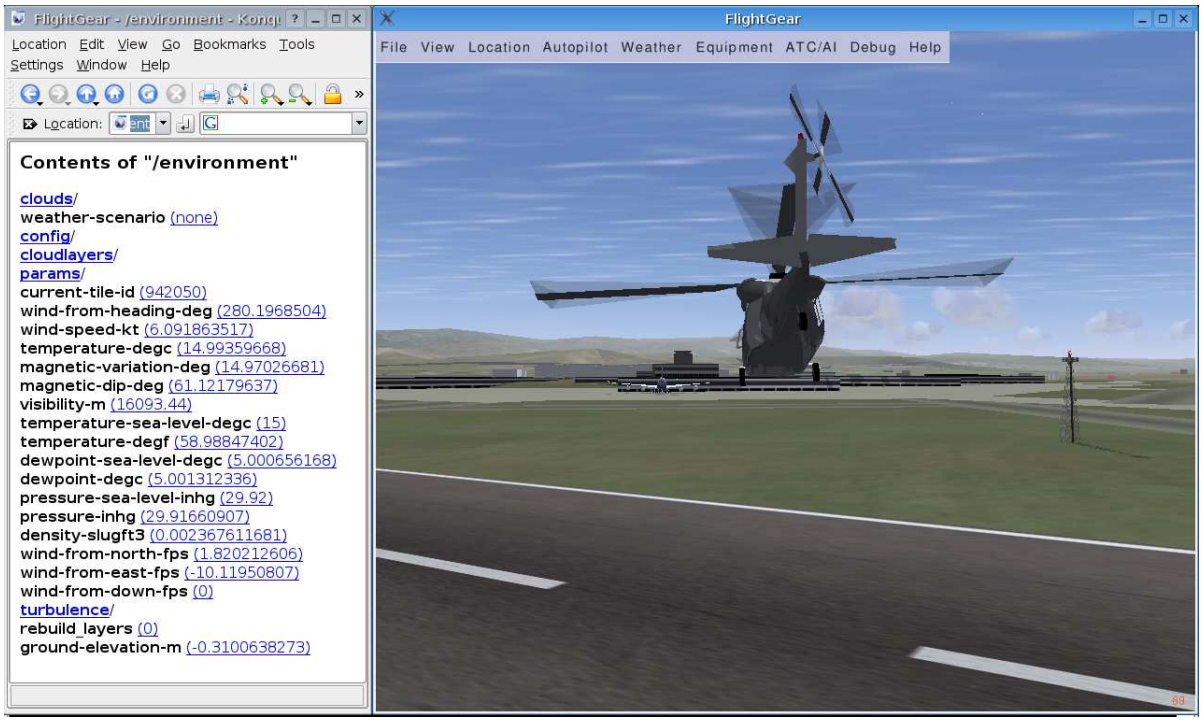
Donde `$FGSRC` indica el directorio raíz del código fuente de FGFS

El canal de comandos utiliza protocolo HTTP y manda información al inicio de la simulación al FDM con la posición inicial de la aeronave. Se encuentra definido en

```
$FGSRC/src/FDM/ExternalNet/ExternalNet.hxx
$FGSRC/src/FDM/ExternalNet/ExternalNet.cxx
```

A.4.4. Ficheros de FlightGear

Es necesario indicar a FGFS la aeronave a utilizar. Se puede obtener una lista de las aeronaves disponibles con la opción `--show-aircraft` que determi-



nará las aeronaves disponibles en función de lo que encuentre en el directorio

```
$FGROOT/Aircraft
```

Donde cada nave tiene su propio subdirectorio. Se ha incluido ya un archivo listo para descomprimir en el directorio de aeronaves que contiene la descripción necesaria para FGFS del Lynx. Dicho archivo contiene el modelo 3D, las texturas del modelo y los dos archivos XML con la configuración y descripción del panel de instrumentos.

También es necesario disponer de los terrenos necesarios. Para ello es necesario bajarse e instalar manualmente los archivos en el directorio

```
$FGROOT/Scenery
```

o utilizar la utilidad TerraSync, un programa aparte que se baja los archivos según sean necesarios desde internet. Se han incluido ya los dos archivos necesarios para la península ibérica listos para descomprimir en el directorio de escenarios.

Para más información consultar la documentación de FlightGear.

Apéndice B

Código

B.1. icaro.py

```
#!/usr/bin/env python
# -*- coding:latin 1 -*-
"""
Contiene el bucle principal
"""

from flightgear import FlightGear
from optparse import OptionParser
import sys
import imp

# Parseamos las opciones de la linea de comandos
parser = OptionParser(
        usage="%prog --modelo=M, [--logfile=F, --logprop=a, [b,c...]] " +
        "[--host=H], [--puertos=p0,p1,p2]",
        prog="icaro",
        version="1.0")

parser.add_option("--modelo",
        help = "Nombre del archivo que contiene el modelo, sin la " +
                "extension .py",
        type="string",
        dest="modelo")

parser.add_option("--logfile",
        help = "Fichero opcional de salida de datos para logging",
        type="string",
        dest="logfile")

parser.add_option("--logprop",
        help = "Propiedades a escribir en el fichero de logging. \n" +
                "Consultar documentacion para ver la lista de posibilidades",
        type="string",
        dest="logprop")

parser.add_option("--host",
        help = "Direccion de la maquina ejecutando FlightGear, si no se " +
        "especifica se asume localhost",
        type="string",
        dest="host")
```

```
parser.add_option("--puertos",
        help = "Puertos de conexion con FlightGear, respectivamente son " +
        "de controles, del FDM y de comandos. Por defecto son los " +
        "puertos 5001, 5002 y 5003 respectivamente",
        type="string",
        dest="puertos")

opt, args = parser.parse_args()
if opt.modelo == None:
    parser.error("--modelo requerido")

try:
        (file_modelo, path_modelo, desc_modelo) = imp.find_module(opt.modelo)
except ImportError:
        sys.exit("Error: no se encontro el modelo")

modulo = imp.load_module(opt.modelo, file_modelo, path_modelo, desc_modelo)

if opt.logprop!=None:
        logprop = opt.logprop.split(",")
else:
        logprop = []

if opt.host!=None:
    host = opt.host
else:
    host = "localhost"

if opt.puertos!=None:
    try:
        p1, p2, p3 = map(float, opt.puertos.split(","))
    except:
        parser.error("No se especificaron correctamente los puertos")
else:
    p1 = 5001
    p2 = 5002
    p3 = 5003

# Creamos el vuelo
vuelo = FlightGear(modulo.modelo, logfile=opt.logfile, props=logprop,
        host=host, port1=p1, port2=p2, port3=p3)

# Entramos en el bucle de la simulacion
try:
        while True:
                vuelo.loop()
except KeyboardInterrupt:
        print "# Deteniendo la simulacion..."
        vuelo.close()
        file_modelo.close()
```

B.2. flightgear.py

```
# -*- coding: latin-1 -*-
"""
Gestion de la conexion externa con FlightGear
"""

import logging
```

```
import socket
import time
import os

import protocolo
from historial import *
from matematicas import *
from geodesia import Geoide


def ft2m(feet):
    """
    Convierte de pies a metros.
    """

    return 0.3048*feet

class FlightGear:
    def __init__(self, modelo, logfile='vuelo.dat', props=[], host='localhost',
            port1=5001, port2=5002, port3=5003):
        """
        Gestiona la conexion con FlightGear.
            modelo: helicoptero que se quiera, una instancia de clase modelo.
            logfile: archivo donde se guarden las variables de logging
            props: lista con las variables a guardar
            host: direccion del ordenador que corre FlightGear.
            port1: puerto de controles.
            port2: puerto de FDM.
            port3: puerto de comandos.
        """

        # LOGGING DE ERRORES
        self.logger = logging.getLogger('FlightGear')
        handler = logging.StreamHandler()
        handler.setFormatter(logging.Formatter("# %(levelname)s: %(message)s"))
        self.logger.addHandler(handler)
        self.logger.setLevel(logging.INFO)

        # CARGAMOS EL MODELO
        self.logger.info('Leyendo modelo de helicoptero %s' % modelo.nombre)
        self.modelo = modelo
        self.modelo.init()

        # COMUNICACIONES
        self.logger.info('Iniciando conexion de entrada de controles')
        self.sockCtrls = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        self.sockFDM = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        self.sockCmd = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.cmds_recibidos = False

        try:
            self.sockCtrls.setblocking(False)
            self.sockCtrls.bind((host, port1))
        except socket.error, msg:
            self.logger.warning("No se pudo abrir socket de \
            entrada de controles: %s" % msg)
        try:
            self.sockFDM.setblocking(False)
            self.sockFDM.connect((host, port2))
        except socket.error, msg:
            self.logger.warning("No se pudo abrir socket de salida\
            de estado de FDM: %s" % msg)
```

```
        try:
            self.sockCmd.setblocking(False)
            self.sockCmd.bind((host, port3))
            self.sockCmd.listen(1)
            # Todavia hay que aceptar las conexiones
        except socket.error, msg:
            self.logger.warning("No se pudo abrir socket de \
            comandos: %s" % msg)

        self.props = Historial(10000)
        self.props.append({})
        self.cmds = Historial(10000)
        self.cmds.append({})

        # LOGGING
        self.logProps = props
        if logfile != None:
            self.logger.info('Abriendo archivo log: %s' % logfile)
            try:
                self.archivoLog = open(logfile, 'w')
            except:
                self.archivoLog = None
                self.logger.warning('No se pudo abrir archivo para logging')
        else:
            self.archivoLog = None
        self.logger.info('Entrando en el bucle de la simulacion. ')
        self.logger.info('Presione Ctrl-C para cancelar la simulacion. ')

    def close(self):
        """
        Cierra todos los sockets y archivos.
        """

        self.logger.info("Cerrando conexiones...")
        self.sockCtrls.close()
        self.sockFDM.close()
        self.sockCmd.close()

        self.logger.info("Cerrando archivos...")
        if self.archivoLog != None:
            self.archivoLog.close()

    def sendFDM(self):
        """
        Manda el diccionario de las propiedades.
        """

        try:
            self.sockFDM.send(protocolo.fdmPack(self.props[0]['fdm']))
        except socket.error, value:
            self.logger.warning("No se pudo mandar estado FDM: %s" % value)

    def recvCtrls(self):
        """
        Recibe el diccionario de controles.
        """

        try:
            str = self.sockCtrls.recv(protocolo.ctrlBufSize)
            self.props[0]['ctrls'] = protocolo.ctrlUnpack(str)
        except:
            raise
```

```
    def recvCmd(self):
        """
        Recibe los comandos de inicializacion. Cuando termina de
        recibir todos ajusta los valores en el modelo.
        """

        try:
            conn, addr = self.sockCmd.accept()
            conn.setblocking(True)
        except:
            raise
        else:
            msg = ""
            chunk = conn.recv(1)
            while chunk != "":
                msg += chunk
                chunk = conn.recv(1)
            match = protocolo.reCmd.match(msg)
            # Debemos responder a la inicializacion de las siguientes
            # propiedades (por orden):
            # longitude-deg, latitude-deg, altitude-ft, ground-m, speed-kts,
            # heading-deg, reset(=ground)
            if match != None:
                def longitud(strVal):
                    self.cmds[0]['longitud'] = rad(float(strVal))

                def latitud(strVal):
                    self.cmds[0]['latitud'] = rad(float(strVal))

                def altitud(strVal):
                    self.cmds[0]['altitud'] = 0.0
                    # Por algun motivo FG no nos pasa la altitud correcta
                    # asi que de momento esto lo dejamos en paz, mas adelante
                    # se corrige haciendo altitud = altitud del suelo

                def altura(strVal):
                    # Realmente quiere decir altitud del suelo
                    self.cmds[0]['altura'] = float(strVal)

                def orientacion(strVal):
                    self.cmds[0]['azimuth'] = rad(float(strVal))

                # Todos los valores recibidios
                def reset(strVal):
                    self.logger.info("Posicion inicial recibida:")
                    self.logger.info(
                            "\tlongitud = %f rad" % self.cmds[0]['longitud'])
                    self.logger.info(
                            "\tlatitud  = %f rad" % self.cmds[0]['latitud'] )
                    self.logger.info(
                            "\taltitud  = %f m"   % self.cmds[0]['altura'] )
                    self.logger.info(
                            "\tazimuth  = %f rad" % self.cmds[0]['azimuth'] )
                    self.logger.info(
                            "\taltura del suelo  = %f m" %
                            self.cmds[0]['altura'])

                    self.cmds_recibidos = True

                prop, val = match.group('prop'), match.group('value')
                try:
```

```
                    {
                    'longitude-deg':    longitud,
                    'latitude-deg':     latitud,
                    'altitude-ft':      altitud,
                    'ground-m':         altura,
                    'heading-deg':      orientacion,
                    'reset':            reset,
                    # No se encuentra implementado
                    'speed-kts':        lambda x: None,
                    }[prop](val)
                except KeyError:
                    self.logger.warning(
                            "Comando inesperado: %s = %s" % (prop, val))
            else:
                self.logger.warning("Error de protocolo")


    def loop(self):
        """
        Recibe los controles, calcula la respuesta y manda el
        resultado.
        """
        self.props.append({'ctrls': None, 'fdm':None, 'time': time.time()})

        if not self.cmds_recibidos:
            try:
                self.recvCmd()
            except socket.error:
                pass

        try:
            self.recvCtrls()
        except socket.error:
            pass
        else:
            ##########################################################################
            #   CON LA INFORMACION QUE NOS LLEGA ACTUALIZAMOS EL HELICOPTERO   #
            ##########################################################################

            # CONTROL DE TIEMPOS
            # si no hemos parado la simulacion nos aseguramos de que el reloj
            # esta corriendo, si ya esta corriendo la siguiente llamada no
            # afecta el funcionamiento
            if self.props[0]['ctrls']['freeze'] == 0:
                self.modelo.reloj.arranca()
            else:
                self.modelo.reloj.para()
                return
            t = self.modelo.reloj()
            # hemos recibido los controles y el reloj esta corriendo, ajustamos
            # el modelo.
            self.modelo.controles.et0p.append(
                    1-self.props[0]['ctrls']['throttle'][1], t)
            self.modelo.controles.et1cp.append(
                    -self.props[0]['ctrls']['aileron'], t)
            self.modelo.controles.et1sp.append(
                    -self.props[0]['ctrls']['elevator'], t)
            self.modelo.controles.etpp.append(
                    -self.props[0]['ctrls']['rudder'], t)

            # atmosfera
            self.modelo.T = self.props[0]['ctrls']['temp_c'] + 273.15
```

```
            self.modelo.P = self.props[0]['ctrls']['press_inhg']*3369.39
            self.modelo.viento_vel = self.props[0]['ctrls']['wind_speed']*0.814444
            self.modelo.viento_dir = rad(self.props[0]['ctrls']['wind_dir_deg'])

            # Encendemos o apagamos el motor
            if self.props[0]['ctrls']['magnetos'][0] > 0:
                self.modelo.motor.start = True
                self.modelo.motor.nmotores = 2
            else:
                self.modelo.motor.start = False
                self.modelo.motor.nmotores = 0

            # Altura del suelo sobre el geoide de referencia
            self.modelo.hG = self.props[0]['ctrls']['hground']

            if self.cmds_recibidos:
                # Vector de estado, condicion inicial
                self.modelo.t   = 0.0
                self.modelo.hG  = self.cmds[0]['altura']

                # Preparamos el integrador
                # Vector de estado
                u = 0.0
                v = 0.0
                w = 0.0

                p = 0.0
                q = 0.0
                r = 0.0

                ( q0, q1, q2, q3 ) = Quaternion.rotacion([
                    (pi - self.cmds[0]['azimuth'], 0, 0, 1),
                    (pi, 1, 0, 0) ])

                Om = 0.0
                DTQ = 0.0
                Q = 0.0

                x_i = 0.0
                y_i = 0.0
                z_i = self.modelo.hG + self.modelo.fuselaje.ht

                # Motores apagados
                self.modelo.motor.nmotores = 0

                self.modelo.tren.reset()

                self.modelo.integrador.init(
                    F = self.modelo.paso,
                    ti = 0,
                    xi = array(
                        [ u, v, w,
                          p, q, r,
                          q0, q1, q2, q3,
                          Om, DTQ, Q,
                          x_i, y_i, z_i ]))

                # Posicion inicial y ejes inerciales
                self.modelo.geoide.ejesLocales(
                    self.cmds[0]['latitud'],
                    self.cmds[0]['longitud'],
                    0.0
```

```
        )
        # Reseteamos cmds
        self.cmds.append({})
        self.cmds_recibidos = False


    ##########################################################################
    #   CALCULAMOS EL RESULTADO                                              #
    ##########################################################################
    # integramos las ecuaciones hasta el instante actual
    self.modelo.avanza(self.modelo.reloj())

    ##########################################################################
    #   EJECUTAMOS EL LOGGING DE PROPIEDADES                                 #
    ##########################################################################
    if self.archivoLog != None:
        if self.logProps != []:
            str = "%.4f "
            val = [self.modelo.t]
        for P in self.logProps:
            str += "%.4f "
            try:
                v = self.modelo.get(P)
            except KeyError:
                v = 0.0
            val.append(v)

        str += "\n"
        self.archivoLog.write(str % tuple(val))
        # Si no hacemos un flush no funciona para graficos interactivos
        self.archivoLog.flush()

    ##########################################################################
    #   MANDAMOS LA INFORMACION CALCULADA                                    #
    ##########################################################################

    # preparamos la informacion del modelo.
    # asignemos o no valores hay que rellenar correctamente el
    # diccionario
    fdm = {
        'version':      24,
        'padding':      0,
        # Posiciones
        # Coordenadas geodesicas en radianes
        'longitude':            0.0,
        'latitude':             0.0,
        'altitude':             0.0,
        'agl':                  1.0,    # Above Ground Level
        'phi':                  0.0,    # Euler en radianes
        'theta':                0.0,
        'psi':                  0.0,
        'alpha':                1.0,    # Ataque (rad)
        'beta':                 2.0,    # Resbalamiento (rad)
        # Velocidades
        'phidot':               3.0,
        'thetadot':             4.0,
        'psidot':               5.0,
        'vcas':                 6.0,
        'climb_rate':           7.0,
        'v_north':              8.0,
        'v_east':               9.0,
        'v_down':               10.0,
```

```
        'v_wind_body_north':    11.0,
        'v_wind_body_east':     12.0,
        'v_wind_body_down':     13.0,
        # Aceleraciones
        'A_X_pilot':            14.0,
        'A_Y_pilot':            15.0,
        'A_Z_pilot':            16.0,
        # Perdida
        'stall_warning':        0.0,
        'slip_deg':             0.0,
        # Motores
        'num_engines':          2,
        'eng_state':            [0, 0, 0, 0],
        'rpm':                  [0.0, 0.0, 0.0, 0.0],
        'fuel_flow':            [0.0, 0.0, 0.0, 0.0],
        'fuel_px':              [0.0, 0.0, 0.0, 0.0],
        'egt':                  [0.0, 0.0, 0.0, 0.0],
        'cht':                  [0.0, 0.0, 0.0, 0.0],
        'mp_osi':               [0.0, 0.0, 0.0, 0.0],
        'tit':                  [0.0, 0.0, 0.0, 0.0],
        'oil_temp':             [0.0, 0.0, 0.0, 0.0],
        'oil_px':               [0.0, 0.0, 0.0, 0.0],
        # Consumibles
        'num_tanks':            0,
        'fuel_quantity':        [0.0, 0.0, 0.0, 0.0],
        # Tren de aterrizaje
        'num_wheels':           0,
        'wow':                  [0, 0, 0],
        'gear_pos':             [0.0, 0.0, 0.0],
        'gear_steer':           [0.0, 0.0, 0.0],
        'gear_compression':     [0.0, 0.0, 0.0],
        # Entorno
        'cur_time':             0,
        'warp':                 0,
        'visibility':           1000.0,
        # Controles
        'elevator':             0.0,
        'elevator_trim_tab':    0.0,
        'left_flap':            0.0,
        'right_flap':           0.0,
        'left_aileron':         0.0,
        'right_aileron':        0.0,
        'rudder':               0.0,
        'nose_wheel':           0.0,
        'speedbrake':           0.0,
        'spoilers':             0.0
    }
    # version
    fdm['version'] = 24

    fdm['latitude']  = self.modelo.lat
    fdm['longitude'] = self.modelo.lon
    fdm['altitude']  = self.modelo.alt

    fdm['psi']   = self.modelo.ch
    fdm['theta'] = self.modelo.th
    fdm['phi']   = self.modelo.fi

    fdm['agl'] = self.modelo.alt - self.modelo.hG

    fdm['phidot']   = self.modelo.DTfi
    fdm['thetadot'] = self.modelo.DTth
```

```
            fdm['psidot']    = self.modelo.DTch

            fdm['A_X_pilot'] = self.modelo.ecs_modelo[0][0]
            fdm['A_Y_pilot'] = self.modelo.ecs_modelo[0][1]
            fdm['A_Z_pilot'] = self.modelo.ecs_modelo[0][2]

            fdm['beta']  = self.modelo.fuselaje.be
            fdm['alpha'] = self.modelo.fuselaje.al

            fdm['vcas'] = sqrt(self.modelo.u**2 + self.modelo.v**2 +\
                    self.modelo.w**2)/0.3048
            fdm['rpm'][0] = 20.0

            # Guardamos resultado
            self.props[0]['fdm'] = fdm

            # mandamos la informacion del modelo
            self.sendFDM()
```

B.3. modelo.py

```
# -*- coding: latin-1 -*-
"""
Implementa la clase Modelo, encargada del calculo de las fuerzas del helicóptero
"""

from pylab import *
from numarray import *
from matematicas import *
import numarray.linear_algebra as la

from rotor import Rotor
from rotor_cola import RotorCola
from motor import Motor
from fuselaje import Fuselaje
from estabilizador import EstabilizadorHorizontal, EstabilizadorVertical
from controles import Controles
from geodesia import Geoide
from colision import Tren
from historial import *

class Modelo:
    """
    La clase principal de la simulacion:

    1º Contiene a todos los subsistemas:
        rotor
        rotor_cola
        fuselaje
        estabilizador_horizontal
        estabilizador_vertical
        motor
        controles

    2º Calcula las ecuaciones del tipo
        ecs_modelo[1]*(dx/dt) = ecs_modelo[0]

    3º Para llamar a calcula_ecs_modelo hay que asegurarse de tener
    los valores correctos en el vector de estado:
```

```
        Velocidad en ejes cuerpo
            self.u
            self.v
            self.w
        Velocidad angular
            self.p
            self.q
            self.r
        Cuaternio
            self.q0
            self.q1
            self.q2
            self.q3
        Controles:
            self.controles.th0
            self.controles.th1c
            self.controles.th1s
        Ambiente:
            self.ro
            self.hG
            self.Vw_i
        Posicion:
            self.x_i
            self.y_i
            self.z_i
        Motor/Rotor:
            self.Om
            self.Q1
            self.DTQ1
    """

    def __init__(self, reloj=None):
        """
        Crea los subsistemas y un reloj para controlar el tiempo
        """

        self.rotor = Rotor(self)
        self.rotor_cola = RotorCola(self)
        self.motor = Motor(self)
        self.fuselaje = Fuselaje(self)
        self.estabilizador_horizontal = EstabilizadorHorizontal(self)
        self.estabilizador_vertical = EstabilizadorVertical(self)
        self.controles = Controles(self)
        self.geoide = Geoide()
        self.tren = Tren(self)

        if reloj == None:
            self.reloj = Reloj()
        else:
            self.reloj = reloj

        # control de tiempo
        self.t = 0.0
        # altura del suelo
        self.hG = 0.0
        # Inicializamos con un valor cualquiera
        self.geoide.ejesLocales(0.0, 0.0, 0.0)

        self.trim = False

        # Cada variable de los subsistemas tiene una cadena de texto que el
        # simbolo mediante el cual se le hace referencia inequivocamente.
```

```
        self.__logstr = {
                "Om":       ("modelo", "Om"),
                "u":        ("modelo", "u"),
                "v":        ("modelo", "v"),
                "w":        ("modelo", "w"),
                "p":        ("modelo", "p"),
                "q":        ("modelo", "q"),
                "r":        ("modelo", "r"),
                "h":        ("modelo", "h"),
                "x_i":      ("modelo", "x_i"),
                "y_i":      ("modelo", "y_i"),
                "z_i":      ("modelo", "z_i"),
                "q0":       ("modelo", "q0"),
                "q1":       ("modelo", "q1"),
                "q2":       ("modelo", "q2"),
                "q3":       ("modelo", "q3"),
                "th":       ("modelo", "th"),
                "fi":       ("modelo", "fi"),
                "ch":       ("modelo", "ch"),
                "alt":      ("modelo", "alt"),
                "lon":      ("modelo", "lon"),
                "lat":      ("modelo", "lat"),

                "la0":      ("rotor", "la0"),
                "la1c_w":   ("rotor", "la1c_w"),
                "la1s_w":   ("rotor", "la1s_w"),
                "XR_b":     ("rotor", "Xr_b"),
                "YR_b":     ("rotor", "Yr_b"),
                "ZR_b":     ("rotor", "Zr_b"),
                "LR_b":     ("rotor", "Lr_b"),
                "MR_b":     ("rotor", "Mr_b"),
                "NR_b":     ("rotor", "Nr_b"),
                "cT":       ("rotor", "cT"),
                "th1c_w":   ("rotor", "th1c_w"),
                "th1s_w":   ("rotor", "th1s_w"),
                "be0":      ("rotor", "be0"),
                "be1c_w":   ("rotor", "be1c_w"),
                "be1s_w":   ("rotor", "be1s_w"),

                "Xf_b":     ("fuselaje", "Xf"),
                "Yf_b":     ("fuselaje", "Yf"),
                "Zf_b":     ("fuselaje", "Zf"),
                "Lf_b":     ("fuselaje", "Lf"),
                "Mf_b":     ("fuselaje", "Mf"),
                "Nf_b":     ("fuselaje", "Nf"),
                "alf":      ("fuselaje", "al"),
                "bef":      ("fuselaje", "be"),

                "th0":      ("controles", "th0"),
                "th1c":     ("controles", "th1c"),
                "th1s":     ("controles", "th1s"),
                "th0T":     ("controles", "th0T"),
                "et0p":     ("controles", "l_et0p"),
                "et1sp":    ("controles", "l_et1sp"),
                "et1cp":    ("controles", "l_et1cp"),
                "etpp":     ("controles", "l_etpp"),

                "Q1":       ("modelo", "Q1"),
                "DTQ1":     ("modelo", "DTQ1"),

                "QT":       ("rotor_cola", "QT"),
                "la0T":     ("rotor_cola", "la0T"),
```

```
                "cTT":      ("rotor_cola", "cTT"),
                "XT_b":     ("rotor_cola", "XT_b"),
                "YT_b":     ("rotor_cola", "YT_b"),
                "ZT_b":     ("rotor_cola", "ZT_b"),
                "LT_b":     ("rotor_cola", "LT_b"),
                "MT_b":     ("rotor_cola", "MT_b"),
                "NT_b":     ("rotor_cola", "NT_b"),

                "Xfn_b":    ("estabilizador_vertical", "Xfn_b"),
                "Yfn_b":    ("estabilizador_vertical", "Yfn_b"),
                "Zfn_b":    ("estabilizador_vertical", "Zfn_b"),
                "Lfn_b":    ("estabilizador_vertical", "Lfn_b"),
                "Mfn_b":    ("estabilizador_vertical", "Mfn_b"),
                "Nfn_b":    ("estabilizador_vertical", "Nfn_b"),
                "alfn":     ("estabilizador_vertical", "al"),
                "befn":     ("estabilizador_vertical", "be"),

                "Xtp_b":    ("estabilizador_horizontal", "Xtp_b"),
                "Ytp_b":    ("estabilizador_horizontal", "Ytp_b"),
                "Ztp_b":    ("estabilizador_horizontal", "Ztp_b"),
                "Ltp_b":    ("estabilizador_horizontal", "Ltp_b"),
                "Mtp_b":    ("estabilizador_horizontal", "Mtp_b"),
                "Ntp_b":    ("estabilizador_horizontal", "Ntp_b"),
                "altp":     ("estabilizador_horizontal", "al"),
                "betp":     ("estabilizador_horizontal", "be"),
        }


    ###########################################################################
    #   INICIALIZACION DE VARIABLES                                           #
    ###########################################################################
    def init(self):
        """
        Inicializa los subsistemas
        """

        self.rotor.init()
        self.rotor_cola.init()
        self.motor.init()
        self.fuselaje.init()
        self.estabilizador_horizontal.init()
        self.tren.init()
        self.controles.init()

    ###########################################################################
    #   UTILIDADES                                                            #
    ###########################################################################
    def euler(self, ch, th, fi):
        """
        Dados los angulos de Euler actualiza el cuaternio de rotacion
        """

        ( self.q0,
          self.q1,
          self.q2,
          self.q3 )= Quaternion.euler(
                  ch, th, fi)*Quaternion.rot(pi, 0.0, 1.0, 0.0)


    ###########################################################################
    #   PASO DE TIEMPO                                                        #
```

```
    ###########################################################################
    def paso(self, x, t):
        """
        Devuelve el vector f(x, t) tal que: dx/dt = f(x,t).
        Esta rutina es necesaria para el integrador numerico.
        """

        self.t = t
        ( self.u,
          self.v,
          self.w,
          self.p,
          self.q,
          self.r,
          self.q0,
          self.q1,
          self.q2,
          self.q3,
          self.Om,
          self.DTQ1,
          self.Q1,
          self.x_i,
          self.y_i,
          self.z_i ) = x

        self.calcula_ecs_modelo()
        ecs_0, ecs_1 = self.ecs_modelo
        self.f = dot(
            la.inverse(ecs_1),
            ecs_0
        )

        return self.f

    def avanza(self, t):
        """
        Avanza el estado del modelo hasta el instante de tiempo t
        """
        if self.tren.colision == 0:
            # Guardamos el estado para el tren antes de dar un paso de integracion
            self.tren.x[0:6] = (self.u, self.v, self.w, self.p, self.q, self.r)

            self.tren.x[6] = self.p0
            self.tren.x[7] = self.p1
            self.tren.x[8] = self.p2
            self.tren.x[9] = self.p3

            self.tren.x[10] = self.x_l
            self.tren.x[11] = self.y_l
            self.tren.x[12] = self.z_l - self.hG

            self.tren.t = self.t

        self.tren.FC = [0.0, 0.0, 0.0]
        self.tren.MC = [0.0, 0.0, 0.0]
        self.tren.FB = self.F0 + self.M*self.G
        self.tren.MB = self.M0

        # Damos un paso de integracion
        self.integrador(t)
        # Damos un paso de colision
        xc0, yc0, zc0 = self.tren.x[10:13]
```

```
        self.tren.step(t)
        xc1, yc1, zc1 = self.tren.x[10:13]
        # tren.step afecta a:
        # tren.x, tren.t, tren.F, tren.colision


        if self.tren.colision:
            q0 = self.tren.x[6]
            q1, q2, q3 = dot(self.LIL, self.tren.x[7:10])

            # !! Parece ser que se acumula un error numerico
            # El codigo es formalmente correcto
#            xc, yc, zc = dot(self.LCL,
#                     [self.tren.x[10],
#                         self.tren.x[11],
#                         self.tren.x[12] + self.hG]) + self.Ol
#            xi, yi, zi = self.geoide.cart2loc(xc, yc, zc)

            # Para evitar el error numerico calulamos los desplazamientos
            # en ejes inerciales
            dxi, dyi, dzi = dot(self.LIL, [xc1 - xc0, yc1 - yc0, zc1 - zc0])
            xi = self.x_i + dxi
            yi = self.y_i + dyi
            zi = self.z_i + dzi

            # No es correcto, pero por lo menos no se acumula
            # error
#            xi, yi, zi = dot(self.LIL,
#                     [self.tren.x[10],
#                         self.tren.x[11],
#                         self.tren.x[12] + self.hG])

            x = array(shape=(16,), type='Float64')
            x[0:6  ] = self.tren.x[0:6]
            x[6:10 ] = (q0, q1, q2, q3)
            x[13:16] = (xi, yi, zi)
            x[10   ] = self.integrador.x0[10]
            x[11   ] = self.integrador.x0[11]
            x[12   ] = self.integrador.x0[12]

            ( self.u, self.v, self.w,
                    self.p, self.q, self.r,
                    self.q0, self.q1, self.q2, self.q3,
                    self.Om, self.DTQ1, self.Q1,
                    self.x_i, self.y_i, self.z_i ) = x

            self.t = self.tren.t

            self.calcula_preparativos()
            self.calcula_FM()

            F = array(shape=(16, ), type='Float64')
            F[0:6  ] = self.tren.F[0:6]
            F[6    ] = self.tren.F[6]
            F[7:10 ] = dot(self.LIL, self.tren.F[7:10 ])
            F[13:16] = dot(self.LIL, self.tren.F[10:13])
            F[10   ] = self.integrador.F0[10]
            F[11   ] = self.integrador.F0[11]
            F[12   ] = self.integrador.F0[12]

#            self.integrador.init(self.paso, self.tren.t, x)
            self.integrador.x0 = x
```

```
            self.integrador.F0 = F
            self.integrador.t0 = self.tren.t
            self.integrador.ci = 0


        # Comprobamos si se producen colisiones
#        self.colisiones()

        self.t = t

    def reset(self):
        """
        Coloca el modelo en unas condiciones estandard.
        """
        self.u = 0.0
        self.v = 0.0
        self.w = 0.0
        self.p = 0.0
        self.q = 0.0
        self.r = 0.0

        self.Om = 0.0
        self.Q1 = 0.0
        self.DQ1 = 0.0

        self.x_i = 0.0
        self.y_i = 0.0
        self.z_i = self.fuselaje.ht

        self.nmotores = 0.0

    def calcula_x(self):
        """
        Actualiza el vector de estado.
        """

        self.x = array([
            self.u,
            self.v,
            self.w,
            self.p,
            self.q,
            self.r,
            self.q0,
            self.q1,
            self.q2,
            self.q3,
            self.Om,
            self.DTQ1,
            self.Q1,
            self.x_i,
            self.y_i,
            self.z_i], type='Float64')

    def calcula_LBI(self):
        """
        Calcula la matriz de cambio de ejes inercia a ejes cuerpo y
        viceversa.
        """

        # La matriz de cambio de base es la transpuesta de la matriz de
        # rotacion
```

```
        self.LIB = Quaternion(
                self.q0, self.q1, self.q2, self.q3).toMatrix()
        self.LBI = transpose(self.LIB)

    def calcula_G(self):
        """
        Calcula la aceleracion de la gravedad en ejes cuerpo.
        """

        self.Gx, self.Gy, self.Gz = self.G = dot(self.LBI, [0.0, 0.0, -9.81])

    def calcula_Vw_i(self):
        """
        Calcula la velocidad del viento en ejes inerciales a partir
        de modulo y azimuth
        """
        self.Vw_i = dot(self.LIL,
                [ self.viento_vel*cos(self.viento_dir),
                  self.viento_vel*sin(self.viento_dir),
                  0 ])

    def calcula_Vrw_b(self):
        """
        Velocidad relativa al viento en ejes cuerpo.
        """
        self.Vrw_b = [self.u, self.v, self.w] - dot(self.LBI, self.Vw_i)


    def calcula_ro(self):
        """
        Calcula la densidad a considerando al aire como gas perfecto.
        """
        self.ro = self.P/(self.T*286.9)

    def calcula_velocidades(self):
        """
        Diversas velocidades para pasarlas a FlightGear.
        """
        p = self.p
        q = self.q
        r = self.r

        Sfi, Cfi = sin(self.fi), cos(self.fi)
        Sth, Cth = sin(self.th), cos(self.th)

        if Cth!=0:
            self.DTfi = p + (q*Sfi + r*Cfi)*Sth/Cth
        else:
            self.DTfi = 0.0

        self.DTth = q*Cfi - r*Sfi
        if Sth!=0:
            self.DTch = (q*Sfi + r*Cfi)*Cth/Sth
        else:
            self.DTch = 0.0

    def colisiones(self):
        """
        Esta funcion se tiene que llamar en caso de que se estrelle el
        helicoptero, o se presente un valor absurdo, ya que coloca
        al helicoptero en el suelo con valores razonables.
        """
```

```
        x = self.integrador.x1
        u_i, v_i, w_i = dot(self.LIB, [x[0], x[1], x[2]])
        if self.alt - self.fuselaje.ht - self.hG < 0 and w_i<0.0:
            # No penetrabilidad y no deslizamiento
            x[0] = x[1] = x[2] = 0.0
            x[3] = x[4] = x[5] = 0.0

            # Colocamos el helicoptero en posicion horizontal
            E = Quaternion(x[6], x[7], x[8], x[9]).toEuler()
            x[6], x[7], x[8], x[9] = Quaternion.euler(E[0], 0.0, pi)

            # Y corregimos el error de penetracion
            self.alt = self.hG + self.fuselaje.ht
            x[15] = self.geoide.geod2loc(
                    self.lat,
                    self.lon,
                    self.alt )[2] - 0.01

            # reseteamos el integrador
            self.integrador.init(F = self.paso,
                    ti = self.t,
                    xi = x)

    def calcula_FM(self):
        """
        Calcula las fuerzas y momentos totales que actuan sobre el C.M del
        helicoptero sumando todas las fuerzas y momentos de los subsistemas.
        """

        # A lo mejor queremos controles fijos, por ejemplo
        # los calculados por el trimado
        if not self.trim:
            self.controles.calcula_controles()

        # Calculamos las fuerzas de los diferentes subsistemas
        self.rotor.calcula_FMR()
        self.rotor_cola.calcula_FMT()
        self.fuselaje.calcula_FMf()
        self.estabilizador_horizontal.calcula_FMtp()
        self.estabilizador_vertical.calcula_FMfn()

        FM0 = ( self.rotor.FMR[0] +
                self.rotor_cola.FMT[0] +
                self.fuselaje.FMf[0] +
                self.estabilizador_vertical.FMfn[0] +
                self.estabilizador_horizontal.FMtp[0]
        )

        FM1 = ( self.rotor.FMR[1] +
                self.rotor_cola.FMT[1] +
                self.fuselaje.FMf[1] +
                self.estabilizador_vertical.FMfn[1] +
                self.estabilizador_horizontal.FMtp[1]
        )

        self.F0 = FM0[0:3]
        self.M0 = FM0[3:6]
        self.F1 = FM1[0:3]
        self.M1 = FM1[3:6]

    def calcula_geo(self):
```

```
        self.lat, self.lon, self.alt = \
                self.geoide.loc2geod(
                        self.x_i,
                        self.y_i,
                        self.z_i)

    def calcula_locales(self):
        Slat, Clat = sin(self.lat), cos(self.lat)
        Slon, Clon = sin(self.lon), cos(self.lon)
        # matriz ejes locales-cartesianos geocentricos
        LLC = array( [
            [  Slat*Clon,  Slat*Slon, -Clat ],
            [ -Slon,       Clon,       0    ],
            [  Clat*Clon,  Clat*Slon,  Slat ]] ,type='Float64')

        self.LLC = LLC
        # matriz ejes cartesianos geocentricos-locales
        self.LCL = transpose(LLC)

        # matriz ejes locales-ejes inerciales
        LLI = dot(LLC, self.geoide.Lcl)

        # origen en geocentricas de los ejes locales
        self.Ol = self.geoide.geod2cart(self.lat, self.lon, 0.0)

        r_c = self.geoide.loc2cart(self.x_i, self.y_i, self.z_i)
        # Coordenadas locales
        ( self.x_l,
                self.y_l,
                self.z_l ) = dot(LLC, r_c - self.Ol)

        # Cuaternio en ejes locales
        self.p0 = self.q0
        ( self.p1,
                self.p2,
                self.p3 ) = dot(LLI, [self.q1, self.q2, self.q3])

        self.LLI = LLI
        self.LIL = transpose(LLI)

    def calcula_euler(self):
        Q = Quaternion(self.p0, self.p1, self.p2, self.p3 )
        E = Q.toEuler()
        self.ch =  pi - E[0]
        self.th = -E[1]
        self.fi =  pi + E[2]

        # eps y 2pi + eps son equivalentes, pero no para
        # el sistema de control, asi que metemos los valores
        # en el rango -pi, pi
        if self.th > pi:
            self.th -= 2*pi
        elif self.th<-pi:
            self.th += 2*pi

        if self.fi > pi:
            self.fi -= 2*pi
        elif self.fi<-pi:
            self.fi += 2*pi

    def calcula_preparativos(self):
        """
```

```
        Suponiendo conocidas las magnitudes del vector de estado
        calcula el resto de magnitudes que faltan para ya poder pasar
        al calculo de las fuerzas y moentos y las ecuaciones
        del motor y de solido rigido.
        """

        # Posicion geodesica
        self.calcula_geo()

        # Ejes locales
        self.calcula_locales()

        # Angulos de Euler
        self.calcula_euler()

        # Orientacion de los ejes cuerpo respecto a los inerciales
        self.calcula_LBI()

        # Densidad
        self.calcula_ro()

        # Velocidad del viento en ejes inerciales
        self.calcula_Vw_i()

        # Velocidad relativa al viento en ejes cuerpo
        self.calcula_Vrw_b()

        # Varias velocidades
        self.calcula_velocidades()

        # Gravedad
        self.calcula_G()

        # renormalizamos el cuaternio de rotacion
        modq = sqrt(self.q0**2 + self.q1**2 + self.q2**2 + self.q3**2)
        self.q0 /=modq
        self.q1 /=modq
        self.q2 /=modq
        self.q3 /=modq

    def calcula_ecs_modelo(self):
        """
        Calcula las variables ecs_0 y ecs_1 tal que la ecuacion a integrar
        para el vector de estado es:
            ecs_1*(dx/dt) = ecs_0

        En el proceso calcula tambien multitud de variables intermedias
        """

        if not self.tren.colision:
            self.calcula_preparativos()
            # Calculamos fuerzas y momentos de subsistemas
            self.calcula_FM()

        # Calculamos ecuaciones del motor
        self.motor.calcula_ecs_motor()

        # alias para facilitar la legibilidad del codigo
        Ixx = self.Ixx
        Iyy = self.Iyy
        Izz = self.Izz
        Ixz = self.Ixz
```

```
        M = self.M

        u = self.u
        v = self.v
        w = self.w
        p = self.p
        q = self.q
        r = self.r

        q0 = self.q0
        q1 = self.q1
        q2 = self.q2
        q3 = self.q3

        ########################################################################
        #   TERMINOS DEPENDIENTES LINEALES                                     #
        ########################################################################
        ecs_1 = zeros(shape=(16, 16), type='Float64')

        # Ecuaciones de velocidad lineal en ejes cuerpo
        ecs_1[0:3, 0:3] = [
                [ 1.0, 0.0, 0.0 ],
                [ 0.0, 1.0, 0.0 ],
                [ 0.0, 0.0, 1.0 ]]
        # Ecuaciones de velocidad angular en ejes cuerpo
        ecs_1[3:6, 3:6] = [
                [  Ixx, 0.0, -Ixz ],
                [  0.0, Iyy,  0.0 ],
                [ -Ixz, 0.0,  Izz ]]
        # Relaciones cinematicas del cuaternio cambio ejes cuerpo-inerciales
        ecs_1[6:10, 6:10] = [
                [ 1.0, 0.0, 0.0, 0.0 ],
                [ 0.0, 1.0, 0.0, 0.0 ],
                [ 0.0, 0.0, 1.0, 0.0 ],
                [ 0.0, 0.0, 0.0, 1.0 ]]

        # Ecuaciones de la posicion del avion
        ecs_1[13:16, 13:16] = [
                [ 1.0, 0.0, 0.0 ],
                [ 0.0, 1.0, 0.0 ],
                [ 0.0, 0.0, 1.0 ]]

        # Añadimos los terminos dependientes a las ecuaciones
        ecs_1[0:3, :]   -= self.F1/M
        ecs_1[3:6, :]   -= self.M1
        ecs_1[10:13, :] += self.motor.ecs_motor[1]

        ########################################################################
        #   TERMINOS INDEPENDIENTES  NO LINEALES                               #
        ########################################################################
        ecs_0 = zeros( shape = (16,), type = 'Float64')

        # Aceleraciones y momentos de inercia
        ecs_0[0] = -( w*q - v*r )
        ecs_0[1] = -( u*r - w*p )
        ecs_0[2] = -( v*p - u*q )
        ecs_0[3] =  ( Iyy - Izz )*q*r + Ixz*p*q
        ecs_0[4] =  ( Izz - Ixx )*r*p + Ixz*( r**2 - p**2 )
        ecs_0[5] =  ( Ixx - Iyy )*p*q + Ixz*q*r

        # Gravedad y fuerzas de rotor, cola, etc...
```

```
        ecs_0[0:3] += self.G + self.F0/M
        ecs_0[3:6] += self.M0

        # Lado derecho de las ecuaciones del quaternio
        # Mucho ojo que hay que pasar la velocidad a angular a ejes inerciales
        p_i, q_i, r_i = dot(self.LIB, [p, q, r])

        ecs_0[6] = -0.5*(  q1*p_i + q2*q_i + q3*r_i )
        ecs_0[7] = -0.5*( -q0*p_i - q3*q_i + q2*r_i )
        ecs_0[8] = -0.5*(  q3*p_i - q0*q_i - q1*r_i )
        ecs_0[9] = -0.5*( -q2*p_i + q1*q_i - q0*r_i )

        # Lado derecho de las ecuaciones del motor
        ecs_0[10:13] = self.motor.ecs_motor[0]

        # Lado derecho de las ecuaciones cinematicas
        ecs_0[13:16] = dot(self.LIB, [u,v,w])

        ###########################################################################
        #            CONTACTO CON EL SUELO                                        #
        ###########################################################################
        # ESTO NECESITA MUCHISIMA MEJORA
        # si tocamos el suelo, y vamos hacia el
#        if (self.alt - self.fuselaje.ht - self.hG) <= 0.0 and w>=0:
#            # No penetrabilidad
#            if ecs_0[2]>0:
#                ecs_0[2] = 0.0                      # Fuerzas verticales
#            # No deslizamiento
#            ecs_0[0] = ecs_0[1] = 0.0               # Fuerzas tangenciales
#            ecs_0[3] = ecs_0[4] = ecs_0[5] = 0.0    # Momentos
#
#            ecs_1[0:6] = zeros(shape=(6, len(self.x)), type='Float64')
#            ecs_1[0:6, 0:6] = identity(6)

        #-------------------------------------------------------------------#
        self.ecs_modelo = ecs_0, ecs_1
        #-------------------------------------------------------------------#

        # logging
        ( self.ecs_u,
                self.ecs_v,
                self.ecs_w,
                self.ecs_p,
                self.ecs_q,
                self.ecs_r ) = ecs_0[:6]

        ( self.ecs_q0,
                self.ecs_q1,
                self.ecs_q2,
                self.ecs_q3 ) = ecs_0[6:10]

        self.ecs_Om, self.ecs_DTQ1, self.ecs_Q1 = ecs_0[10:13]



    ###########################################################################
    #            ACCESO MEDIANTE SIMBOLOS                                     #
    ###########################################################################
    def __path(self, s):
        return self.__logstr[s]

    def get(self, s):
```

```
        p = self.__path(s)
        try:
            if p[0] == 'modelo':
                x = getattr(self, p[1])
            else:
                x = getattr(getattr(self, p[0]), p[1])
        except AttributeError:
            x = 0.0
        return x

    def set(self, s, v):
        p = self.__path(s)
        if p[0] == 'modelo':
            setattr(self, p[1], v)
        else:
            setattr(getattr(self, p[0]), p[1], v)

    ###########################################################################
    #            SERIES TEMPORALES                                            #
    ###########################################################################
    def series(self, T, et0p, et1sp, et1cp, etpp, dt=0.03, *args):
        """
        La siguiente funcion calcula la respuesta de las magnitudes pedidas
        en args (simbolos) desde el instante actual (self.t) hasta el instante
        T, dando un paso de tiempo dt. et0p, et1sp, et1cp, etpp son funciones
        del tiempo o valores constantes. Devuelve los resultados en un
        diccionario que contiene por lo menos las llaves 't' con los pasos
        de tiempo dados, 'et0p', 'et1sp', 'et1cp' y 'etpp' con los valores
        en cada instante de tiempo y ademas una llave por cada simbolo que se
        pidiese calcular.

        Un ejemplo de una posible sesion en python seria:

        In [1]: from Lynx import modelo
        In [2]: modelo.init()
        In [3]: tr = modelo.trimado(0, 0, 0, 0.0, 100., 1.225)
                  ...    (Informacion soltada por el proceso de trimado)
                  ...
                  ...
        In [4]: modelo.t = 0.0
        In [5]: s = modelo.series(0.2, 0, 0, 0, 0, 0.03, 'zi')
        In [6]: s['t']
        Out[6]: array([ 0.03,  0.06,  0.09,  0.12,  0.15,  0.18,  0.21])
        In [7]: s['zi']
        Out[7]:
        [100.0,
         100.0,
         99.986765709951754,
         99.964728999868001,
         99.933898156358978,
         99.894286155391967,
         99.845908960219973]
        """

        # Hay que calcular los controles
        self.trim = False

        def reloj_tonto():
            return self.t

        # Ya tenemos el instante actual
        # El instante final es T, incluido
```

```
        DT = arange(self.t + dt, T + dt, dt)

        # Diccionario con los resultados
        r = {
            'et0p':  [],
            'et1sp': [],
            'et1cp': [],
            'etpp':  []
        }
        for k in args:
            r[k] = []

        # Si alguno de los controles es una constante lo
        # transformamos en una funcion constante
        def func(x):
            if not hasattr(x, '__call__'):
                return lambda y: x
            else:
                return x

        f_et0p  = func(et0p )
        f_etpp  = func(etpp )
        f_et1sp = func(et1sp)
        f_et1cp = func(et1cp)

        # Preparamos el integrador
        self.controles.et0p.append (f_et0p  (self.t))
        self.controles.et1sp.append(f_et1sp (self.t))
        self.controles.et1cp.append(f_et1cp (self.t))
        self.controles.etpp.append (f_etpp  (self.t))

        self.integrador.init(
                    F  = self.paso,
                    ti = self.t,
                    xi = array([
                        self.u,
                        self.v,
                        self.w,
                        self.p,
                        self.q,
                        self.r,
                        self.q0,
                        self.q1,
                        self.q2,
                        self.q3,
                        self.Om,
                        self.DTQ1,
                        self.Q1,
                        self.x_i,
                        self.y_i,
                        self.z_i]
                    )
        )


        for t in DT:
            # Controles en el instante actual
            self.controles.et0p.append (f_et0p  (t), t)
            self.controles.et1sp.append(f_et1sp (t), t)
            self.controles.et1cp.append(f_et1cp (t), t)
            self.controles.etpp.append (f_etpp  (t), t)
```

```
            # Avanzamos al siguiente instante
            self.avanza(t)
            self.t = t

            # Guardamos resultados
            for k in r.keys():
                r[k].append(self.get(k))
        r['t'] = DT
        return r

    def trimado(self, V_t, ga_t, be_t, Oma_t, z_t, ro_t, metodo='Fijo'):
        """
        Calcula el trimado del helicoptero para los parametros:
            V_t:   Velocidad del helicoptero.
            ga_t:  Angulo de subida
            be_t:  Angulo de resbalamiento.
            Oma_t: Velocidad de giro
            z_t:   Altura
            ro_t:  Densidad del aire
            metodo: 'Fijo' o 'Newton', modifica el método numérico.
                      Se recomienda MUCHO utilizar Fijo porque Newton no
                      funciona bien todavia.
        """

        # motores encendidos!!!
        self.motor.nmotores = 2
        self.motor.start = False
        ##########################################################################
        #   ALIAS                                                                #
        ##########################################################################
        # Para no tener self-verborrea se acortan los siguientes nombres
        M   = self.M
        Ixx = self.Ixx
        Iyy = self.Iyy
        Izz = self.Izz
        Ixz = self.Ixz

        tht = self.rotor.tht
        a0 =  self.rotor.a0
        s =   self.rotor.s

        # Para calcular los mandos del piloto
        c0 = self.controles.c0
        c1 = self.controles.c1
        S0 = self.controles.S0
        S1 = self.controles.S1
        S2 = self.controles.S2

        # Constantes de la TCMM
        ki =  (9./5.)**0.25
        knu = (5./4.)**0.25
        g =  9.81

        # Precalculamos senos y cosenos que permanecen constantes
        Cbe, Sbe = cos(be_t), sin(be_t)
        Cga, Sga = cos(ga_t), sin(ga_t)
        CK = cos(self.rotor_cola.K)
        SK = sin(self.rotor_cola.K)



        ##########################################################################
```

```
#   ESTIMACION VALORES INICIALES                                          #
##########################################################################
# Coeficientes del fuselaje, practicamente no varian
cDf, cLf, cYf, clf, cmf, cnf = self.fuselaje.aero.coefs(0.0, 0.0)

# Angulos de euler:
# th compensa la resistencia del fuselaje
# fi compensa la aceleracion centrifuga
th = -0.5*ro_t*V_t**2*self.fuselaje.Sp*cDf/(M*g)
fi =  atan(Oma_t*V_t/9.81)

# Hay que iniciar una serie de variables, aunque sea a cero
# Fuerzas y momentos del rotor de cola
XT = YT = ZT = 0.0
LT = MT = NT = 0.0
QT = 0.0

# Fuerzas y momentos del rotor
XR = YR = LR = MR = NR = 0.0
LRh_b = NRh_b = YRh_b = 0.0
LRh_w = NRh_w = QRh_w= YRh_w = 0.0

# Velocidad inducida
la1c_w = la1s_w = 0.0
la0 = la0NG = 0.006

# Batimiento del rotor
be0 = be1c_w = be1s_w = 0.0
# Batimiento del rotor de cola
be1cT = be1sT = be1sT_w = 0.0

# Velocidad de giro del rotor
Om = self.motor.Omi
# Coeficiente de traccion
cT = M*g/(ro_t*(Om*self.rotor.R)**2*pi*self.rotor.R**2)

# Pasos
th0 = th1s = th1c = th0T = 0.0
th1c_w = th1s_w = 0.0
# Controles que necesita hacer el piloto
et0p = et1cp = et1sp = etpp = 0.0

##########################################################################
#   FUNCIONES AUXILIARES                                                 #
##########################################################################
# Esta funcion, que se llama en dos puntos del programa calcula el
# vector be0, la1s_w, la1c_w, th0, th1s_w, th1c_w
def f_controles():
    # Velocidades en el rotor
    C = (nu/knu)**2 + (1/knu**2 - 1/ki**2)*nuz**2 +\
            ((nuz - la0NG)/ki)**2
    VT = sqrt(C)
    barV = VT*(1 - la0NG*(nuz - la0NG)/ki**2/C)

    # Constantes del batimiento
    labe2 = 1. + self.rotor.Kbe/( self.rotor.Ibe*Om**2)
    ga = ro_t*self.rotor.c*a0*self.rotor.R**4/self.rotor.Ibe

    #
    # Batimiento y controles del rotor principal
    #
```

```
A_bebe = array([
    [ 8*labe2/ga, 0,              0                 ],
    [ 4./3.*nu,   8*(labe2-1)/ga, 1 + 0.5*nu**2     ],
    [ 0,          -1. + 0.5*nu**2, 8*(labe2 - 1)/ga  ]],
    type='Float64')

A_bela = array([
    [ 2./3.*nu, 0. ],
    [ 0.,       1. ],
    [ 1.,       0. ]], type='Float64')

# Lado derecho
# Influencia del paso sobre el batimiento
A_beth = array([
    [
        1. + nu**2,
        4./5. + 2./3.*nu**2,
        4./3.*nu,
        0.
    ],
    [
        0.,
        0.,
        0.,
        1. + 0.5*nu**2
    ],
    [
        8./3.*nu,
        2*nu,
        1. + 1.5*nu**2,
        0.
    ]], type='Float64')

# Influencia de la aceleracion angular sobre el batimiento
A_beDChiom = 8./ga*array([
    [ 0., 0. ],
    [ 0., 1. ],
    [ 1., 0. ]], type='Float64')

# Influencia de la velocidad angular sobre el batimiento
A_beom = array([
    [ 2./3.*nu,  0       ],
    [ 16./ga,    1.      ],
    [ 1.,       -16./ga  ]], type='Float64')

# Influencia de la velocidad inducida media sobre el batimiento
A_bela0 = array([
    4./3.,
    0.,
    2*nu], type='Float64')
# Matrices de las 2 ecuaciones de velocidad inducida
# Matriz L de Pitt-Peters
X = tan(abs(xi/2.))
K = array([
        [ 0,                2*(1. + X**2)/barV, 0.                  ],
        [ 15.*pi/(64.*VT)*X, 0.,                2.*(1. - X**2)/barV ]]
        , type='Float64')

# Matriz de fuerzas y momentos aerodinamicos
# N = { F0/2, M1s, M1c }
Nth = array([
    [
```

```
            2./3. + nu**2,
            0.5*(1. + nu**2),
            nu,
            0.
        ],
        [
            2./3.*nu,
            0.5*nu,
            0.25*(1. + 1.5*nu**2),
            0
        ],
        [
            0.,
            0.,
            0.,
            0.25*(1. + 0.5*nu**2)
        ]], type='Float64')

    Nbe = array([
        [ 0.,   0.,                       0.                          ],
        [ 0.,   0.25*(1. - 0.5*nu**2), 0.                             ],
        [-nu/3.,                          0, -0.25*(1. + 0.5*nu**2) ]],
        type='Float64')

    Nla = -0.5*array([
        [ nu,  0.  ],
        [ 0.5, 0.  ],
        [ 0.,  0.5 ]], type='Float64')

    Nom = -Nla

    Nla0 = array([
        1.,
        0.5*nu,
        0.], type='Float64')

    # Lado izquierdo de las ecuaciones de la velocidad inducida
    m = a0*s/4.
    A_labe = -m*dot(K, Nbe)
    A_lala =  identity(2) - m*dot(K, Nla)
    # Matriz de influencia de controles, velocidad angular y velocidad
    # inducida uniforme sobre la velocidad inducida
    A_lath =  m*dot(K, Nth)
    A_laom =  m*dot(K, Nom)
    A_lala0 = m*dot(K, Nla0)
    # Matriz de influencia de la aceleracion angular sobre la velocidad
    # inducida
    A_laDChiom = zeros(shape=(2,2), type='Float64')

    lhs = array(shape=(6,6), type='Float64')
    lhs[0:3, 0  ] =  A_bebe[:, 0]
    lhs[0:3, 1:3] =  A_bela
    lhs[0:3, 3  ] = -A_beth[:, 0]
    lhs[0:3, 4:6] = -A_beth[:, 2:4]
    lhs[3:5, 0  ] =  A_labe[:, 0]
    lhs[3:5, 1:3] =  A_lala
    lhs[3:5, 3  ] = -A_lath[:, 0]
    lhs[3:5, 4:6] = -A_lath[:, 2:4]
    lhs[5,   0:6] =  array([0, 0, 0, 1./3. + nu**2/2., nu/2., 0],
            type='Float64')

    rhs = array(shape=(6), type='Float64')
```

```
            rhs[0:3] = (
                    -dot(A_bebe[:, 1:3], [be1c_w, be1s_w]) +
                        A_beth[:, 1]*self.rotor.tht +
                        dot(A_beom, [p_w, q_w]) +
                        (nuz-la0)*A_bela0
            )
            rhs[3:5] = (
                    -dot(A_labe[:, 1:3], [be1c_w, be1s_w]) +
                    A_lath[:, 1]*self.rotor.tht +
                    dot(A_laom, [p_w, q_w]) +
                    (nuz-la0)*A_lala0
            )
            rhs[5] = (
                    2*cT/(a0*s) -
                    nu/4.*p_w -
                    0.5*(nuz - la0) -
                    0.25*(1. + nu**2)*self.rotor.tht
            )

            # be0, la1s_w, la1c_w, th0, th1s_w, th1c_w
            return la.solve_linear_equations(lhs, rhs)

        # Esta funcion, que se llama en dos puntos del programa calcula
        # controles a partir de los pasos
        def f_etp(th0, th1s, th1c, th0T, p, q, r, th, fi, ch):
            et =  dot(la.inverse(c1),
            array([th0, th1s, th1c, th0T], type='Float64') - c0)
            print et
            # Controles del piloto a partir de los controles
            def f(x):
                th_0 = self.controles.th_0(x[1])
                fi_0 = self.controles.fi_0(x[2])
                ch_0 = self.controles.ch_0(x[3])
                return dot(S0, x) + dot(S1, [p, q, r]) +\
                        dot(S2, [th - th_0, fi - fi_0, ch - ch_0]) - et

            etp = solveNewtonMulti(f, [0., 0., 0., 0.], 100, 1e-5)

            return etp

        ########################################################################
        #   CONSTANTES NUMÉRICAS                                               #
        ########################################################################

        # Errores numericos: en ciertos puntos del programa si
        # abs(x)<eps0-->x = 0
        eps0 = 1e-3

        # Precision deseada en Om, fi y th
        eps_Om = 1e-3
        eps_fi = 1e-3
        eps_th = 1e-3

        # Numero maximo de iteraciones en los respectivos bucles
        Nmax_Om = 50
        Nmax_fi = 50
        Nmax_th = 200

        # Numero minimo de iteraciones en los respectivos bucles
        Nmin_Om = 3
        Nmin_fi = 3
        Nmin_th = 3
```

```
        # Amortiguadores para asegurar convergencia, valores
        # experimentales
        k_th = 1.0
        k_fi = 0.5
        k_Om = 1.0

        # Bandera para controlar el caso ga=pi/2, que numericamente
        # es problematico, si se detecta  entonces
        # se activa y se tiene en cuenta
        resb = False


        ##########################################################################
        #   COMIENZO DE LA ITERACION                                             #
        ##########################################################################

        # COMIENZA BUCLE OM #
        n_Om = 0        # Numero de iteraciones realizadas
        err_Om = 1      # error en la ultime iteracion
        while err_Om>eps_Om or n_Om<Nmin_Om:
            if n_Om>Nmax_Om:
                print "No converge Om, amortiguando el proceso!!"
                k_Om -=0.1
                # reseteamos el bucle
                n_Om = 0
                Om = self.motor.Omi
                if k_Om <0.2:
                    print ( "Muy amortiguado ya, no se puede alcanzar " +
                            "convergencia" )
                    print "Terminando trimado"
                    break

            # COMIENZA BUCLE FI #
            n_fi = 0
            err_fi = 1
            while err_fi>eps_fi or n_fi<Nmin_fi:
                if n_fi>Nmax_fi:
                    print "No converge fi, amortiguando el proceso!!"
                    k_fi -=0.1
                    # reseteamos el bucle
                    n_fi = 0
                    fi = atan(Oma_t*V_t/9.81)
                    if k_fi <0.2:
                        print ( "Muy amortiguado ya, no se puede alcanzar " +
                                "convergencia" )
                        print "Saltando trimado de fi"
                        k_th = 1.0
                        break

                # No cambia en todo el bucle TH
                Cfi, Sfi = cos(fi), sin(fi)

                # COMIENZA BUCLE TH #
                st = 0
                n_th = 0
                err_th = 1
                while err_th>eps_th or n_th<Nmin_th:
                    if n_th>Nmax_th:
                        print "No converge th, amortiguando el proceso!!"
                        k_th -= 0.2
                        # reseteamos el bucle
```

```
            n_th = 0
            th = -0.5*ro_t*V_t**2*self.fuselaje.Sp*cDf/(M*g)
            if k_th <0.2:
                print ( "Muy amortiguado ya, no se puede alcanzar "
                        + "convergencia" )
                print "Saltando trimado de theta"
                k_th = 1.0
                break
        Cth, Sth = cos(th), sin(th)
        ############################################################
        #   VELOCIDAD Y VELOCIDAD ANGULAR                          #
        ############################################################
        # Calculamos la velocidad angular
        p = -Oma_t*Sth
        q =  Oma_t*Sfi*Cth
        r =  Oma_t*Cfi*Cth
        # Calculamos la velocidad
        # Resolvemos la ecuacion de segundo orden para el seno
        # del angulo de tracking: a = b*cos(xi) + c*sin(xi)
        a = Sbe - Sga*Sfi*Cth
        b = Cga*Sth*Sfi
        c = Cga*Cfi
        # A*sin(xi)^2 + B*sin(xi) + C = 0
        A =  c**2 + b**2
        B = -2*a*c
        C =  a**2 - b**2
        # sin(xi) = (-B +- D)/(2A)
        D =  B**2 - 4*A*C
        # Manejamos errores numericos que nos echan del dominio
        if abs(A)<eps0:
            # Suponemos que ha pasado ga = pi/2, en cuyo caso
            # be no esta definido, en cualquier caso el valor
            # de xi no afecta el valor de la velocidad
            # Obligamos a=0 ajustando be
            Sbe = Sga*Sfi*Cth
            Sxi = 0.
            Cxi = 1.
            # Solo avisamos la primera vez
            if resb == False:
                print ">ga=pi/2? Dejando libre  al resbalamiento"
                resb = True
        else:
            # Corregimos error numerico, -1e-16 daria error
            if D<0:
                if -D<eps0:
                    D = 0.0
                else:
                    # La ecuacion no tiene solucion obligatoriamente
                    sys.exit("Error calculando el ángulo de " +
                             "tracking, raíz imaginaria")

            # Las dos raíces
            Sxi1 = (-B - sqrt(D))/(2.*A)
            Sxi2 = (-B + sqrt(D))/(2.*A)
            # Descartamos una, nos quedamos con la mas pequeña
            if abs(Sxi1) < abs(Sxi2):
                Sxi = Sxi1
            else:
                Sxi = Sxi2
            if Sxi>1:
                if (Sxi-1)<eps0:
                    Sxi = 1.0
```

```
                else:
                    sys.exit("Error calculando el ángulo de " +
                             "tracking, sin(xi)>1")
            Cxi2 = 1. - Sxi**2
            # Si es menor que 0 tiene que ser error numerico seguro
            if Cxi2 < 0:
                Cxi2 = 0.0
            Cxi = sqrt(Cxi2)

        # Velocidad en ejes cuerpo
        u = V_t*(Cth*Cga*Cxi - Sth*Sga)
        v = V_t*Sbe
        w = V_t*(Cga*Sth*Cfi*Cxi + Cga*Sfi*Sxi + Sga*Cfi*Cth)

        # Para aprovechar funciones ya disponibles
        # Cuaternio de rotacion de ejes inerciales a cuerpo
        _Q = Quaternion.rotacion([
            (pi, 0, 1, 0), # Ejes inerciales a auxiliares
            (th, 0, 1, 0), # Angulos de euler, ch = 0
            (fi, 1, 0, 0)])

        # Solo estas variables influyen en el calculo de las
        # fuerzas aerodinamicas
        self.u = u
        self.v = v
        self.w = w

        self.p = p
        self.q = q
        self.r = r

        self.q0 = _Q.q0
        self.q1 = _Q.q1
        self.q2 = _Q.q2
        self.q3 = _Q.q3

        self.Om = Om

        self.rotor.la0 = la0

        ##############################################################
        #   CALCULO DE EJES VIENTO                                   #
        ##############################################################
        # Calculamos ejes viento del rotor
        self.calcula_LBI()
        self.Vw_i = array([0.0, 0.0, 0.0], type='Float64')
        self.ro = ro_t
        self.calcula_Vrw_b()

        # Velocidad angular en ejes cuerpo
        W_b = array([p, q, r], type='Float64')

        # Velocidad del rotor relativa al viento, en ejes cuerpo
        Vhrw_b = self.Vrw_b +\
                 cross(W_b, [-self.rotor.xcg, 0.0, -self.rotor.hR])
        # en ejes rotor
        uhrw_h, vhrw_h, whrw_h = dot(self.rotor.LHB, Vhrw_b)

        # Velocidad angular en ejes rotor
        W_h = dot(self.rotor.LHB, W_b)

        # Velocidad relativa del rotor respecto al viento, en ejes
```

```
# viento
uhrw_w = sqrt(uhrw_h**2 + vhrw_h**2)
whrw_w = whrw_h

# Angulo que forman los ejes viento con los ejes rotor.
# Si no hay velocidad se supone 0
if uhrw_w!=0:
    Cchw = uhrw_h/uhrw_w
    Schw = vhrw_h/uhrw_w
else:
    if vhrw_h==0:
        Cchw = 1.0
        Schw = 0.0
    elif vhrw_h<0:
        Cchw =  0.0
        Schw = -1.0
    else:
        Cchw = 0.0
        Schw = 1.0

# Matriz cambio de base de ejes viento a ejes cuerpo
# nos es util para pasar las fuerzas y momentos a ejes
# cuerpo
LHW = array([
    [Cchw, -Schw, 0.0],
    [Schw,  Cchw, 0.0],
    [0.0,   0.0,  1.0]], type = 'Float64')
LBW = dot(self.rotor.LBH, LHW)
LWB = transpose(LBW)

# Parametros de avance
nu =  uhrw_w/( Om*self.rotor.R )
nuz = whrw_w/( Om*self.rotor.R )

# Velocidad angular adimensionalizada en ejes viento
p_w = (  Cchw*W_h[0] + Schw*W_h[1] )/Om
q_w = ( -Schw*W_h[0] + Cchw*W_h[1] )/Om

#############################################################
#   FUERZAS DE FUSELAJE Y ESTABILIZADORES                   #
#############################################################
# Vrw_b, [p,q,r] y [q0,q1,q2,q3] ya estan calculados,
# que es lo que cuenta
# Angulo de la estela, para el estabilizador horizontal
xi = self.rotor.xi = ang(
        abs(la0*Om*self.rotor.R - whrw_w), uhrw_w)

# Calculamos las fuerzas aerodinamicas y guardamos los
# resultados, en ejes cuerpo. Aprovechamos funciones ya
# declaradas.

# Fuselaje
self.fuselaje.calcula_FMf()
Xf, Yf, Zf, Lf, Mf, Nf = self.fuselaje.FMf[0]

self.estabilizador_horizontal.calcula_FMtp()
Xtp, Ytp, Ztp, Ltp, Mtp, Ntp = \
        self.estabilizador_horizontal.FMtp[0]

# Estabilizador vertical
self.estabilizador_vertical.calcula_FMfn()
Xfn, Yfn, Zfn, Lfn, Mfn, Nfn = \
```

```
        self.estabilizador_vertical.FMfn[0]


#############################################################
#   RESOLUCION ECUACIONES LONGITUDINALES                    #
#############################################################
# Despejamos las fuerzas vertical del rotor
ZR = M*(-q*u + p*v) - (Zf + Zfn + Ztp + ZT) - M*g*Cth*Cfi

# Despejamos el momento del rotor
MR = -(Izz - Ixx)*r*p - Ixz*(r**2 - p**2) - \
        (Mf + Mfn + Mtp + MT)
# Trasladmos las fuerzas y momentos del rotor en ejes cuerpo
# en el centro de masas a fuerzas y momentos en ejes viento
# en el rotor
XRh_b = XR
ZRh_b = ZR
MRh_b = MR - ZR*self.rotor.xcg + XR*self.rotor.hR

# FIX: Si tenemos angulo de resbalamiento habria que
# investigar como de necesario es realizar un buen calculo
# de YRh_b, LRh_b y NRh_b antes de despejar valores en ejes
# viento
ZRh_w = dot(LWB[2], [XRh_b, YRh_b, ZRh_b])
MRh_w = dot(LWB[1], [LRh_b, MRh_b, NRh_b])

# Dimension de fuerzas para el rotor
dimFR = ro_t*(Om*self.rotor.R)**2*pi*self.rotor.R**2

# Calculamos cT
cT = -ZRh_w/dimFR

# Coeficiente de rozamiento de los perfiles
de = self.rotor.de0 + self.rotor.de2*cT**2

# Calculamos el batimiento be1c_w usando la ecuacion de
# momento de cabeceo
# FIX: Como de bueno necesitamos a Qrh_w?
be1c_w = -(MRh_w - QRh_w/2.*be1s_w)*2/(
        self.rotor.Nb*self.rotor.Kbe)

# Estimamos un nuevo valor para XR, aprovechando el nuevo
# valor de be1c_w
# El problema es que necesitamos los valores de los
# controles para poder realizar una buena estimación
be0, la1s_w, la1c_w, th0, th1s_w, th1c_w = f_controles()

# Utilizamos funciones para calcular los terminos de
# F1 y F2 porque mas adelante los calcularemos de nuevo

#Angulo de ataque efectivo en el perfil de la pala
al1s_w = p_w - la1s_w + be1c_w + th1s_w
al1c_w = q_w - la1c_w - be1s_w + th1c_w

# Primeros armonicos F1
def f_F11s():
    return al1s_w/3. + nu*( th0 + nuz - la0 + 2./3.*tht )

def f_F11c():
    return al1c_w/3. - 0.5*nu*be0

# Segundos armonicos F1
```

```
def f_F12s():
    return  0.5*nu*( 0.5*al1c_w + 0.5*( th1c_w - be1s_w )
            - nu*be0 )
def f_F12c():
    return -0.5*nu*( 0.5*al1s_w + 0.5*( th1s_w + be1c_w )
            + nu*( th0 + 0.5*tht ) )

# Primeros armonicos F2
def f_F21s():
    return 0.5*(nu**2)*be0*be1s_w  + \
            ( nuz - la0 - 0.25*nu*be1c_w ) *\
               ( al1s_w - th1s_w ) -\
               0.25*nu*be1s_w*( al1c_w - th1c_w ) +\
           th0*(
             (al1s_w - th1s_w)/3. +
              nu*( nuz - la0 ) -
              0.25*(nu**2)*be1c_w
           ) +\
           tht*(
                 (al1s_w - th1s_w)/4. +
              0.5*nu*(nuz - la0 - 0.25*nu*be1c_w)
           )+ \
           th1s_w*(
                0.5*( nuz - la0 ) +
                nu*( 3./8.*( p_w - la1s_w ) +
                0.25*be1c_w)
              ) +\
           th1c_w*0.25*nu*(
                    0.5*(q_w - la1c_w) -
                be1s_w -
                nu*be0
                  )-\
           de*nu/a0
def f_F21c():
    return ( al1c_w - th1c_w - 2*be0*nu )*(
             nuz - la0 - 0.75*nu*be1c_w )-\
           0.25*nu*be1s_w*( al1s_w - th1s_w ) + \
           th0*(
             (al1c_w - th1c_w)/3. -
              0.5*nu*( be0 + 0.5*nu*be1s_w )
           ) +\
           tht*(
                 0.25*(al1c_w - th1c_w) -
             nu*(be0/3. + 0.125*nu*be1s_w)
           ) +\
           th1c_w*(
                    0.5*( nuz - la0 ) +
            0.25*nu*( 0.5*( p_w - la1s_w ) -
            be1c_w )
              ) +\
           th1s_w*0.25*nu*(
                    0.5*( q_w - la1c_w ) -
                be1s_w -
                nu*be0
                  )
# _Cxw = 2*Cxw/(a0*s)
# _Cyw = 2*Cyw/(a0*s)
# Comunes _Cxw y _Cyw
F12c = f_F12c()
F12s = f_F12s()

_Cxw = ( cT/(a0*s) + F12c/4. )*be1c_w + f_F11c()/2.*be0 +\
```

```
                F12s/4.*be1s_w + f_F21s()/2.
        _Cyw = (-cT/(a0*s) + F12c/4. )*be1s_w - f_F11s()/2.*be0 -\
                F12s/4.*be1c_w + f_F21c()/2.

        XRh_w = dimFR*a0*s/2.*_Cxw
        YRh_w = dimFR*a0*s/2.*_Cyw
        XR = XRh_b = dot(LBW[0], [XRh_w, YRh_w, ZRh_w])

        # Calculamos el angulo theta a partir de la ecuacion
        # horizontal y vertical
        # Fuerzas de la gravedad
        XG = M*(-r*v + q*w) - (Xf + Xfn + Xtp + XT + XR)
        ZG = M*(-q*u + p*v) - (Zf + Zfn + Ztp + ZT + ZR)

        # Iteramos th
        if metodo == 'Newton':
            f_th = atan(-Cfi*XG/ZG)
            epsNew = 0.001
            if st==0:
                f_th0 = f_th
                th = th - epsNew
                st += 1
            elif st==1:
                f_th1 = f_th
                th = th + epsNew
                st += 1
            elif st==2:
                f_th2 = f_th
                th_0 = th
                th = th_0 - k_th*(th_0 - f_th0)/(
                        1 - (f_th2 - f_th1)/(2*epsNew))
                st = 0
                err_th = abs(th - th_0)
                n_th += 1
        elif metodo == 'Fijo':
            th_0 = th
            th = th_0 + k_th*(atan(-Cfi*XG/ZG) - th_0)
            err_th = abs(th - th_0)
            n_th += 1
        else:
            sys.exit("Metodo de iteracion desconocido: %s" % metodo)

        print ( ( "\t\tth = %f, be1c_w = %f, cT = %f, altp0 = %f, "+
                "err_th = %f") %
                 (th, be1c_w, cT,
                     self.estabilizador_horizontal.altp0, err_th))

    # FIN BUCLE TH #
    Cth, Sth = cos(th), sin(th)

    ######################################################################
    #   VELOCIDAD INDUCIDA                                               #
    ######################################################################
    # Calculamos la velocidad inducida en el rotor
    # ki, knu = 1.0, 1.0
    def f(la0):
        def VT(la0):
            return (nu/knu)**2 + (1./knu**2 - 1./ki**2)*nuz**2  +\
                    ((nuz - la0)/ki)**2
        # return la0/ki - cT/(2*sqrt(VT(la0)))
        return 2*sqrt(VT(la0))*la0/ki - cT
```

```
la0NG = solveSecante(f, 0.1, 0.5, 100, 1e-12)

z = z_t + self.rotor.h0
if z>self.rotor.R/10.:
    K0G = 1 - 1./(16.*(z/self.rotor.R)**2)/(1 + (nu/la0NG)**2)
else:
    K0G = 1.0
# Velocidad con efecto suelo
la0 = K0G*la0NG + nuz*(1-K0G)

#####################################################################
#   TRACCION DE COLA                                                #
#####################################################################
# Calculamos el momento del rotor
NRh_w = QRh_w = self.rotor.R*(-(nuz - la0)*(-ZRh_w) +\
        nu*XRh_w + dimFR*de*s/8.*(1. + 3.*nu**2))
NRh_b = dot(LBW[2], [LRh_w, MRh_w, NRh_w])

# FIX: de nuevo hay que investigar la influencia de YRh_b,
# podria dar valores de error altos para helicoptero como el
# Lynx o el BlackHawk con xcg grande
QR = NR = NRh_b - YRh_b*self.rotor.xcg

# Despejamos la traccion de cola a partir de la ecuación de
# momentos de guiñada
# FIX: que influencia tiene QT en helicopteros con K!=0 como el
# Black Hawk?
NT = -(Ixx - Iyy)*p*q + Ixz*q*r - (Nf + Nfn + Ntp + NR)
TT =  (-NT + SK*QT)/((self.rotor_cola.lT +\
        self.rotor.xcg)*(CK - be1sT*SK))

# Fuerzas y momentos en ejes cola
XT_T =  TT*be1cT
YT_T = -TT*be1sT
ZT_T = -TT
LT_T =  0.0
MT_T =  0.0
NT_T =  QT

# Fuerzas y momentos en ejes cuerpo
XT_b, YT_b, ZT_b = dot(self.rotor_cola.LBT, [XT_T, YT_T, ZT_T])
LT_b, MT_b, NT_b = dot(self.rotor_cola.LBT, [LT_T, MT_T, NT_T])

# Fuerzas y momentos en ejes cuerpo respecto al centro de masas
l = self.rotor_cola.lT + self.rotor.xcg
XT, YT, ZT = XT_b, YT_b, ZT_b
LT = LT_b + YT_b*self.rotor_cola.hT
MT = MT_b - XT_b*self.rotor_cola.hT + ZT_b*l
NT = NT_b - YT_b*l

# Despejamos el batimiento lateral a partir del momento de
# balance
LR = (Ixx - Iyy)*q*r - Ixz*p*q - (Lf + Lfn + Ltp + LT)
LRh_b = LR - YR*self.rotor.hR
LRh_w = dot(LWB[0], [LRh_b, MRh_b, NRh_b])
be1s_w = -(LRh_w + QRh_w/2.*be1c_w)*2/self.rotor.Nb\
        /self.rotor.Kbe

# Recalculamos la fuerza lateral del rotor con la mejor
# aproximacion de be1s_w
# _Cyw = 2*Cyw/(a0*s)
_Cyw = (-cT/(a0*s) + f_F12c()/4. )*be1s_w - f_F11s()/2.*be0 -\
```

```
                f_F12s()/4.*be1c_w + f_F21c()/2.
        YRh_w = dimFR*a0*s/2.*_Cyw
        YR = YRh_b = dot(LBW[1], [XRh_w, YRh_w, ZRh_w])

        # Despejamos fi de la ecuacion de fuerza lateral
        if Cth == 0:
            sys.exit("Error: th = pi/2")
        Sfi = 1/(M*g*Cth)*( M*(u*r - w*p) - (Yf + Yfn + Ytp + YT + YR) )
        if abs(Sfi) > 1:
            sys.exit("Error: sin(fi)>1")
        Cfi = 1 - Sfi**2
        if Cfi<0:
            Cfi = 0.0
        fi_0 = fi
        fi = fi_0 + k_fi*(asin(Sfi) - fi_0)
        err_fi = abs(fi - fi_0)
        n_fi += 1
        # print "YR = %f, YT = %f, LR = %f, LT = %f" % (YR, YT, LR, LT)
        print "\tfi= %f, la0 = %f, be1s_w = %f, err_fi = %f" \
                % (fi, la0, be1s_w, err_fi)

        # FIN BUCLE FI #

    ##########################################################################
    #   VELOCIDAD INDUCIDA DE COLA                                           #
    ##########################################################################
    # Velocidad de giro del rotor de cola
    OmT = self.rotor_cola.gT*Om

    # Velocidad de la cola, relativa al viento en ejes cuerpo
    VTrw_b = self.Vrw_b +\
            cross(  W_b,
                    [-(self.rotor.xcg + self.rotor_cola.lT),
                        0.0, -self.rotor_cola.hT])

    # Velocidad de la cola, relativa al viento en ejes cola
    uTrw_T, vTrw_T, wTrw_T = dot(self.rotor_cola.LTB, VTrw_b)

    # Velocidad de la cola, relativa al viento en ejes viento
    uTrw_w = sqrt( uTrw_T**2 + vTrw_T**2 )
    wTrw_w = wTrw_T

    # Angulo ejes viento-ejes cola
    if uTrw_w!=0:
        CchwT = uTrw_T/uTrw_w
        SchwT = vTrw_T/uTrw_w
    else:
        if vTrw_T==0:
            CchwT = 1.0
            SchwT = 0.0
        elif vTrw_T<0:
            CchwT =  0.0
            SchwT = -1.0
        else:
            CchwT = 0.0
            SchwT = 1.0

    # Parametros de avance
    nuT =  uTrw_w/( OmT*self.rotor_cola.RT )
    nuzT = wTrw_w/( OmT*self.rotor_cola.RT )

    # Coeficiente de traccion de la cola
```

```
dimFT = ro_t*(OmT*self.rotor_cola.RT)**2*pi*self.rotor_cola.RT**2
FT = 1. - 3./4*self.estabilizador_vertical.Sfn/(
        pi*self.rotor_cola.RT**2 )
cTT = TT/(dimFT*FT)

# Resistencia
deT = self.rotor_cola.de0T + self.rotor_cola.de2T*cTT**2

def fT(la0T):
    def VTT(la0):
        return (nuT/knu)**2 + (1./knu**2 - 1./ki**2)*nuzT**2  +\
                ((nuzT - la0T)/ki)**2
    return 2*sqrt(VTT(la0T))*la0T/ki - cTT

la0T = solveSecante(fT, 0.1, 0.5, 100, 1e-12)

#########################################################################
#   ECUACION DEL MOTOR                                                  #
#########################################################################
# Par de la cola
QT = self.rotor_cola.RT*( -(nuzT - la0T)*TT/FT +\
        dimFT*deT*self.rotor_cola.sT/8.*( 1. + 3*nuT**2) )

# Par del motor
Q1 = (1. + self.motor.P)*\
        (QR + self.rotor_cola.gT*QT)/self.motor.nmotores
# Revoluciones del rotor
self.motor.calcula_K3()
Om_0 = Om
self.Om = Om = Om_0 +\
        k_Om*(self.motor.Omi - Q1/self.motor.K3 - Om_0)
err_Om = abs(Om - Om_0)
n_Om += 1

#########################################################################
#   CONTROLES                                                           #
#########################################################################
# Resolvemos simulataneamente los controles
# th0, th1c_w, th1s_w, th0T y los batimientos
# t velocidades inducidas que todavia no se han calculado:
# be0, be1cT_w, be1sT_w, la1cw_w,
# la1s_w. Tambien calculamos las  fuerzas y momentos que faltan del
# rotor de cola

be0, la1s_w, la1c_w, th0, th1s_w, th1c_w = f_controles()

th1c =  Cchw*th1c_w + Schw*th1s_w
th1s = -Schw*th1c_w + Cchw*th1s_w

be1c =  Cchw*be1c_w + Schw*be1s_w
be1s = -Schw*be1c_w + Cchw*be1s_w

#
# Batimiento y controles del rotor de cola
#

# Numero de Locke
gaT = ro_t*self.rotor_cola.cT*self.rotor_cola.a0T*\
        self.rotor_cola.RT**4/self.rotor_cola.IbeT
# Frecuencia natural de batimiento al cuadrado
labeT2 = 1. + self.rotor_cola.KbeT/( self.rotor_cola.IbeT*OmT**2 )
```

```
# Lado izquierdo de la ecuacion de batimiento
A_bebeT = array(shape=(3, 3), type='Float64')

A_bebeT[0, 0] = (8.*labeT2/gaT) - self.rotor_cola.k3*(1. + nuT**2)
A_bebeT[0, 1] =  0.
A_bebeT[0, 2] = -self.rotor_cola.k3*4./3.*nuT

A_bebeT[1, 0] =  4./3.*nuT
A_bebeT[1, 1] =  8.*(labeT2 - 1.)/gaT - \
        self.rotor_cola.k3*(1. + 0.5*nuT**2)
A_bebeT[1, 2] =  1 + 0.5*nuT**2

A_bebeT[2, 0] = -self.rotor_cola.k3*8./3.*nuT
A_bebeT[2, 1] = -1. + 0.5*nuT**2
A_bebeT[2, 2] =  8.*(labeT2 - 1.)/gaT -\
        self.rotor_cola.k3*(1. + 1.5*nuT**2)

# Matriz de influencia del colectivo de cola sobre el batimiento
A_beth0T = array([
    1. + nuT**2,
    0.,
    8./3.*nuT ], type='Float64')

# Matriz de influencia de la velocidad normal al rotor sobre el
# batimiento
A_bela0T = array([
    4./3.,
    0.,
    2*nuT ], type='Float64')

# Matriz de influencia de la torsion de cola sobre el batimiento
A_bethtT = array([
    4*(1./5. + nuT**2/6.),
    0.,
    2.*nuT ], type='Float64')


lhs = array(shape=(4,4), type='Float64')
lhs[0:3, 0:3] =  A_bebeT
lhs[0:3, 3  ] = -A_beth0T
lhs[3, 0] = self.rotor_cola.k3*(1./3. + nuT**2/2.)
lhs[3, 1] = 0.
lhs[3, 2] = nuT/2.*self.rotor_cola.k3
lhs[3, 3] = 1./3. + nuT**2/2.

rhs = array(shape=(4), type='Float64')
rhs[0:3] = (nuzT-la0T)*A_bela0T + self.rotor_cola.tht*A_bethtT
rhs[3] = 2*cTT/(self.rotor_cola.a0T*self.rotor_cola.sT) -\
        0.5*(nuzT - la0T)

be0T, be1cT_w, be1sT_w, th0T = la.solve_linear_equations(lhs, rhs)

# Batimientos en ejes cola
be1cT = be1cT_w*CchwT + be1sT_w*SchwT
be1sT = be1sT_w*CchwT - be1cT_w*SchwT

# Calculamos las fuerzas y momentos del rotor de cola
XT_T =  TT*be1cT
YT_T = -TT*be1sT
ZT_T = -TT
LT_T =  0.0
MT_T =  0.0
```

```
            NT_T =  QT

            # Fuerzas y momentos en ejes cuerpo
            XT_b, YT_b, ZT_b = dot(self.rotor_cola.LBT, [XT_T, YT_T, ZT_T])
            LT_b, MT_b, NT_b = dot(self.rotor_cola.LBT, [LT_T, MT_T, NT_T])

            # Fuerzas y momentos en ejes cuerpo respecto al centro de masas
            l = self.rotor_cola.lT + self.rotor.xcg
            XT, YT, ZT = XT_b, YT_b, ZT_b
            LT = LT_b + YT_b*self.rotor_cola.hT
            MT = MT_b - XT_b*self.rotor_cola.hT + ZT_b*l
            NT = NT_b - YT_b*l

            # Controles del piloto
            et0p, et1sp, et1cp, etp =\
                    f_etp(th0, th1s, th1c, th0T, p, q, r, th, fi, 0.0)

            print "be0 = %f, la1s_w = %f, la1c_w = %f" %\
                    (be0, la1s_w, la1c_w)
            print "be0T = %f, be1cT_w = %f, be1sT_w = %f" \
                    % (be0T, be1cT_w, be1sT_w)
            print "cTT = %f, la0T = %f, Om = %f, err_Om = %f" \
                    % (cTT, la0T, Om, err_Om)
            print ( "**********************************************************" +
                    "************************" )
            print "th0 = %f, th1s = %f, th1c = %f, th0T = %f"  \
                    % (th0, th1s, th1c, th0T)
            print ( "**********************************************************" +
                    "************************" )

        # Colocamos al helicopter en estado de trimado
        _Q = Quaternion.rotacion([
                (pi, 0, 1, 0), # Ejes inerciales a auxiliares
                (th, 0, 1, 0), # Angulos de euler, ch = 0
                (fi, 1, 0, 0)])

        self.u, self.v, self.w = u, v, w
        self.p, self.q, self.r = p, q, r
        self.q0, self.q1, self.q2, self.q3 = _Q.q0, _Q.q1, _Q.q2, _Q.q3
        self.Om, self.Q1, self.DTQ1 = Om, Q1, 0.0
        self.z_i = z_t

        self.controles.th0 =  th0
        self.controles.th1c = th1c
        self.controles.th1s = th1s
        self.controles.th0T = th0T

        # Varios valores intermedios calculados

        return {
                'th':       th,
                'fi':       fi,
                'Om':       Om,
                'be1c_w':   be1c_w,
                'be1s_w':   be1s_w,
                'be0':      be0,
                'la0':      la0,
                'la1c_w':   la1c_w,
                'la1s_w':   la1s_w,
                'be1c':     be1c,
                'be1s':     be1s,
                'cT':       cT,
```

```
            'cTT':      cTT,
            'be1cT':    be1cT,
            'be1sT':    be1sT,
            'be1cT_w':  be1cT_w,
            'be1sT_w':  be1sT_w,
            'la0T':     la0T,
            'u':        u,
            'v':        v,
            'w':        w,
            'p':        p,
            'q':        q,
            'r':        r,
            'th0':      th0,
            'th1c':     th1c,
            'th1s':     th1s,
            'th0T':     th0T,
            'Xf':       Xf,
            'Yf':       Yf,
            'Zf':       Zf,
            'Lf':       Lf,
            'Mf':       Mf,
            'Nf':       Nf,
            'Xtp':      Xtp,
            'Ytp':      Ytp,
            'Ztp':      Ztp,
            'Ltp':      Ltp,
            'Mtp':      Mtp,
            'Ntp':      Ntp,
            'Xfn':      Xfn,
            'Yfn':      Yfn,
            'Zfn':      Zfn,
            'Lfn':      Lfn,
            'Mfn':      Mfn,
            'Nfn':      Nfn,
            'XT':       XT,
            'YT':       YT,
            'ZT':       ZT,
            'LT':       LT,
            'MT':       MT,
            'NT':       NT,
            'XR':       XR,
            'YR':       YR,
            'ZR':       ZR,
            'LR':       LR,
            'MR':       MR,
            'NR':       NR,
            'q0':       _Q.q0,
            'q1':       _Q.q1,
            'q2':       _Q.q2,
            'q3':       _Q.q3,
            'DTQ1':     0.0,
            'Q1':       Q1,
            'QT':       QT,
            'nu':       nu,
            'nuz':      nuz,
            'altp0':    self.estabilizador_horizontal.altp0,
            'et0p':     et0p,
            'et1sp':    et1sp,
            'et1cp':    et1cp,
            'etpp':     etp,
            'altp':     self.estabilizador_horizontal.al,
            'cLtp':     self.estabilizador_horizontal.cL,
```

```
                'cDtp':     self.estabilizador_horizontal.cD,
                'befn':     self.estabilizador_vertical.be,
                'cLfn':     self.estabilizador_vertical.cL,
                'cDfn':     self.estabilizador_vertical.cD,
                'alf':      self.fuselaje.al,
                'cLf':      self.fuselaje.cL,
                'cDf':      self.fuselaje.cD,
                'cYf':      self.fuselaje.cY,
                'clf':      self.fuselaje.cl,
                'cmf':      self.fuselaje.cm,
                'cnf':      self.fuselaje.cn,
            }
```

B.4. rotor.py

```
# -*- coding: latin-1 -*-
from numarray import *
import numarray.linear_algebra as la
from matematicas import *


class Rotor:
    def __init__(self, modelo):
        self.modelo = modelo
        self.params = {}

    def init(self):
        """
        Inicializa algunas variables que permanecen constantes y declara
        algunos parametros.
        """
        Cgas = cos(self.gas)
        Sgas = sin(self.gas)
        # Matriz de cambio de base ejes rotor-cuerpo
        self.LHB = array([
            [  Cgas, 0.0, Sgas ],
            [  0.0,  1.0, 0.0  ],
            [ -Sgas, 0.0, Cgas ]], type = 'Float64' )

        self.LBH = transpose(self.LHB)

        # Solidez del rotor
        self.s = self.Nb*self.c/(pi*self.R)

        # Coeficientes de la TCMM
        self.ki =  (9./5.)**0.25
        self.knu = (5./4.)**0.25

        # Parametros de la solución numérica de la velocidad inducida
        self.params['la0'] = {'ila0': -1.0, 'Nmax': 50, 'eps': 1e-6}


    def calcula_FMR(self):
        """
        Calcula las fuerzas y momentos del rotor

        NECESITA:
            modelo.Vrw_b
            controles.
```

```
            th0
            th1c
            th1s

    PROVEE:
        rotor.FMR
        rotor.la0
        rotor.xi
        rotor.QR
    """
    # Unos alias para mejorar la legibilidad
    a0, s = self.a0, self.s
    Om = self.modelo.Om

    # Si el rotor no gira nada de esto tiene sentido y ademas no se puede
    # adimensionalizar con Om
    if Om < 1:
        self.FMR = (
                zeros(shape=(6, ),    type='Float64'),
                zeros(shape=(6, 16), type='Float64') )
        self.la0 = 0.0
        self.xi = 0.0
        self.QR = (
                0.0,
                zeros(shape=(16, ), type='Float64') )
        return
    ##########################################################################
    #   CAMBIO A EJES VIENTO DEL ROTOR                                       #
    ##########################################################################

    # Velocidad angular en ejes cuerpo
    W_b = array([ self.modelo.p, self.modelo.q, self.modelo.r ],
            type='Float64')

    # Velocidad del rotor relativa al viento, en ejes cuerpo
    Vhrw_b = self.modelo.Vrw_b +\
            cross(W_b, [-self.xcg, 0.0, -self.hR])

    # en ejes rotor
    uhrw_h, vhrw_h, whrw_h = dot(self.LHB, Vhrw_b)

    # Velocidad angular en ejes rotor
    p_h, q_h, r_h = dot(self.LHB, W_b)

    # Velocidad relativa del rotor respecto al viento, en ejes viento
    uhrw_w = sqrt(uhrw_h**2 + vhrw_h**2)
    whrw_w = whrw_h

    # Angulo que forman los ejes viento con los ejes rotor.
    # Si no hay velocidad se supone 0.
    if uhrw_w!=0:
        Cchw = uhrw_h/uhrw_w
        Schw = vhrw_h/uhrw_w
    else:
        # Por convenio chw = 0 si u = v = 0
        if vhrw_h>0:
            Cchw = 0.0
            Schw = 1.0
        elif vhrw_h<0:
            Cchw =  0.0
            Schw = -1.0
        else:
```

```
            Cchw = 1.0
            Schw = 0.0

        # Matriz cambio de base de ejes viento a ejes cuerpo
        # nos es util para pasar las fuerzas y momentos a ejes
        # cuerpo
        LHW = array([
            [ Cchw, -Schw, 0.0 ],
            [ Schw,  Cchw, 0.0 ],
            [ 0.0,   0.0,  1.0 ]], type = 'Float64')
        LBW = dot(self.LBH, LHW)

        # Cambiamos a ejes viento los angulos de paso del rotor
        th0 = self.modelo.controles.th0
        tht = self.tht
        th1c_w = self.modelo.controles.th1c*Cchw -\
                self.modelo.controles.th1s*Schw
        th1s_w = self.modelo.controles.th1c*Schw +\
                self.modelo.controles.th1s*Cchw

        # Parametros de avance
        nu  = uhrw_w/( Om*self.R )
        nuz = whrw_w/( Om*self.R )

        # Velocidad angular adimensionalizada en ejes viento
        p_w = (  Cchw*p_h + Schw*q_h )/Om
        q_w = ( -Schw*p_h + Cchw*q_h )/Om

        # Las derivadas respecto al tiempo del angulo chw y de las
        # velocidad son matrices, que al multiplicar por la derivada del
        # vector de estado devuelven el valor que corresponda.

        # Derivada respecto al tiempo del angulo chw, en funcion del
        # vector de estado
        m = uhrw_h**2 + vhrw_h**2
        # Por convenio DTchw = 0 si u=v=0
        DTchw = zeros( shape = (16,), type = 'Float64' )
        if m!=0:
            DTchw[:2] = 1./m*dot( [-vhrw_h, uhrw_h ], self.LHB[:2, :2] )
        # Derivada respecto al tiempo de W_h, en funcion del vector de
        # estado
        DTW_h = zeros(shape = (3, 16), type = 'Float64')
        DTW_h[:, 3:6] = self.LHB

        # Derivada respecto al azimuth de p_w y q_w (adimensionalizadas),
        # en funcion del  vector de estado
        DChip_w = 1./(Om**2)*( DTchw*( -Schw*p_h + Cchw*q_h ) +\
                Cchw*DTW_h[0] + Schw*DTW_h[1])
        DChiq_w = 1./(Om**2)*( DTchw*( -Cchw*p_h - Schw*q_h ) -\
                Schw*DTW_h[0] + Cchw*DTW_h[1])

        ########################################################################
        #   CALCULO DE LA VELOCIDAD INDUCIDA UNIFORME                          #
        ########################################################################
        # Coeficientes de la TCMM
        ki, knu = self.ki, self.knu
        # z = altura del rotor sobre el suelo
        # En el instante inicial con helicoptero tocando suelo z_i = 0
        # h0 = distancia rotor a punto mas bajo del helicoptero
        z = self.modelo.alt - self.modelo.hG + self.h0

        # Resolvemos h(la0) = 0 mediante el metodo de la secante
```

```
        def h(la0):
            # Efecto suelo
            def K0G(la0):
                # Si no el helicoptero no se estrella nunca!!!
                if z>self.R/3. and la0!=0.0:
                    return 1. - 1./(16.*(z/self.R)**2)/(1 + (nu/la0)**2)
                else:
                    return 1.0
            # Coeficiente de sustentacion del rotor
            def cT(la0):
                return a0*s/2.*(th0*(1./3. + nu**2/2.) +\
                        nu/2.*(th1s_w + p_w/2.) + K0G(la0)*(nuz - la0)/2. +\
                        1./4.*(1. + nu**2)*tht)
            # Velocidad en el rotor al cuadrado
            def VT2(la0):
                return (nu/knu)**2 + (1./knu**2 - 1./ki**2)*nuz**2  +\
                        ((nuz - la0)/ki)**2
            # Ecuacion que combina la TEP con la TCMM
            # return la0/ki - cT(la0)/(2*sqrt(VT2(la0)))
            return 2*sqrt(VT2(la0))*la0/ki - cT(la0)
        # la0NG no incluye efecto suelo
        la0NG = solveSecante(h, -0.2, 0.2, 100, 1e-12)
        # Si la iteracion no ha convergido asignamos un valor razonable
        # para que no se note :)
        if la0NG == None:
            la0NG = 0.06

        ##########################################################################
        #   CALCULO DEL BATIMIENTO                                               #
        ##########################################################################
        # VT = Velocidad del aire en el rotor
        VT2 = (nu/knu)**2 + (1/knu**2 - 1/ki**2)*(nuz**2) +\
                ((nuz - la0NG)/ki)**2
        VT = sqrt(VT2)
        # Velocidad del aire en el rotor corregida para terminos lineales
        barV = VT*(1 - la0NG*(nuz - la0NG)/ki**2/VT2)
        # Efecto suelo para la componente no lineal de la velocidad inducida
        if z>self.R/10:
            K0G = 1 - 1./(16.*(z/self.R)**2)/(1 + (nu/la0NG)**2)
        else:
            K0G = 1.0
        # Velocidad con efecto suelo
        la0 = K0G*la0NG + nuz*(1-K0G)
        # Angulo de la estela
        xi = ang(abs(la0*Om*self.R - whrw_w), uhrw_w)
        # Numero de Locke, varia con la densidad
        ga = self.modelo.ro*self.c*a0*self.R**4/self.Ibe
        # Frecuencia natural de batimiento, al cuadrado
        # Varia con la velocidad de rotacion del rotor
        labe2 = 1. + self.Kbe/( self.Ibe*Om**2)

        # Matrices de las 3 ecuaciones de batimiento
        # Lado izquierdo
        A_bebe = array([
            [ 8*labe2/ga,  0,              0                ],
            [ 4./3.*nu,    8*(labe2-1)/ga, 1 + 0.5*nu**2    ],
            [ 0,          -1. + 0.5*nu**2, 8*(labe2 - 1)/ga ]], type='Float64')

        A_bela = array([
            [ 2./3.*nu, 0  ],
            [ 0.,       1. ],
            [ 1.,       0. ]], type='Float64')
```

```
# Lado derecho
# Influencia del paso sobre el batimiento
A_beth = array([
    [ 1. + nu**2, 4./5. + 2./3.*nu**2, 4./3.*nu,       0.             ],
    [ 0.,         0.,                  0.,             1. + 0.5*nu**2 ],
    [ 8./3.*nu,   2*nu,                1. + 1.5*nu**2, 0              ]
    ], type='Float64')

# Influencia de la aceleracion angular sobre el batimiento
A_beDChiom = 8./ga*array([
    [0., 0.],
    [0., 1.],
    [1., 0.]], type='Float64')

# Influencia de la velocidad angular sobre el batimiento
A_beom = array([
    [ 2./3.*nu, 0      ],
    [ 16./ga,   1.     ],
    [ 1.,      -16./ga ]], type='Float64')

# Influencia de la velocidad inducida media sobre el batimiento
A_bela0 = array([
    4./3.,
    0.,
    2*nu], type='Float64')

# Matrices de las 2 ecuaciones de velocidad inducida
# Matriz L de Pitt-Peters
X = tan(abs(xi/2.))
K = array([
    [ 0,                  2*(1. + X**2)/barV, 0.                  ],
    [ 15.*pi/(64.*VT)*X, 0.,                  2.*(1. - X**2)/barV ]],
    type='Float64')

# Matriz de fuerzas y momentos aerodinamicos
# N = { F0/2, M1s, M1c}
Nth = array([
    [ 2./3. + nu**2, 0.5*(1. + nu**2), nu,                    0. ],
    [ 2./3.*nu,      0.5*nu,           0.25*(1. + 1.5*nu**2), 0  ],
    [ 0.,            0., 0.,           0.25*(1. + 0.5*nu**2)     ]],
    type='Float64')

Nbe = array([
    [  0.,    0.,                     0.                     ],
    [  0.,    0.25*(1. - 0.5*nu**2),  0.                     ],
    [ -nu/3., 0,                     -0.25*(1. + 0.5*nu**2) ]],
    type='Float64')

Nla = -0.5*array([
    [ nu,  0. ],
    [ 0.5, 0. ],
    [ 0.,  0.5]], type='Float64')

Nom = -Nla

Nla0 = array([
    1.,
    0.5*nu,
    0. ], type='Float64')

# Lado izquierdo de las ecuaciones de la velocidad inducida
```

```
        m = a0*s/4.
        A_labe = -m*dot(K, Nbe)
        A_lala = identity(2) - m*dot(K, Nla)
        # Matriz de influencia de controles, velocidad angular y velocidad
        # inducida uniforme sobre la velocidad inducida
        A_lath, A_laom, A_lala0 = [m*dot(K, x) for x in [Nth, Nom, Nla0]]
        # Matriz de influencia de la aceleracion angular sobre la velocidad
        # inducida
        A_laDChiom = zeros(shape=(2,2), type='Float64')

        # Ensamblamos el sistema
        # Lado izquierdo de las ecuaciones
        lhs = array(shape=(5, 5), type='Float64')
        lhs[0:3, 0:3], lhs[0:3, 3:5] = A_bebe, A_bela
        lhs[3:5, 0:3], lhs[3:5, 3:5] = A_labe, A_lala

        # Lado derecho de las ecuaciones
        # Terminos en funcion del vector de estado
        rhs_0 = dot(concatenate([A_beth, A_lath]), [th0, tht, th1s_w, th1c_w]) \
                + dot(concatenate([A_beom, A_laom]), [p_w, q_w]) \
                + (nuz - la0)*concatenate([A_bela0, A_lala0])

        # Terminos dependientes linealmente de la derivada del vector de
        # estado
        rhs_1 = dot(concatenate([A_beDChiom, A_laDChiom]),
                array([DChip_w, DChiq_w]))

        # Y resolvemos el batimiento
        lhs_inv = la.inverse(lhs)
        # {be0, be1c_w, be1s_w, la1s_w, la1c_w} = bat[0] + bat[1]DTx
        bat = [ dot(lhs_inv, rhs_0), dot(lhs_inv, rhs_1) ]

        ##########################################################################
        #   CALCULO DE LAS FUERZAS DEL ROTOR                                     #
        ##########################################################################
        # Alias
        # Para los resultados del batimiento
        be0, be1c_w, be1s_w, la1s_w, la1c_w = bat[0]
        # Angulo de ataque efectivo en el perfil de la pala
        al1s_w = p_w - la1s_w + be1c_w + th1s_w
        al1c_w = q_w - la1c_w - be1s_w + th1c_w

        # F1 = Fuerza aerodinámica normal al rotor
        # F1 = int(UT^2*th + UP*UT, r, 0, 1)
        # F1 = F10 + F11c*cos(ch) + F11s*sin(ch) + ...
        F10 = th0*( 1./3. + 0.5*nu**2 ) + 0.5*nu*( th1s_w + p_w/2. ) +\
                0.5*( nuz - la0 ) + 0.25*( 1. + nu**2)*tht
        # Coeficiente de resistencia medio
        de = self.de0 + self.de2*( 1./2.*a0*s*F10 )**2
        # Coeficiente de sustentacion
        cT = F10/2*a0*s
        # Primeros armonicos
        F11s = al1s_w/3. + nu*( th0 + nuz - la0 + 2./3.*tht )
        F11c = al1c_w/3. - 0.5*nu*be0
        # Segundos armonicos
        F12s =  0.5*nu*( 0.5*al1c_w + 0.5*( th1c_w - be1s_w ) - nu*be0 )
        F12c = -0.5*nu*( 0.5*al1s_w + 0.5*( th1s_w + be1c_w ) +\
                nu*( th0 + 0.5*tht ) )

        # F2 = Componente tangencial de la fuerza aerodinamica
        # F2 = int(UP*UT*th + UP^2 - de*UT^2/a0, r, 0, 1)
        # F2 = F20 + F21c*cos(ch) + F21s*sin(ch)
```

```
        # Solo necesitamos los primeros armonicos
        F21s = 0.5*(nu**2)*be0*be1s_w  + ( nuz - la0 - 0.25*nu*be1c_w ) *\
                   ( al1s_w - th1s_w ) - 0.25*nu*be1s_w*( al1c_w - th1c_w ) +\
               th0*(
                 (al1s_w - th1s_w)/3. +
                  nu*( nuz - la0 ) -
                  0.25*(nu**2)*be1c_w
               ) +\
               tht*(
                     (al1s_w - th1s_w)/4. +
                  0.5*nu*(nuz - la0 - 0.25*nu*be1c_w)
               )+ \
               th1s_w*(
                    0.5*( nuz - la0 ) +
                    nu*( 3./8.*( p_w - la1s_w ) +
                    0.25*be1c_w)
                  ) +\
               th1c_w*0.25*nu*(
                        0.5*(q_w - la1c_w) -
                    be1s_w -
                    nu*be0
                      )-\
               de*nu/a0

        F21c = ( al1c_w - th1c_w - 2*be0*nu )*( nuz - la0 - 0.75*nu*be1c_w )-\
               0.25*nu*be1s_w*( al1s_w - th1s_w ) + \
               th0*(
                 (al1c_w - th1c_w)/3. -
                  0.5*nu*( be0 + 0.5*nu*be1s_w )
               ) +\
               tht*(
                     0.25*(al1c_w - th1c_w) -
                 nu*(be0/3. + 0.125*nu*be1s_w)
               ) +\
               th1c_w*(
                        0.5*( nuz - la0 ) +
                0.25*nu*( 0.5*( p_w - la1s_w ) -
                be1c_w )
                  ) +\
               th1s_w*0.25*nu*(
                        0.5*( q_w - la1c_w ) -
                    be1s_w -
                    nu*be0
                      )

        # Coeficientes de fuerzas
        # _Cxw = 2*Cxw/(a0*s)
        # _Cyw = 2*Cyw/(a0*s)
        # _Czw = 2*Czw/(a0*s)
        _Cxw = ( F10/2. + F12c/4.)*be1c_w + F11c/2.*be0 + F12s/4.*be1s_w +\
                F21s/2.
        _Cyw = (-F10/2. + F12c/4.)*be1s_w - F11s/2.*be0 - F12s/4.*be1c_w +\
                F21c/2.
        _Czw = -F10
        # Dimensiones de las fuerzas * a0s/2
        dim_F = (a0*s)/2.*self.modelo.ro*((0m*self.R)**2)*pi*(self.R**2)
        # Componentes en ejes viento de las fuerzas sobre el rotor
        # Parte estacionaria
        F_w_0 = array([
                dim_F*_Cxw,
                dim_F*_Cyw,
                dim_F*_Czw ], type='Float64')
```

```
        # Para las fuerzas no hay terminos en aceleraciones angulares
        # Parte no estacionaria
        F_w_1 = zeros(shape=(3, 16), type='Float64')

        ##########################################################################
        #   CALCULO DE LOS MOMENTOS DEL ROTOR                                    #
        ##########################################################################
        # Momentos en ejes viento, lado derecho
        M_w_0 = array( shape = (3, ), type = 'Float64' )
        # Parece ser que esta formula esta mal en el Padfield, mas abajo la
        # formula coregida, segun viene en el  Bramwell
        # M_w_0[2] = self.R*( ( nuz - la0 )*F_w_0[2] + nu*F_w_0[0] + \
        # dim_F*de/( 4.*a0 )*( 1 + 7./3.*nu**2 ) )
        M_w_0[2] =  self.R*( ( nuz - la0 )*F_w_0[2] + nu*F_w_0[0] +\
                dim_F*de/( 4.*self.a0 )*( 1 + 3.*nu**2 ) )
        # Debido a la inclinacion del disco el par de motor contribuye un
        # 10% aproximadamente
        M_w_0[0] = -self.Nb/2.*self.Kbe*be1s_w - M_w_0[2]/2.*be1c_w
        M_w_0[1] = -self.Nb/2.*self.Kbe*be1c_w + M_w_0[2]/2.*be1s_w

        # Los momentos tienen terminos en DTx
        M_w_1 = array( shape = (3, 16), type = 'Float64' )
        M_w_1[0, :] = -self.Nb/2.*self.Kbe*bat[1][2]
        M_w_1[1, :] = -self.Nb/2.*self.Kbe*bat[1][1]
        M_w_1[2, :] =  zeros(shape=(16,), type='Float64')
        M_w_1[2, 10] = self.modelo.motor.Irt[self.modelo.motor.n] + self.Nb*self.Ibe

        ##########################################################################
        #   CALCULO DE FM EN EJES CUERPO                                         #
        ##########################################################################
        # Pasamos de ejes viento a ejes cuerpo
        # Estas son las fuerzas sobre la cabeza del rotor, pero ya en ejes
        # cuerpo
        F_b_0 = dot(LBW, F_w_0)
        F_b_1 = dot(LBW, F_w_1)
        M_b_0 = dot(LBW, M_w_0)
        M_b_1 = dot(LBW, M_w_1)
        # Utilizamos notacion matricial para el producto vectorial para
        # poder pasar los terminos que dependen de la derivada del
        # vector de estado (que son matrices)
        r_cross = array([
                [ 0.0,      self.hR,  0.0      ],
                [-self.hR,  0.0,      self.xcg ],
                [ 0.0,     -self.xcg, 0.0      ]], type='Float64')

        # Trasladamos al centro de masas
        F_0 = F_b_0
        F_1 = F_b_1
        M_0 = M_b_0 + dot(r_cross, F_b_0)
        M_1 = M_b_1 + dot(r_cross, F_b_1)

        # FM: Fuerzas y momentos conjuntos en un solo vector
        FM_0 = array(shape = (6,), type = 'Float64')
        FM_0[0:3] = F_0
        FM_0[3:6] = M_0

        FM_1 = array(shape = (6, 16), type = 'Float64')
        FM_1[0:3, :] = F_1
        FM_1[3:6, :] = M_1

        ##########################################################################
```

```
        #   HACEMOS NOTAR LOS CAMBIOS                                          #
        ###############################################################################

        #-------------------------------------------------------------------------#
        self.FMR = FM_0, FM_1
        #-------------------------------------------------------------------------#

        # Otras magnitudes usadas en otros subsistemas
        # Angulo de la estela, para el estabilizador horizontal, respecto al
        # eje z_b
        self.xi = xi + self.gas
        # La velocidad inducida, para el estabilizador horizontal y en
        # un futuro tambien podria ser para el fuselaje
        self.la0 = la0
        # El par del rotor, para el motor
        self.QR = M_w_0[2], M_w_1[2, :]

        # Logging de magnitudes intermedias
        self.Xr_w, self.Yr_w, self.Zr_w = F_w_0
        self.Lr_w, self.Mr_w, self.Nr_w = M_w_0
        self.Xr_b, self.Yr_b, self.Zr_b = F_b_0
        self.Lr_b, self.Mr_b, self.Nr_b = M_0
        self.Xr_i, self.Yr_i, self.Zr_i = dot(self.modelo.LIB, F_b_0)
        self.Lr_i, self.Mr_i, self.Nr_i = dot(self.modelo.LIB, M_b_0)

        self.nu, self.nuz = nu, nuz
        self.Cchw, self.Schw = Cchw, Schw
        self.cT, self.de = cT, de
        self.K0G, self.VT = K0G, VT
        self.be0, self.be1c_w, self.be1s_w = bat[0][:3]
        self.la1s_w, self.la1c_w = bat[0][3:]
        self.DTchw, self.DChip_w, self.DChiq_w = DChip_w, DChiq_w, DTchw
        self.th1c_w, self.th1s_w = th1c_w, th1s_w
        self.A_bebe, self.A_bela, self.A_labe, self.A_lala =\
                A_bebe, A_bela, A_labe, A_lala
        self.A_beth, self.A_lath, self.A_beom, self.A_laom =\
                A_beth, A_lath, A_beom, A_laom
        self.A_bela0, self.A_lala0, self.A_beDChiom, self.A_laDChiom =\
                A_bela0, A_lala0, A_beDChiom, A_laDChiom
        self.A = lhs
        self.A_th = concatenate([A_beth, A_lath])
        self.A_om = concatenate([A_beom, A_laom])
        self.A_la0 = concatenate([A_bela0, A_lala0])
        self.A_DChiom = concatenate([A_beDChiom, A_laDChiom])
        self.barV = barV
        self.labe2, self.ga = labe2, ga
        self.p_w, self.q_w = p_w, q_w
```

B.5. rotor_cola.py

```
from numarray import *
import numarray.linear_algebra as la
from matematicas import *

class RotorCola:
    def __init__(self, modelo):
        self.params = {}
        self.modelo = modelo
```

```
    def init(self):
        """
        Inicializa algunos parametros de la cola.
        """

        self.params['bat'] = {'ilaOT': -1.0, 'Nmax': 100, 'eps': 1e-4}
        SK, CK = sin(self.K), cos(self.K)
        # Matriz de cambio de base de ejes cuerpo a ejes cola
        self.LBT = array([
            [ 1.0, 0.0, 0.0 ],
            [ 0.0, SK, -CK  ],
            [ 0.0, CK,  SK  ]], type='Float64')
        self.LTB = transpose(self.LBT)
        self.sT = 4*self.cT/pi/self.RT

    def calcula_FMT(self):
        """
        Calcula las fuerzas y momentos del rotor de cola

        NECESITA:
            modelo.Vrw_b
            controles.th0T
        PROVEE:
            rotor_cola.FMT
            rotor_cola.QT

        """
        ###########################################################################
        #   CAMBIO A EJES VIENTO DEL ROTOR                                        #
        ###########################################################################

        # Velocidad de giro del rotor de cola
        OmT = self.gT*self.modelo.Om
        if OmT<1.0:
            self.FMT = (
                    zeros(shape=(6, ),   type='Float64'),
                    zeros(shape=(6, 16), type='Float64')
            )
            self.QT = 0.0
            return

        # Velocidad de la cola, relativa al viento en ejes cuerpo
        VTrw_b =  self.modelo.Vrw_b +\
                cross(
                    [   self.modelo.p, self.modelo.q,  self.modelo.r ],
                    [ -(self.modelo.rotor.xcg + self.lT), 0.0, -self.hT ]
                )

        # Velocidad de la cola, relativa al viento en ejes cola
        uTrw_T, vTrw_T, wTrw_T = dot(self.LTB, VTrw_b)
        # Velocidad de la cola, relativa al viento en ejes viento
        uTrw_w = sqrt( uTrw_T**2 + vTrw_T**2 )
        wTrw_w = wTrw_T
        # Angulo ejes viento-ejes cola
        if uTrw_w!=0:
            CchwT = uTrw_T/uTrw_w
            SchwT = vTrw_T/uTrw_w
        else:
            if vTrw_T==0:
                CchwT = 1.0
                SchwT = 0.0
            elif vTrw_T<0:
```

```
            CchwT =  0.0
            SchwT = -1.0
        else:
            CchwT = 0.0
            SchwT = 1.0
    # Parametros de avance
    nuT =  uTrw_w/( OmT*self.RT )
    nuzT = wTrw_w/( OmT*self.RT )


    ##########################################################################
    #   CALCULO DEL BATIMIENTO                                               #
    ##########################################################################

    # Numero de Locke
    gaT = self.modelo.ro*self.cT*self.a0T*self.RT**4/self.IbeT
    # Frecuencia natural de batimiento al cuadrado
    labeT2 = 1. + self.KbeT/( self.IbeT*(OmT**2) )

    # Lado izquierdo de la ecuacion de batimiento
    A_bebeT = array(shape=(3, 3), type='Float64')

    A_bebeT[0, 0] = (8.*labeT2/gaT) - self.k3*(1. + nuT**2)
    A_bebeT[0, 1] =  0.
    A_bebeT[0, 2] = -self.k3*4./3.*nuT

    A_bebeT[1, 0] =  4./3.*nuT
    A_bebeT[1, 1] =  8.*(labeT2 - 1.)/gaT - self.k3*(1. + 0.5*nuT**2)
    A_bebeT[1, 2] =  1 + 0.5*nuT**2

    A_bebeT[2, 0] = -self.k3*8./3.*nuT
    A_bebeT[2, 1] = -1. + 0.5*nuT**2
    A_bebeT[2, 2] =  8.*(labeT2 - 1.)/gaT - self.k3*(1. + 1.5*nuT**2)

    # Matriz de influencia del colectivo de cola sobre el batimiento
    A_beth0T = array([
        1. + nuT**2,
        0.,
        8./3.*nuT ], type='Float64')

    # Matriz de influencia de la velocidad normal al rotor sobre el
    # batimiento
    A_bela0T = array([
        4./3.,
        0.,
        2*nuT ], type='Float64')

    # Matriz de influencia de la torsion de cola sobre el batimiento
    A_bethtT = array([
        4*(1./5. + nuT**2/6.),
        0.,
        2.*nuT ], type='Float64')

    # Despejamos el batimiento en funcion de la velocidad inducida
    invA_bebeT = la.inverse(A_bebeT)
    beth0T = dot(invA_bebeT, A_beth0T)
    bela0T = dot(invA_bebeT, A_bela0T)
    bethtT = dot(invA_bebeT, A_bethtT)

    # Estas funciones tambien seran utiles mas adelante, por eso no
    # se declaran dentro de h
    def f_be0T(la0T):
```

```
        return beth0T[0]*self.modelo.controles.th0T +\
                bela0T[0]*(nuzT - la0T) +\
                bethtT[0]*self.modelo.rotor_cola.tht

    def f_be1swT(la0T):
        return beth0T[2]*self.modelo.controles.th0T +\
                bela0T[2]*(nuzT - la0T) +\
                bethtT[2]*self.modelo.rotor_cola.tht

    # No interviene en el calculo de la0T pero si en el de las fuerzas
    def f_be1cwT(la0T):
        return beth0T[1]*self.modelo.controles.th0T +\
                bela0T[1]*(nuzT - la0T) +\
                bethtT[1]*self.modelo.rotor_cola.tht

    # Angulos de paso efectivos
    def f_th0T(la0T):
        return self.modelo.controles.th0T + self.k3*f_be0T(la0T)

    def f_th1swT(la0T):
        return self.k3*f_be1swT(la0T)

    # Coeficiente de sustentacion de la cola
    def f_cTT(la0T):
        return self.a0T*self.sT/2.*(f_th0T(la0T)*(1./3. + 0.5*nuT**2) +\
                0.5*nuT*f_th1swT(la0T) +\
                0.5*(nuzT - la0T))

    # Constantes de la TCMM
    ki  = (9./5.)**0.25
    knu = (5./4.)**0.25
    # Cuadrado de la velocidad inducida en el rotor
    def f_VT2(la0T):
        return (nuT/knu)**2 + (1./knu**2 - 1./ki**2)*(nuzT**2) +\
                ((nuzT - la0T)/ki)**2

    def f_h(la0T):
        #return la0T/ki - f_cTT(la0T)/(2*sqrt(f_VT2(la0T)))
        return la0T/ki*2*sqrt(f_VT2(la0T)) - f_cTT(la0T)

    la0T = solveSecante(f_h, -0.2, 0.2, 100, 1e-12)
    # Calculamos otras magnitudes
    # Coeficiente de sustentacion
    cTT = f_cTT(la0T)
    # Coeficiente medio de resistencia
    deT = self.de0T + self.de2T*(cTT**2)
    # Coeficiente de par
    cQT = -(nuzT - la0T)*cTT + deT*self.sT/8.*(1. + 3*nuT**2)
    be0T = f_be0T(la0T)
    be1swT = f_be1swT(la0T)
    be1cwT = f_be1cwT(la0T)
    # Pasamos los angulos de batimiento a ejes rotor de cola
    be1cT = be1cwT*CchwT + be1swT*SchwT
    be1sT = be1swT*CchwT - be1cwT*SchwT

    ##########################################################################
    #   CALCULO DE LAS FUERZAS DEL ROTOR DE COLA EN EJES CUERPO              #
    ##########################################################################
    # Factor empirico de bloqueo del rotor cola debido al estabilizador
    # vertical
    FT = 1. - 3./4*self.modelo.estabilizador_vertical.Sfn/( pi*self.RT**2 )
    # Dimension de fuerza para el rotor de cola
```

```
        dim = self.modelo.ro*( OmT*self.RT )**2*( pi*self.RT**2 )
        # Fuerzas y momentos con dimesiones
        TT = dim*FT*cTT
        QT = dim*self.RT*cQT
        # Fuerzas y momentos en ejes cola
        XT_T =  TT*be1cT
        YT_T = -TT*be1sT
        ZT_T = -TT
        LT_T =  0.0
        MT_T =  0.0
        NT_T =  QT
        # Fuerzas y momentos en ejes cuerpo
        XT_b, YT_b, ZT_b = dot(self.LBT, [XT_T, YT_T, ZT_T])
        LT_b, MT_b, NT_b = dot(self.LBT, [LT_T, MT_T, NT_T])
        # Fuerzas y momentos en ejes cuerpo respecto al centro de masas
        l = self.lT + self.modelo.rotor.xcg
        XT = XT_b
        YT = YT_b
        ZT = ZT_b
        LT = LT_b + YT_b*self.hT
        MT = MT_b - XT_b*self.hT + ZT_b*l
        NT = NT_b - YT_b*l

        FMT_0 = array([XT, YT, ZT, LT, MT, NT], type='Float64')
        FMT_1 = zeros(shape = (6, 16), type = 'Float64')

        ###########################################################################
        #   HACEMOS NOTAR LOS CAMBIOS                                             #
        ###########################################################################

        #-------------------------------------------------------------------------#
        self.FMT = FMT_0, FMT_1
        #-------------------------------------------------------------------------#

        # Otros subsistemas necesitan esto
        # self.QT = -M_b[1]
        self.QT = QT

        # Logging
        self.deT = deT
        self.la0T = la0T
        self.cTT = cTT
        self.be1swT = be1swT
        self.be1cwT = be1cwT
        self.OmT = OmT
        self.XT_b, self.YT_b, self.ZT_b = XT, YT, ZT
        self.LT_b, self.MT_b, self.NT_b = LT, MT, NT
        self.nuT, self.nuzT = nuT, nuzT
```

B.6. fuselaje.py

```
from matematicas import *
from numarray import *

class Fuselaje:
    def __init__(self, modelo):
        self.modelo = modelo

    def init(self):
```

```
        pass

    def calcula_FMf(self):
        """
        Calcula las fuerzas aerodinamicas sobre el fuselaje, en ejes viento.

        NECESITA:
            modelo.Vrw_b

        PROVEE:
            fuselaje.FMf
        """

        # Un alias para le velocidad relativa al viento en ejes cuerpo
        urw_b, vrw_b, wrw_b = self.modelo.Vrw_b
        # Velocidad realativa al viento, al cuadrado, en ejes cuerpo
        Vf2rw_b = urw_b**2 + vrw_b**2 + wrw_b**2
        # Calculamos angulo de resbalamiento (positivo si el viento incide
        # por la derecha)
        be = ang(sqrt(urw_b**2 + wrw_b**2), vrw_b)
        # Calculamos angulo de ataque
        al = ang(urw_b, wrw_b)
        # Presion dinamica
        pd = 1./2.*self.modelo.ro*Vf2rw_b
        # Calculamos las fuerzas interpolando los coeficientes
        cD, cL, cY, cl, cm, cn = self.aero.coefs(al, be)

        Df_w = pd*self.Sp*cD
        Yf_w = pd*self.Ss*cY
        Lf_w = pd*self.Sp*cL
        lf_w = pd*self.Ss*self.lf*cl
        mf_w = pd*self.Sp*self.lf*cm
        nf_w = pd*self.Ss*self.lf*cn

        # Pasamos de ejes viento a ejes cuerpo
        Cal, Sal = cos(al), sin(al)
        Cbe, Sbe = cos(be), sin(be)
        # Matriz de cambio de ejes viento a ejes cuerpo
        Lbw = array( [
            [ Cal*Cbe, -Cal*Sbe, -Sal ],
            [ Sbe,      Cbe,      0.0 ],
            [ Sal*Cbe, -Sal*Sbe,  Cal ]
            ], type = 'Float64')
        # Fuerzas y momentos en el centro aerodinamico, en ejes cuerpo
        Xfca_b, Yfca_b, Zfca_b = dot(Lbw, [-Df_w, Yf_w, -Lf_w])
        Lfca_b, Mfca_b, Nfca_b = dot(Lbw, [lf_w, mf_w, nf_w])

        # Sobre el CM
        Xf, Yf, Zf = Xfca_b, Yfca_b, Zfca_b
        Mf = Mfca_b + Xf*self.zca - Zf*self.xca
        Lf = Lfca_b - Yf*self.zca
        Nf = Nfca_b + Yf*self.xca

        # Fuerzas y Momentos del fuselaje
        self.FMf = (
                array([Xf, Yf, Zf, Lf, Mf, Nf], type='Float64'),
                zeros(shape=(6, 16), type='Float64')
        )

        # logging
        self.be = be
        self.al = al
```

```
        self.cD, self.cL, self.cY = cD, cL, cY
        self.cl, self.cm, self.cn = cl, cm, cn
        self.Xf_b = Xf
        self.Yf_b = Yf
        self.Zf_b = Zf
        self.Lf_b = Lf
        self.Mf_b = Mf
        self.Nf_b = Nf
```

B.7. estabilizador.py

```
from matematicas import *
from numarray import *

class EstabilizadorHorizontal:
    def __init__(self, modelo):
        self.modelo = modelo

    def init(self):
        x = self.modelo.rotor.hR - self.modelo.rotor_cola.hT
        # angulos entre los cuales la estela afecta al estabilizador
        # xi1<x<xi2
        self.xi1 = ang(x, self.modelo.rotor.xcg + self.ltp -
                self.modelo.rotor.R)
        self.xi2 = ang(x, self.modelo.rotor.xcg + self.ltp +
                self.modelo.rotor.R)

    def calcula_FMtp(self):
        """
        Calcula las fuerzas y momentos del estabilizador horizontal.

        NECESITA:
            modelo.Vrw_b
            rotor.la0
            rotor.xi

        PROVEE:
            estabilizador_horizontal.FMtp
        """

        # Velocidad relativa al viento de la cola, en ejes cuerpo
        urwtp_b = self.modelo.Vrw_b[0] - self.modelo.q*self.htp
        vrwtp_b = ( self.modelo.Vrw_b[1]
                - self.modelo.r*(self.ltp + self.modelo.rotor.xcg)
                + self.modelo.p*self.htp  )
        wrwtp_b = self.modelo.Vrw_b[2] + self.modelo.q*\
                (self.ltp + self.modelo.rotor.xcg)

        # Angulo de ataque
        al = self.altp0 + ang(urwtp_b, wrwtp_b)
        Sal, Cal = sin(al), cos(al)
        # Angulo de resbalamiento
        be = ang(urwtp_b, vrwtp_b)
        # Coeficientes en ejes viento
        cL, cD = self.aero.coefs(al, be)
        # Coeficientes en ejes cuerpo
        cZ = -cD*Sal - cL*Cal
        cX =  cL*Sal - cD*Cal
        # Presion dinamica
```

```
        pd = 0.5*self.modelo.ro*(urwtp_b**2 + wrwtp_b**2)
        Ztp = pd*self.Stp*cZ
        Xtp = pd*self.Stp*cX

        self.FMtp = (
                array([Xtp, 0.0, Ztp, 0.0,
                    (self.ltp + self.modelo.rotor.xcg)*Ztp -
                    self.htp*Xtp, 0.0],  type='Float64'),
                zeros(shape=(6, 16), type='Float64')
        )

        # logging
        self.al = al
        self.be = be
        self.cL, self.cD = cL, cD
        ( self.Xtp_b, self.Ytp_b, self.Ztp_b,
                self.Ltp_b, self.Mtp_b, self.Ntp_b ) =\
                self.FMtp[0]

class EstabilizadorVertical:
    def __init__(self, modelo):
        self.modelo = modelo

    def calcula_FMfn(self):
        """
        Calcula las fuerzas y momentos del estabilizador vertical.

        NECESITA:
            modelo.Vrw_b
        PROVEE:
            estabilizador_vertical.FMfn
        """

        # Velocidad relativa al viento del estab vert en ejes cuerpo
        vrwfn_b = self.modelo.Vrw_b[1] - self.modelo.r*\
                (self.lfn + self.modelo.rotor.xcg) + self.hfn*self.modelo.p
        urwfn_b = self.modelo.Vrw_b[0] - self.modelo.q*self.hfn
        wrwfn_b = self.modelo.Vrw_b[2] + self.modelo.q*\
                (self.lfn + self.modelo.rotor.xcg)
        # Angulo de ataque del estabilizador
        be = self.befn0 + ang(urwfn_b, vrwfn_b)
        Sbe, Cbe = sin(be), cos(be)
        # Angulo de resbalamiento
        al = ang(urwfn_b, wrwfn_b)
        # Coeficientes en ejes viento
        cL, cD = self.aero.coefs(be, al)
        # Coeficientes en ejes cuerpo
        cY = -cD*Sbe - cL*Cbe
        cX =  cL*Sbe - cD*Cbe
        # Fuerzas
        pd = 0.5*self.modelo.ro*(urwfn_b**2 + vrwfn_b**2)
        Xfn = pd*self.Sfn*cX
        Yfn = pd*self.Sfn*cY
        Zfn = 0.0

        Lfn, Mfn, Nfn = cross(
                [-(self.lfn + self.modelo.rotor.xcg), 0.0, -self.hfn],
                [  Xfn,                               Yfn,  Zfn     ])

        #Nfn = -Yfn*(self.lfn + self.modelo.rotor.xcg)
        self.FMfn = (
                array([Xfn, Yfn, Zfn, Lfn, Mfn, Nfn], type='Float64'),
```

```
                zeros(shape=(6, 16), type='Float64')
        )

        # logging
        self.be = be
        self.al = al
        self.cL = cL
        self.cD = cD
        self.Xfn_b = Xfn
        self.Yfn_b = Yfn
        self.Zfn_b = Zfn
        self.Lfn_b = Lfn
        self.Mfn_b = Mfn
        self.Nfn_b = Nfn
```

B.8. controles.py

```
# -*- coding: latin-1 -*-
from numarray import *
from historial import *
from math import *

class Controles:
    def __init__(self, modelo):
        self.modelo = modelo
        # para actualizar todos los relojes al mismo tiempo, utilizamos
        # esta version absurda de reloj
        def reloj_tonto():
            return self.modelo.t

        # Colectivo
        self.et0p  = SerieTemporal(1000, reloj_tonto)
        # Ciclico lateral
        self.et1cp = SerieTemporal(1000, reloj_tonto)
        # Ciclico longitudinal
        self.et1sp = SerieTemporal(1000, reloj_tonto)
        # Colectivo de cola (pedal)
        self.etpp  = SerieTemporal(1000, reloj_tonto)

    def init(self):
        def func(x):
            if not hasattr(x, '__call__'):
                return lambda y: x
            else:
                return x
        self.th_0 = func(self.th_0)
        self.fi_0 = func(self.fi_0)
        self.ch_0 = func(self.ch_0)

    def calcula_controles(self):
        t = self.modelo.t
        # Controles
        et0p  = self.et0p(t)
        et1sp = self.et1sp(t)
        et1cp = self.et1cp(t)
        etpp  = self.etpp(t)

        et = dot(self.S0, [ et0p, et1sp, et1cp, etpp ]) +\
                dot(self.S1, [ self.modelo.p,
```

```
                                self.modelo.q,
                                self.modelo.r ]) +\
                    dot(self.S2, [ self.modelo.th - self.th_0(et1sp),
                                self.modelo.fi - self.fi_0(et1cp),
                                self.modelo.ch - self.ch_0(etpp)   ])

        # Transforma la posicion del control en un porcentaje
        # Estas formulas no son validas para el colectivo que
        # unicamente se mueve en el rango de 0 a 1
        def per(x):
            return (x + 1.)/2.

        # Transorma un porcentaje en posicion de un control
        def iper(x):
            return 2*x - 1

        p1sp = per(et1sp)
        p1cp = per(et1cp)
        ppp  = per(etpp)
        p1s  = per(et[1])
        p1c  = per(et[2])
        pp   = per(et[3])

        limite = self.L

        if p1s>p1sp + limite:
            et[1] = iper(p1sp + limite)
        elif p1s<p1sp - limite:
            et[1] = iper(p1sp - limite)

        if p1c>p1cp + limite:
            et[2] = iper(p1cp + limite)
        elif p1c<p1cp - limite:
            et[2] = iper(p1cp - limite)

        if pp>ppp + limite:
            et[3] = iper(ppp + limite)
        elif pp<ppp - limite:
            et[3] = iper(ppp - limite)

        # Pasos
        th = self.c0 + dot(self.c1, et)
        self.th0, self.th1s, self.th1c, self.th0T = th

        self.l_et0p  = et0p
        self.l_et1sp = et1sp
        self.l_et1cp = et1cp
        self.l_etpp  = etpp
```

B.9. motor.py

```
# -*- coding: latin-1 -*-
from numarray import *

class Motor:
    def __init__(self, modelo):
        self.modelo = modelo

    def init(self):
```

```
        self.start = True
        self.K3_bak = self.K3

    def calcula_K3(self):
        pass
        #if self.start:
        #    self.K3 = 1e4
        #else:
        #    self.K3 = self.K3_bak


    def calcula_Ir(self):
        """
        Inercia de giro, referida al rotor, del sistema transmision + rotor.
        """

        # Inercia del sistema de transmision (Irt) mas inercia de las palas
        # (Nb*Ibe)
        self.Ir = self.Irt[self.nmotores] +\
                self.modelo.rotor.Nb*self.modelo.rotor.Ibe

    def calcula_tae3(self):
        """
        Constante de tiempos.
        """
        Q =  self.Qef/self.Qmc
        if Q<=1:
            x1 = self.Qid/self.Qmc
            self.tae3 = self.a3 + (self.b3 - self.a3)/(1 - x1)*(Q - x1)
        else:
            x2 = self.Qto/self.Qmc
            self.tae3 = self.b3 + (self.c3 - self.b3)/(x2 - 1)*(Q - 1)

    def calcula_tae2(self):
        """
        Constante de tiempos.
        """
        Q =  self.Qef/self.Qmc
        if Q<=1:
            x1 = self.Qid/self.Qmc
            self.tae2 = self.a2 + (self.b2 - self.a2)/(1 - x1)*(Q - x1)
        else:
            x2 = self.Qto/self.Qmc
            self.tae2 = self.b2 + (self.c2 - self.b2)/(x2 - 1)*(Q - 1)

    def calcula_tae1(self):
        """
        Constante de tiempos.
        """
        pass

    def calcula_ecs_motor(self):
        # Corregimos velocidad angular
        if self.modelo.Om<0:
            self.modelo.integrador.x1[10] = self.modelo.Om = 0

        # Evidentemente para Om pequena las formulas siguientes no
        # son validas, pero tiramos adelante igual
        Om = self.modelo.Om
        if Om<1.0:
            Om = 1.0
        k = (self.modelo.ro/1.225)**(self.x)
```

```
        self.Wto = self.Wto_0*k
        self.Wmc = self.Wmc_0*k
        # Par maximo
        self.Qto = self.Wto/Om
        # Para maximo continuo
        self.Qmc = self.Wmc/Om
        # Par minimo
        self.Qid = self.Wid/Om

        # Corregimos par
        if self.modelo.Q1>self.Qto:
            self.modelo.integrador.x0[12] = self.modelo.Q1 = self.Qto

        # Par efectivo para calculo de coeficientes
        # Qid <= Qef <= Qto
        if self.modelo.Q1<self.Qid:
            self.Qef = self.Qid
        else:
            self.Qef = self.modelo.Q1

        # Calculamos constantes de tiempo
        self.calcula_tae1()
        self.calcula_tae2()
        self.calcula_tae3()

        if Om>0.8*self.Omi:
            self.start = False

        # Rigidez del motor
        self.calcula_K3()

        # Inercia total del sistema
        self.calcula_Ir()

        # Mucho ojo con esto, porque si se modifica x hay que modificar
        # tambien esto
        ecs_1 = zeros(shape = (3, 16), type = 'Float64')
        ecs_1[0, 5 ] = -1.0
        ecs_1[0, 10] =  1.0
        ecs_1[0] += 1./self.Ir*self.modelo.rotor.QR[1]
        ecs_1[1, 10:13] = [0.0, 0.0, 1.0]
        ecs_1[2, 10:13] = [self.K3*self.tae2, self.tae1*self.tae3,
                self.tae1 + self.tae3]

        # Para de rotor
        QR =   self.modelo.rotor.QR[0]
        # Para de cola
        QT =   self.modelo.rotor_cola.QT
        # Par total
        Qtot = self.Qid + (1. + self.P)*(QR + self.modelo.rotor_cola.gT*QT)
        # Diferencia entre par de motor y total
        Qdif = 1./self.Ir*(self.nmotores*self.modelo.Q1 - Qtot)

        ecs_0 = array(shape = (3,), type = 'Float64')
        ecs_0[0] =  Qdif
        ecs_0[1] =  self.modelo.DTQ1
        ecs_0[2] = -self.modelo.Q1 + self.K3*( self.Omi - Om )

        self.ecs_motor = ecs_0, ecs_1
```

B.10. colision.py

```
from ctypes import *

##############################################################################
#       TIPOS ADICIONALES DE LA LIBRERIA EN C                                #
##############################################################################
class CPunto(Structure):
    _fields_ = [
            ( "x", c_double * 3),
            ( "y", c_double * 3),
            ( "D", c_long      ),
            ( "n", c_double    ),
            ( "a", c_double    ),
            ( "b", c_double    ),
            ( "c", c_double    ),
            ( "d", c_double    ),
            ( "e", c_double    ),
            ( "f", c_double    ) ]


class CTren(Structure):
    _fields_ = [
            ( "M",          c_double                ),
            ( "Ixx",        c_double                ),
            ( "Iyy",        c_double                ),
            ( "Izz",        c_double                ),
            ( "Ixz",        c_double                ),
            ( "FC",         c_double * 3            ),
            ( "MC",         c_double * 3            ),
            ( "FB",         c_double * 3            ),
            ( "MB",         c_double * 3            ),
            ( "x",          c_double * 13           ),
            ( "t",          c_double                ),
            ( "T0",         c_double                ),
            ( "P",          POINTER(POINTER(CPunto)) ),
            ( "nP",         c_long                  ),
            ( "colision",   c_long                  ),
            ( "x0",         POINTER(c_double)       ),
            ( "x1",         POINTER(c_double)       ),
            ( "x2",         POINTER(c_double)       ),
            ( "xa",         POINTER(c_double)       ),
            ( "xi",         POINTER(c_double)       ),
            ( "F0",         POINTER(c_double)       ),
            ( "F1",         POINTER(c_double)       ),
            ( "F2",         POINTER(c_double)       ),
            ( "Fi",         POINTER(c_double)       ),
            ( "Fa",         POINTER(c_double)       ),
            ( "ti",         c_double                ),
            ( "t0",         c_double                ),
            ( "t1",         c_double                ),
            ( "t2",         c_double                ),
            ( "dt1",        c_double                ),
            ( "dt2",        c_double                ) ]

##############################################################################
#       INICIALIZACION DE LA LIBRERIA EN C                                   #
##############################################################################
c_colision = CDLL("./c_colision.so")

# struct Tren * malloc_tren(long)
c_colision.malloc_tren.arg_types = [c_long]
```

```
c_colision.malloc_tren.restype = POINTER(CTren)

# void free_tren(struct Tren *)
c_colision.free_tren.arg_tpes = [POINTER(CTren)]
c_colision.free_tren.restype = None

# void step_tren(struct Tren *, double)
c_colision.step_tren.arg_tpes = [POINTER(CTren), c_double]
c_colision.step_tren.restype = None

###############################################################################
#       FUNCIONES DE CONVERSION DE TIPOS                                      #
###############################################################################
def PyC_Punto(P):
    """
    A partir de un diccionario que representa un punto P devuelve una
    estructura de C.
    """
    C = CPunto()

    C.x[0] = P['x'][0]
    C.x[1] = P['x'][1]
    C.x[2] = P['x'][2]

    C.y[0] = P['y'][0]
    C.y[1] = P['y'][1]
    C.y[2] = P['y'][2]

    C.D = P['D']

    C.n = P['n']
    C.a, C.b, C.c = P['a'], P['b'], P['c']
    C.d, C.e, C.f = P['d'], P['e'], P['f']

    return C


def CPy_Punto(C, P):
    """
    A partir de la estructura en C de un punto asigna los valores en el
    diccionario de python que representa un punto.
    """

    P['x'][0] = C.x[0]
    P['x'][1] = C.x[1]
    P['x'][2] = C.x[2]

    P['y'][0] = C.y[0]
    P['y'][1] = C.y[1]
    P['y'][2] = C.y[2]

    P['D'] = C.D
    P['a'], P['b'], P['c'] = C.a, C.b, C.c
    P['d'], P['e'], P['f'] = C.d, C.e, C.f

###############################################################################
#       CLASE TREN                                                            #
###############################################################################
class Tren:
    def __init__(self, modelo):
        """
        modelo contiene los datos masicos:
```

```
            M, Ixx, Iyy, Izz, Ixz
        """
        self.modelo = modelo
        self.puntos = []

        def ceros(n):
            return [0.0 for x in range(n)]

        self.x  = ceros(13)
        self.F  = ceros(13)
        self.FC = ceros(3)
        self.MC = ceros(3)
        self.FB = ceros(3)
        self.MB = ceros(3)

    def init(self):
        """
        A partir de los puntos reserva la memoria necesaria
        """
        # pC es un puntero a la imagen en C
        nP = len(self.puntos)
        self.pC = c_colision.malloc_tren(nP)
        # Estas magnitudes se inicializan en la imagen y
        # ya no se tocan desde python
        for i in range(nP):
            self.pC.contents.P[i]= pointer(PyC_Punto(self.puntos[i]))
        self.colision = self.pC.contents.colision = 0

    def __del__(self):
        # Liberamos la memoria reservada dentro de C
        c_colision.free_tren(self.pC)

    def reset(self):
        self.colision = self.pC.contents.colision = 0

    def punto(self, x, n, a, b, c, d, e, f):
        self.puntos.append({
            'x':    x,
            'y':    [0., 0., 0.],
            'D':    0,
            'n':    n,
            'a':    a,
            'b':    b,
            'c':    c,
            'd':    d,
            'e':    e,
            'f':    f})

    def step(self, t):
        def copyPC(a, b, n):
            for i in range(n):
                b[i] = c_double(a[i])

        def copyCP(a, b, n):
            for i in range(n):
                b[i] = a[i]



        # De Pyton a C
        self.pC.contents.M   = c_double(self.modelo.M  )
        self.pC.contents.Ixx = c_double(self.modelo.Ixx)
```

```
        self.pC.contents.Iyy = c_double(self.modelo.Iyy)
        self.pC.contents.Izz = c_double(self.modelo.Izz)
        self.pC.contents.Ixz = c_double(self.modelo.Ixz)

        copyPC(self.FC, self.pC.contents.FC, 3)
        copyPC(self.MC, self.pC.contents.MC, 3)
        copyPC(self.FB, self.pC.contents.FB, 3)
        copyPC(self.MB, self.pC.contents.MB, 3)

        copyPC(self.x, self.pC.contents.x, 13)
        self.pC.contents.t = c_double(self.t)

        self.pC.contents.T0 = c_double(self.T0)

        # Llamada
        c_colision.step_tren(self.pC, c_double(t))

        # De C a Python
        copyCP(self.pC.contents.x,  self.x, 13)
        copyCP(self.pC.contents.F0, self.F, 13)
        self.t = self.pC.contents.t
        self.colision = self.pC.contents.colision
```

B.11. algebra.h

```
#ifndef ALGEBRA_H
#define ALGEBRA_H
/*****************************************************************************/
/*  CREACION DE VECTORES Y MATRICES                                          */
/*****************************************************************************/

// Crea un nuevo vector, como un puntero de 3 doubles
double * newvector();

// Crea una nuva matriz, como un puntero de 9 doubles
double * newmatrix();

/*****************************************************************************/
/*      OPERACIONES DE MATRICES
/*****************************************************************************/

// Escritura de elementos
void setM(double *M, int i, int j, double v);

// Lectura de elementos
double getM(double *M, int i, int j);

// Convierte a identidad
void identity(double *R);

// Multiplica por escalar
void dotSM(double s, double *M, double *R);

// Multiplica dos matrices
void dotMM(double *M, double *N, double *R);

// Suma dos matrices
void addMM(double *M, double *N, double *R);
```

```
// Resta dos matrices
void subMM(double *M, double *N, double *R);

// Invierte la matriz
void invertM(double *M, double *R);

// Calcula la transpuesta
void transposeM(double *M, double *R);

/*****************************************************************************/
/*      OPERACIONES DE VECTORES
/*****************************************************************************/
// Copia v1 a v2
void copyVV(double *v1, double *v2);

// Divide vector por escalar
void divVS(double *v, double s, double *r);

// Multiplica escalar por vector
void dotSV(double s, double *v, double *r);

// Producto escalar de los dos vectores
double dotVV(double *v1, double *v2);

// Matriz antisimetrica de V
void antiV(double *v, double *R);

// Producto vectorial de dos vectores
void cross(double *v1, double *v2, double *r);

// Resta dos vectores
void subVV(double *v1, double *v2, double *r);

// Suma dos vectores
void addVV(double *v1, double *v2, double *r);

/*****************************************************************************/
/*      OPERACIONES MIXTAS
/*****************************************************************************/
// Multiplica matriz por vector
void dotMV(double *M, double *v, double *r);


/*****************************************************************************/
/*      ARRAYS                                                               */
/*****************************************************************************/
// Crea un nuevo array de tamno n
double * newarray(long n);

// Copia a en b, ambos miden n
void copyAA(double *a, double *b, int n);

// Multiplica por el escalar
void dotSA(double k, double *a, double *r, int n);

// Suma arrays
void addAA(double *a, double *b, double *r, int n);

// Resta arrays
void subAA(double *a, double *b, double *r, int n);

// Devuelve el maximo
```

```
double maxA(double *a, int n);

#endif
```

B.12. algebra.c

```
#include "algebra.h"

/*****************************************************************************/
/*  CREACION DE VECTORES Y MATRICES                                          */
/*****************************************************************************/
double *newvector() {
    return malloc(sizeof(double)*3);
}

double *newmatrix() {
    return malloc(sizeof(double)*9);
}

/*****************************************************************************/
/*      OPERACIONES DE MATRICES
/*****************************************************************************/
void setM(double *M, int i, int j, double v) {
    M[3*i + j] = v;
}

double getM(double *M, int i, int j) {
    return M[3*i + j];
}

void identity(double *R)
{
    R[0] = 1.0;
    R[1] = 0.0;
    R[2] = 0.0;

    R[3] = 0.0;
    R[4] = 1.0;
    R[5] = 0.0;

    R[6] = 0.0;
    R[7] = 0.0;
    R[8] = 1.0;
}

void dotSM(double s, double *M, double *R)
{
    int i;
    for (i=0;i<9;++i) {
        R[i] = s*M[i];
    }
}

void dotMM(double *M, double *N, double *R)
{
    R[0] = M[0]*N[0] + M[1]*N[3] + M[2]*N[6];
    R[1] = M[0]*N[1] + M[1]*N[4] + M[2]*N[7];
    R[2] = M[0]*N[2] + M[1]*N[5] + M[2]*N[8];
```

```
    R[3] = M[3]*N[0] + M[4]*N[3] + M[5]*N[6];
    R[4] = M[3]*N[1] + M[4]*N[4] + M[5]*N[7];
    R[5] = M[3]*N[2] + M[4]*N[5] + M[5]*N[8];

    R[6] = M[6]*N[0] + M[7]*N[3] + M[8]*N[6];
    R[7] = M[6]*N[1] + M[7]*N[4] + M[8]*N[7];
    R[8] = M[6]*N[2] + M[7]*N[5] + M[8]*N[8];
}

void addMM(double *M, double *N, double *R)
{
    int i;
    for (i=0; i<9; ++i) {
        R[i] = M[i] + N[i];
    }
}

void subMM(double *M, double *N, double *R)
{
    int i;
    for (i=0; i<9; ++i) {
        R[i] = M[i] - N[i];
    }
}

void invertM(double *M, double *R)
{
    double M0 = M[0];
    double M1 = M[1];
    double M2 = M[2];
    double M3 = M[3];
    double M4 = M[4];
    double M5 = M[5];
    double M6 = M[6];
    double M7 = M[7];
    double M8 = M[8];

    double det = M0*(M4*M8 - M5*M7) +
                 M1*(M5*M6 - M3*M8) +
                 M2*(M3*M7 - M4*M6);

    R[0] = (M4*M8 - M5*M7)/det;
    R[1] = (M2*M7 - M1*M8)/det;
    R[2] = (M1*M5 - M2*M4)/det;

    R[3] = (M5*M6 - M3*M8)/det;
    R[4] = (M0*M8 - M2*M6)/det;
    R[5] = (M2*M3 - M0*M5)/det;

    R[6] = (M3*M7 - M4*M6)/det;
    R[7] = (M1*M6 - M0*M7)/det;
    R[8] = (M0*M4 - M1*M3)/det;
}

void transposeM(double *M, double *R)
{
    R[0] = M[0];
    R[1] = M[3];
    R[2] = M[6];

    R[3] = M[1];
    R[4] = M[4];
```

```
    R[5] = M[7];

    R[6] = M[2];
    R[7] = M[5];
    R[8] = M[8];
}


/*****************************************************************************/
/*      OPERACIONES DE VECTORES
/*****************************************************************************/
void copyVV(double *v1, double *v2)
{
    v2[0] = v1[0];
    v2[1] = v1[1];
    v2[2] = v1[2];
}

void divVS(double *v, double s, double *r)
{
    r[0] = v[0]/s;
    r[1] = v[1]/s;
    r[2] = v[2]/s;
}

void dotSV(double s, double *v, double *r)
{
    r[0] = s*v[0];
    r[1] = s*v[1];
    r[2] = s*v[2];
}

double dotVV(double *v1, double *v2) {
    return v1[0]*v2[0] + v1[1]*v2[1] + v1[2]*v2[2];
}

void antiV(double *v, double *R)
{
    R[0] =  0.0;
    R[1] = -v[2];
    R[2] =  v[1];
    R[3] =  v[2];
    R[4] =  0.0;
    R[5] = -v[0];
    R[6] = -v[1];
    R[7] =  v[0];
    R[8] =  0.0;
}

void cross(double *v1, double *v2, double *r)
{
    r[0] =  v1[1]*v2[2] - v1[2]*v2[1];
    r[1] =  v1[2]*v2[0] - v1[0]*v2[2];
    r[2] =  v1[0]*v2[1] - v1[1]*v2[0];
}

void subVV(double *v1, double *v2, double *r)
{
    r[0] = v1[0] - v2[0];
    r[1] = v1[1] - v2[1];
    r[2] = v1[2] - v2[2];
}
```

```
// Suma v1-v2, almacena en r
void addVV(double *v1, double *v2, double *r)
{
    r[0] = v1[0] + v2[0];
    r[1] = v1[1] + v2[1];
    r[2] = v1[2] + v2[2];
}

// Modulo del vector V
double modV(double *v) {
    return sqrt(v[0]*v[0] + v[1]*v[1] + v[2]*v[2]);
}

/*****************************************************************************/
/*      OPERACIONES MIXTAS
/*****************************************************************************/
void dotMV(double *M, double *v, double *r)
{
    r[0] = M[0]*v[0] + M[1]*v[1] + M[2]*v[2];
    r[1] = M[3]*v[0] + M[4]*v[1] + M[5]*v[2];
    r[2] = M[6]*v[0] + M[7]*v[1] + M[8]*v[2];
}
/*****************************************************************************/
/*      OPERACIONES DE ARRAYS
/*****************************************************************************/
double * newarray(long n) {
    return (double *) malloc(sizeof(double)*n);
}

void copyAA(double *a, double *b, int n)
{
    int i;
    for (i=0; i<n; ++i) {
        b[i] = a[i];
    }
}

void dotSA(double k, double *a, double *r, int n)
{
    int i;
    for (i=0; i<n; ++i) {
        r[i] = k*a[i];
    }
}

void addAA(double *a, double *b, double *r, int n)
{
    int i;
    for (i=0; i<n; ++i) {
        r[i] = a[i] + b[i];
    }
}

void subAA(double *a, double *b, double *r, int n)
{
    int i;
    for (i=0; i<n; ++i) {
        r[i] = a[i] - b[i];
    }
}
```

```
double maxA(double *a, int n)
{
    double m = fabs(a[0]);
    int i;
    for (i=1; i<n; ++i) {
        double ma = fabs(a[i]);
        if (ma>m) {
            m = ma;
        }
    }
    return m;
}
```

B.13. colision.h

```
#ifndef COLISION_H
#define COLISION_H

#define VECTOR_SIZE 3
#define ARRAY_SIZE 13

struct Punto
{
    double x[3];
    double y[3];
    long D;

    double n;

    double a;
    double b;

    double c;
    double d;

    double e;
    double f;
};


struct Tren
{

    /* ESTOS ATRIBUTOS DEBEN INICIARSE DESDE PYTHON */
    double M;
    double Ixx;
    double Iyy;
    double Izz;
    double Ixz;

    double FC[VECTOR_SIZE];
    double MC[VECTOR_SIZE];
    double FB[VECTOR_SIZE];
    double MB[VECTOR_SIZE];

    double x[ARRAY_SIZE];
    double t;

    double T0;
```

```
    struct Punto ** P;
    long nP;
    /* FIN DE ATRIBUTOS */

    // Estado del integrador
    long colision;

    double * x0;
    double * x1;
    double * x2;

    double * xa;
    double * xi;

    double * F0;
    double * F1;
    double * F2;
    double * Fi;

    double  * Fa;

    double ti;
    double t0;
    double t1;
    double t2;

    double dt1;
    double dt2;

};

struct Tren * malloc_tren(long);
void free_tren(struct Tren *);
void step_tren(struct Tren *, double );
void F(double *, double, double *, struct Tren *);
#endif
```

B.14. colision.c

```
/*
 * Compilar con gcc -fPIC -shared algebra.c colision.c -o c_colision
 */

//#define DEBUG_MSG

#include <math.h>
#include "algebra.h"
#include "colision.h"

struct Tren * malloc_tren(long nP)
{
    struct Tren * T = malloc(sizeof(struct Tren));

    T->x0 = newarray(ARRAY_SIZE);
    T->x1 = newarray(ARRAY_SIZE);
    T->x2 = newarray(ARRAY_SIZE);

    T->F0 = newarray(ARRAY_SIZE);
```

```
    T->F1 = newarray(ARRAY_SIZE);
    T->F2 = newarray(ARRAY_SIZE);
    T->Fi = newarray(ARRAY_SIZE);

    T->P = malloc(nP*sizeof(struct Punto *));
    T->nP = nP;
#ifdef DEBUG_MSG
    printf("malloc_tren: memoria inicializada para %p\n", T);
#endif
    return T;
}

void free_tren(struct Tren * T)
{
    free(T->x0); free(T->x1);
    free(T->x2);
    free(T->F0); free(T->F1);
    free(T->F2); free(T->Fi);
    free(T->P);
#ifdef DEBUG_MSG
    printf("malloc_tren: memoria liberada para %p", T);
#endif
}

void step_tren(struct Tren * T, double t)
{
#ifdef DEBUG_MSG
    printf("step_tren llamada desde %p, t = %f\n", T, t);
#endif
    // Paso minimo de integracion
    double dtmin = 1e-8;
    // Paso maximo de integracion
    double dtmax = 0.01;

    // Arrays temporales
    double * a1 = newarray(ARRAY_SIZE);
    double * a2 = newarray(ARRAY_SIZE);
    double * a3 = newarray(ARRAY_SIZE);

    if (T->colision == 0 && t>T->t)
    {
#ifdef DEBUG_MSG
        printf("step_tren: iniciando integrador\n");
#endif
        double tf = T->t + dtmin;
        // Condiciones iniciales
        T->xi = T->x;
        T->ti = T->t;
        F(T->xi, T->ti, T->Fi, T);

        copyAA(T->xi, T->x1, ARRAY_SIZE);
        copyAA(T->Fi, T->F1, ARRAY_SIZE);
        T->t1 = T->ti;
        T->dt1 = dtmin;

        // Metodo de Euler: x0 = x1 + dt*F1
        dotSA( T->dt1, T->F1, a1,    ARRAY_SIZE );
        addAA( a1,     T->x1, T->x0, ARRAY_SIZE );

        // Guardamos otros resultados
        T->t = T->t0 = tf;
        F(T->x0, T->t0, T->F0, T);
```

```
        copyAA(T->x0, T->x, ARRAY_SIZE);
//        T->colision = 1;
#ifdef DEBUG_MSG
        printf("step_tren: integrador iniciado\n");
#endif
    }
    if (t>(T->t0 + dtmin))
    {
#ifdef DEBUG_MSG
        printf("step_tren: dando un paso\n");
#endif
        double * xp = newarray(ARRAY_SIZE);
        double * Fp = newarray(ARRAY_SIZE);

        // Se ha producido una colision en algun punto del intervalo?
        long colision = 0;

        // guardamos la memoria de x2 y F2
        T->xa = T->x2;
        T->Fa = T->F2;

        // desplazamos las magnitudes
        T->x2 = T->x1;
        T->F2 = T->F1;
        T->t2 = T->t1;

        T->x1 = T->x0;
        T->F1 = T->F0;
        T->t1 = T->t0;

        T->dt2 = T->dt1;
        T->dt1 = t - T->t0;

        int STOP = 0;
        // STOP si Tn<T0 y t0 > t - dtmin
        while (!STOP)
        {
            T->t0  = T->t1 + T->dt1;
            // Predictor: xp, Fp
#ifdef DEBUG_MSG
            printf("Predictor\n");
#endif
            double bp1 = (2*T->dt2 + T->dt1)/(2*T->dt2);
            double bp2 = -T->dt1/(2*T->dt2);

            dotSA(T->dt1*bp1, T->F1, a1,  ARRAY_SIZE);
            dotSA(T->dt1*bp2, T->F2, a2,  ARRAY_SIZE);
            addAA(a1,     a2,    a3,      ARRAY_SIZE);
            addAA(T->x1,  a3,    xp,      ARRAY_SIZE);

            F(xp, T->t0, Fp, T);
            if (T->colision == 1) {
                colision = 1;
            }

            // Corrector: x0, F0
#ifdef DEBUG_MSG
            printf("Corrector\n");
#endif
            double bc0 = 0.5;
            double bc1 = 0.5;
```

```
            T->x0 = T->xa;
            T->F0 = T->Fa;

            dotSA(T->dt1*bc0, Fp,    a1,    ARRAY_SIZE);
            dotSA(T->dt1*bc1, T->F1, a2,    ARRAY_SIZE);
            addAA(a1,         a2,    a3,    ARRAY_SIZE);
            addAA(T->x1,      a3,    T->x0, ARRAY_SIZE);

            F(T->x0, T->t0, T->F0, T);
            if (T->colision == 1) {
                colision = 1;
            }

            // Error: Tn = |Cc/(Cp + Cc)*(x0p - x00)|
            double Cp = (2*T->dt1 + 3*T->dt2)/(12*T->dt1);
            double Cc = -1./12.;

            subAA(xp, T->x0, a1, ARRAY_SIZE);
            double Tn = fabs((Cc/(Cp + Cc)*maxA(a1, ARRAY_SIZE)));

            if (Tn>1.1*T->T0)
            {
                // Repetimos el paso
                T->t0 = T->t1;
                // Nuevo paso de integracion
                if (Tn>0) {
                    T->dt1 *= pow((T->T0/Tn),(1./3.));
                    if (T->dt1>dtmax) {
                        T->dt1 = dtmax;
                    }
                }
            } else if (T->t0<t-dtmin)
            {
                // No hace falta repetir paso, pero
                // hay que seguir dandolos --> STOP = 0

                // Avanzamos las variables
                T->xa = T->x2;
                T->Fa = T->F2;

                T->x2 = T->x1;
                T->F2 = T->F1;
                T->t2 = T->t1;

                T->x1 = T->x0;
                T->F1 = T->F0;
                T->t1 = T->t0;

                T->dt2 = T->dt1;
                // Nuevo paso de integracion
                if (Tn>0) {
                    T->dt1 *= pow((T->T0/Tn),(1./3.));
                    if (T->dt1>dtmax) {
                        T->dt1 = dtmax;
                    }
                }
            } else {
                STOP = 1;
            }
#ifdef DEBUG_MSG
             printf("step_tren: paso dado con Tn, T0 = %f, %f\n", Tn, T->T0);
#endif
```

```
        }

        // Estamos en colision si en alguna parte del intervalo ha
        // habido colision
        T->colision = colision;
        copyAA(T->x0, T->x, ARRAY_SIZE);
        T->t = T->t0;
        free(xp); free(Fp);

#ifdef DEBUG_MSG
        printf("step_tren: paso dado, integrador finalizado\n");
#endif
    }
    free(a1); free(a2); free(a3);
}


/*
 * De Tren necesita:
 *      FC, MC, FB, MB
 *      puntos
 *
 * En Tren coloca:
 *      colision
 */
void F(double *x, double t, double *y, struct Tren * T)
{
#ifdef DEBUG_MSG
    printf("Funcion F llamada para t=%f\n", t);
#endif
    // Matrices de cambio de ejes cuerpo - ejes colision
    double * LBC = newmatrix();
    double * LCB = newmatrix();

    // Fuerzas de reaccion
    double *  FR = newvector();
    double *  MR = newvector();

    // Fuerzas totales
    double *  FT = newvector();
    double *  MT = newvector();

    // Posicion del punto en ejes colision
    double *  dC = newvector();

    // Desplzamiento y velocidad de desplazamiento
    double *  eC = newvector();
    double *  vC = newvector();

    // Posicion, velocidad y vel. angular CM
    double * rC = newvector();
    double * vB = newvector();
    double * wB = newvector();

    // Vectores temporales
    double * v1 = newvector();
    double * v2 = newvector();
    double * v3 = newvector();
    double * v4 = newvector();
    double * v5 = newvector();
```

```
    // Componentes de la velocidad en ejes cuerpo
    double u = vB[0] = x[0];
    double v = vB[1] = x[1];
    double w = vB[2] = x[2];

    // Componentes de la velocidad angular en ejes cuerpo
    double p = wB[0] = x[3];
    double q = wB[1] = x[4];
    double r = wB[2] = x[5];

    // Componentes del cuaternio
    double q0 = x[6];
    double q1 = x[7];
    double q2 = x[8];
    double q3 = x[9];

    // Componentes del vector de posicion
    double xi = rC[0] = x[10];
    double yi = rC[1] = x[11];
    double zi = rC[2] = x[12];

    setM(LCB, 0, 0, 1 - 2*( q2*q2 + q3*q3));
    setM(LCB, 0, 1,     2*(-q0*q3 + q1*q2));
    setM(LCB, 0, 2,     2*( q0*q2 + q1*q3));
    setM(LCB, 1, 0,     2*( q0*q3 + q1*q2));
    setM(LCB, 1, 1, 1 - 2*( q1*q1 + q3*q3));
    setM(LCB, 1, 2,     2*(-q0*q1 + q2*q3));
    setM(LCB, 2, 0,     2*(-q0*q2 + q1*q3));
    setM(LCB, 2, 1,     2*( q0*q1 + q2*q3));
    setM(LCB, 2, 2, 1 - 2*( q1*q1 + q2*q2));

    transposeM(LCB, LBC);
#ifdef DEBUG_MSG
    printf("F: matriz LBC y LCB calculadas\n");
#endif
    FR[0] = FR[1] = FR[2] = 0.0;
    MR[0] = MR[1] = MR[2] = 0.0;

    // Algun punto contacta?
    long contacto = 0;
    // Margen para el contacto
    double margen = 0.1;

    // Para cada punto del tren
    long iP;
    for (iP = 0; iP<(T->nP); ++iP)
    {
#ifdef DEBUG_MSG
        printf("F: punto %ld de %ld\n", iP + 1, T->nP);
#endif
        struct Punto *P = T->P[iP];

        // Calculamos dC = r + LCB*P.x
        dotMV(LCB, P->x, v1);
        addVV(v1,  rC,  dC);

        if (dC[2]>0)
        {
#ifdef DEBUG_MSG
            printf("F: punto sin contacto\n");
#endif
            // Si no hay colision quitamos el punto de contacto
```

```
            P->D = 0;
            // De todas formas miramos el margen
//            if (dC[2]<margen) {
//                contacto = 1;
//            }
            // y pasamos al siguiente punto
            continue;
        } else
        {
            contacto = 1;
            // Hay colision
            if (P->D == 0)
            {
#ifdef DEBUG_MSG
            printf("F: creando contacto\n");
#endif
                // Pero no habia punto de contacto, lo creamos
                P->y[0] = dC[0];
                P->y[1] = dC[1];
                P->y[2] = 0;
                // Recordamos que ya esta creado
                P->D = 1;
            }
            // Calculamos eC = dC - P.y
            subVV(dC,  P->y, eC);

            // Calculamos vC = LCB*(vB + wB^P.x)
            cross(wB,  P->x, v1);
            addVV(vB,  v1,  v2);
            dotMV(LCB, v2,  vC);

            double es = eC[0];
            double et = eC[1];
            double en = eC[2];

            double vs = vC[0];
            double vt = vC[1];
            double vn = vC[2];

            // Fuerza normal
            double N1 = -P->a*en;
            double N2;
            if (vn<0) {
                N2 = -P->b*vn;
            } else {
                N2 = 0;
            }
            double N = N1 + N2;

            // Fuerzas tangenciales
            double S = -(P->c*es + P->d*vs);
            double T = -(P->e*et + P->f*vt);
#ifdef DEBUG_MSG
            printf("c*es = %f, d*vs = %f\n", P->c*es, P->d*vs);
            printf("e*et = %f, f*vt = %f\n", P->e*et, P->f*vt);
#endif

            // Siempre se oponen a la velocidad
            int mismo_signo(double a, double b){
                return ((a<0 && b<0) || (a>=0 && b>=0));
            }
//            if (mismo_signo(S, vs)){
```

```
//                 S = 0;
//             }
//             if (mismo_signo(T, vt)){
//                 T = 0;
//             }
            // Comprobamos que la fuerza de rozamiento esta dentro
            // de los limietes dados por la normal y el coeficiente
            // de rozamiento
            double R = sqrt(S*S + T*T);
            double Rmax = P->n*N;
            // Se pueden producir errores numericos
            if (Rmax<0) {
                Rmax = 0;
            }

#ifdef DEBUG_MSG
            printf("F: R = %f, Rmax = %f, n = %f, N=%f\n", R, Rmax, P->n, N);
#endif
            if (R>Rmax)
            {
                // Hay deslizamiento, desplazamos el punto de contacto
                double k;
                if (es!=0 || et!=0) {
                    k = Rmax/sqrt(P->c*P->c*es*es + P->e*P->e*et*et);
                } else {
                    k = 1;
                }
                v1[0] = (1-k)*es;
                v1[1] = (1-k)*et;
                v1[2] = 0;
                addVV (P->y, v1,  v2 );
                copyVV(v2,   P->y    );
                // Trucamos el rozamiento
                S = Rmax*S/R;
                T = Rmax*T/R;
            }
            // Sumamos las fuerzas y momentos de reaccion
            v1[0] = S;
            v1[1] = T;
            v1[2] = N;
#ifdef DEBUG_MSG
            printf("F: S,T,N = %f,%f,%f\n", S, T, N);
#endif
            dotMV (LBC,  v1, v2);
            cross (P->x, v2, v3);
            addVV (FR,   v2, v4);
            addVV (MR,   v3, v5);
            copyVV(v4,   FR    );
            copyVV(v5,   MR    );
        }
    } // Fin para el punto
    // Ya hemos calculado todas las reacciones, calculamos
    // las fuerzas y momentos totales FT, MT
    dotMV (LBC, T->FC, v1);
    addVV (v1,  FR,    v2);
    addVV (v2,  T->FB, v3);
    copyVV(v3,  FT);

    dotMV (LBC, T->MC, v1);
    addVV (v1,  MR,    v2);
    addVV (v2,  T->MB, v3);
    copyVV(v3,  MT);
```

```
#ifdef DEBUG_MSG
    printf("F: FR = %f, %f, %f\n", FR[0], FR[1], FR[2]);
    printf("F: MR = %f, %f, %f\n", MR[0], MR[1], MR[2]);
    printf("F: FT = %f, %f, %f\n", FT[0], FT[1], FT[2]);
    printf("F: MT = %f, %f, %f\n", MT[0], MT[1], MT[2]);
#endif
    // Unos alias
    double M   = T->M;
    double Ixx = T->Ixx;
    double Iyy = T->Iyy;
    double Izz = T->Izz;
    double Ixz = T->Ixz;

    // Aceleraciones de inercia
    double axI = -( w*q - v*r );
    double ayI = -( u*r - w*p );
    double azI = -( v*p - u*q );
    // Momentos de inercia
    double LI =  ( Iyy - Izz )*q*r + Ixz*p*q;
    double MI =  ( Izz - Ixx )*r*p + Ixz*( r*r - p*p );
    double NI =  ( Ixx - Iyy )*p*q + Ixz*q*r;

    double l = LI + MT[0];
    double m = MI + MT[1];
    double n = NI + MT[2];
    double k = Ixx*Izz - Ixz*Ixz;

    // Ecuaciones de fuerzas y momentos
    y[0] = axI + FT[0]/M;
    y[1] = ayI + FT[1]/M;
    y[2] = azI + FT[2]/M;
    y[3] = (Izz*l + Ixz*n)/k;
    y[4] = m/Iyy;
    y[5] = (Ixz*l + Ixx*n)/k;

    dotMV(LCB, wB, v1);
    double pi = v1[0];
    double qi = v1[1];
    double ri = v1[2];

    // Relacion cinematica para los cuaternios
    y[6] = -0.5*(  q1*pi + q2*qi + q3*ri );
    y[7] = -0.5*( -q0*pi - q3*qi + q2*ri );
    y[8] = -0.5*(  q3*pi - q0*qi - q1*ri );
    y[9] = -0.5*( -q2*pi + q1*qi - q0*ri );

    dotMV(LCB, vB, v1);
    double ui = v1[0];
    double vi = v1[1];
    double wi = v1[2];

    // Relacion cinematica para la posicion del CM
    y[10] = ui;
    y[11] = vi;
    y[12] = wi;

    T->colision = contacto;
#ifdef DEBUG_MSG
    printf("F: Fin\n");
#endif
    // Liberamos memoria usada
```

```
    free(v1);  free(v2);  free(v3); free(v4); free(v5);
    free(FR);  free(MR);  free(FT); free(MT);
    free(dC);  free(eC);  free(vC);
    free(rC);  free(vB);  free(wB);
    free(LBC); free(LCB);
} // Fin de F
```

B.15. integrador.py

```
from historial import *
from numarray import arange
from matematicas import *

class Euler:
    # Esta se llama en input.py
    def __init__(self):
        pass

    # Esta en FlightGear.reset()
    def init(self, F, ti, xi):
        """
        Coloca las condiciones iniciales
        """
        self.F, self.ti, self.xi = F, ti, xi

        self.x0 = xi
        self.t0 = ti
        self.F0 = self.F(xi, ti)

    # En FlightGear.loop()
    def __call__(self, t):
        """
        Calcula la respuesta hasta el instante t
        """

        if t<self.t0:
            return

        self.x1 = self.x0
        self.t1 = self.t0
        self.F1 = self.F0

        self.x0 = self.x1 + self.F1*(t - self.t1)
        self.t0 = t
        self.F0 = self.F(self.x0, self.t0)

class RungeKutta4:
    def __init__(self):
        pass

    def init(self, F, ti, xi):
        """
        Coloca las condiciones iniciales
        """
        self.F, self.ti, self.xi = F, ti, xi

        self.x0 = xi
        self.t0 = ti
        self.F0 = self.F(xi, ti)
```

```
    def __call__(self, t):
        # La primera vez obtenemos el punto inicial
        # de las condiciones iniciales

        if t<self.t0:
            return

        self.t1 = self.t0
        self.x1 = self.x0
        self.F1 = self.F0

        # paso
        h = t - self.t1

        # Estimaciones de la derivada
        k1 = self.F1
        k2 = self.F(self.x1 + h*k1/2., self.t1 + h/2.)
        k3 = self.F(self.x1 + h*k2/2., self.t1 + h/2.)
        # Calculamos el nuevo punto
        k4 = self.F(self.x1 + h*k3, self.t1 + h)

        self.x0 = self.x1 + h/6.*(k1 + 2*k2 + 2*k3 + k4)
        self.t0 = t
        self.F0 = self.F(self.x0, self.t0)

class AdamsBashforth2:
    """
    Esquema de Adams-Bashforth de orden 2, de paso variable.
    Nota:
        No calcula el paso para integrar hasta el instante
        t, solo da un paso, por lo que es responsabilidad
        de otra funcion elegir el paso.
    """

    def __init__(self):
        pass

    def init(self, F, ti, xi):
        """
        Coloca las condiciones iniciales:

            x(ti) = xi

        y define F(x, t), tal que:

            dx
            -- = F(x, t)
            dt
        """
        self.F, self.ti, self.xi = F, ti, xi
        # x0, t0 guardan los ultimos resultados, de momento
        # esos son las condiciones iniciales
        self.x0 = xi
        self.t0 = ti
        self.F0 = self.F(xi, ti)

        self.ci = 0

    def __call__(self, t):
        """
```

```
        Integra hasta el instante de tiempo t
        El resultado e instante final se guardan en:
            x0
            t0
        Si el esquemas es multipaso de orden p, los anteriores
        pasos se encuenran en:
            x1, x2, ..., xp
            t1, t2, ..., tp
        """
        dtmin = 1e-8
        # Si pedimos un tiempo equivocado, ignoramos
        if t <= self.ti or ( self.ci != 0 and t<self.t0 ):
            return

        # Hemos arrancado el esquema?
        if self.ci == 0:
            self.ci = 1

            self.x1 = self.x0
            self.t1 = self.t0
            self.t0 = self.t1 + dtmin

            self.dt1 = t - self.t0
            self.F1 = self.F0

            # Un Euler para la primera aproximacion
            self.x0 = self.x1 + self.dt1*self.F1
            self.F0 = self.F(self.x0, self.t0)

        self.t2 = self.t1
        self.t1 = self.t0
        self.t0 = t

        self.dt2 = self.dt1
        self.dt1 = t - self.t1

        self.x2 = self.x1
        self.x1 = self.x0

        self.F2 = self.F1
        self.F1 = self.F0

        b1 = (2*self.dt2 + self.dt1)/2./self.dt2
        b2 = -self.dt1/2/self.dt2

        self.x0 = self.x1 + self.dt1*(b1*self.F1 + b2*self.F2)
        self.F0 = self.F(self.x0, self.t0)

class AdamsBashforth3:
    """
    Esquema de Adams-Bashforth de orden 3, de paso variable.
    Nota:
        No calcula el paso para integrar hasta el instante
        t, solo da un paso, por lo que es responsabilidad
        de otra funcion elegir el paso.
    """

    def __init__(self):
        pass

    def init(self, F, ti, xi):
        """
```

```
        Coloca las condiciones iniciales:

            x(ti) = xi

        y define F(x, t), tal que:

            dx
            -- = F(x, t)
            dt
        """
        self.F, self.ti, self.xi = F, ti, xi
        # x0, t0 guardan los ultimos resultados, de momento
        # esos son las condiciones iniciales
        self.x0 = xi
        self.t0 = ti
        self.F0 = self.F(xi, ti)

        self.ci = 0

    def __call__(self, t):
        """
        Integra hasta el instante de tiempo t
        El resultado e instante final se guardan en:
            x0
            t0
        Si el esquemas es multipaso de orden p, los anteriores
        pasos se encuenran en:
            x1, x2, ..., xp
            t1, t2, ..., tp
        """
        dtmin = 1e-8

        # Si pedimos un tiempo equivocado, ignoramos
        if t <= self.ti or ( self.ci != 0 and t<self.t0 ):
            return

        # Hemos arrancado el esquema?
        if self.ci == 0:
            self.ci = 1

            self.x1 = self.x0
            self.t1 = self.t0
            self.t0 = self.t1 + dtmin

            self.dt1 = t - self.t0
            self.F1 = self.F0

            # Un Euler para la primera aproximacion
            self.x0 = self.x1 + self.dt1*self.F1
            self.F0 = self.F(self.x0, self.t0)

        elif self.ci == 1:
            self.ci = 2

            self.t2 = self.t1
            self.t1 = self.t0

            self.dt2 = self.dt1
            self.dt1 = t - self.t1

            self.x2 = self.x1
            self.x1 = self.x0
```

```
            self.F2 = self.F1
            self.F1 = self.F0

            b1 = (2*self.dt2 + self.dt1)/2./self.dt2
            b2 = -self.dt1/2/self.dt2

            # Un AdamsBashForth2
            self.x0 = self.x1 + self.dt1*(b1*self.F1 + b2*self.F2)
            self.t0 = t
            self.F0 = self.F(self.x0, self.t0)

        else:
            self.t3 = self.t2
            self.t2 = self.t1
            self.t1 = self.t0

            self.dt3 = self.dt2
            self.dt2 = self.dt1
            self.dt1 = t - self.t1

            self.x3 = self.x2
            self.x2 = self.x1
            self.x1 = self.x0

            self.F3 = self.F2
            self.F2 = self.F1
            self.F1 = self.F0

            dt1 = self.dt1
            dt2 = self.dt2
            dt3 = self.dt3

            b1 =  1. + dt1*(2*dt1 + 6*dt2 + 3*dt3)/(6*dt2*(dt2 + dt3))
            b2 = -dt1*(2*dt1 + 3*dt2 + 3*dt3)/(6*dt2*dt3)
            b3 =  dt1*(2*dt1 + 3*dt2)/(6*dt3*(dt2 + dt3))

            self.x0 = self.x1 + self.dt1*(b1*self.F1 + b2*self.F2
                    + b3*self.F3)
            self.t0 = t
            self.F0 = self.F(self.x0, self.t0)

class ABM2:
    """
    Predictor-Corrector basado en un Adams-Bashforth 2, y un Adams-Moulton
    1 de paso variable.
    Nota:
        No calcula el paso para integrar hasta el instante
        t, solo da un paso, por lo que es responsabilidad
        de otra funcion elegir el paso.
    """

    def __init__(self):
        pass

    def init(self, F, ti, xi):
        """
        Coloca las condiciones iniciales:

            x(ti) = xi

        y define F(x, t), tal que:
```

```
            dx
            -- = F(x, t)
            dt
        """
        self.F, self.ti, self.xi = F, ti, xi
        # x0, t0 guardan los ultimos resultados, de momento
        # esos son las condiciones iniciales
        self.x0 = xi
        self.t0 = ti
        self.F0 = self.F(xi, ti)

        self.ci = 0

    def __call__(self, t):
        """
        Integra hasta el instante de tiempo t
        El resultado e instante final se guardan en:
            x0
            t0
        Si el esquemas es multipaso de orden p, los anteriores
        pasos se encuenran en:
            x1, x2, ..., xp
            t1, t2, ..., tp
        """
        # Si pedimos un tiempo equivocado, ignoramos
        if t <= self.ti or ( self.ci != 0 and t<self.t0 ):
            return

        # Hemos arrancado el esquema?
        if self.ci == 0:
            self.ci = 1

            self.x1 = self.x0
            self.t1 = self.t0

            self.dt1 = t - self.t1
            self.F1 = self.F0

            # Un Euler para la primera aproximacion
            self.x0 = self.x1 + self.dt1*self.F1
            self.t0 = t
            self.F0 = self.F(self.x0, self.t0)
        else:
            self.t2 = self.t1
            self.t1 = self.t0

            self.dt2 = self.dt1
            self.dt1 = t - self.t1

            self.x2 = self.x1
            self.x1 = self.x0

            self.F2 = self.F1
            self.F1 = self.F0

            b1 = (2*self.dt2 + self.dt1)/2./self.dt2
            b2 = -self.dt1/2/self.dt2

            # Punto estimado por el el predictor
            xp = self.x1 + self.dt1*(b1*self.F1 + b2*self.F2)
            Fp = self.F(xp, self.t0)
```

```
            # Paso del corrector
            self.x0 = self.x1 + self.dt1*0.5*(Fp + self.F1)
            self.t0 = t
            self.F0 = self.F(self.x0, self.t0)
```

B.16. historial.py

```
# -*- coding: latin-1 -*-
"""
Contiene diversas utilidades para la gestion de valores que se obtienen en
forma de series temporales:
"""

import time

class Historial:
    """
    La clase historial se compone de una lista cuyo primer elemento ha sido el
    ultimo en añadirse, su segundo elemento el penultimo, ... asi hasta el
    n-esimo elemento a partir del cual no se han guardado mas datos.
    """

    def __init__(self, longitud):
        """
        longitud: Tamaño del buffer para el historial, es decir, guarda tantos
        valores pasados como se especifica en longitud.
        """

        # Creamos la lista con la lonigtud especificada
        self.longitud = longitud
        self.valores  = [None for x in range(longitud)]

        # señala la posicion del ultimo elemento añadido en la lista
        self.zero = -1

    def __getitem__(self, k):
        """
        Elemento k-veces anterior, es decir 0 es el mas reciente, 1 el anterior,
        etc... asi hasta tantos elementos como se especificaron con el
        parámetro longitud al crear la instancia.
        """

        return self.valores[( self.zero - k )%self.longitud]

    def __setitem__(self, k, v):
        """
        Asigna el valor v al elemento k-veces anterior.
        """

        self.valores[( zero - k )%self.longitud] = v

    def append(self, v):
        """
        Añade un nuevo elemento a la lista.
        """
        self.zero = (self.zero + 1) % self.longitud
        self.valores[self.zero] = v
```

```
class SerieTemporal(Historial):
    """
    Extiende historial para llevar un control del instante en que se añadió un
    elemento al historial, asi como obtener mediante interpolacion el valor en
    un instante arbitrario. Para que esto ultimo sea posible es obvio que el
    valor debe de ser numérico.
    """

    def __init__(self, longitud, timeFunc=time.time):
        """
        timeFunc: funcion que cada vez que se la llama devuelve el tiempo en ese
        mismo momento. Puede ser el tiempo real o no. Por defecto si no se
        espefica nada consulta el reloj del sistema. Cada vez que se añade un
        elemento con append se llama a esta funcion para asignarle un valor de
        tiempo al valor numérico.
        """

        Historial.__init__(self, longitud)
        self.timeFunc = timeFunc

    def __setitem__(self, k, v):
        """
        Sin sentido, no hace nada. Si se quiere añadir un valor nuevo, ver
        append.
        """

        pass

    def __getitem__(self, k):
        """
        Obtiene el valor del k-esimo elemento.
        """

        it = Historial.__getitem__(self, k)
        if it != None:
            return it[1]
        else:
            return it

    def append(self, v, t=None):
        if t==None:
            Historial.append(self, (self.timeFunc(), v))
        else:
            Historial.append(self, (t, v))

    def time(self, k):
        """
        Obtiene el tiempo del k-esimo elemento (por el final).
        """

        it = Historial.__getitem__(self, k)
        if it != None:
            return it[0]
        else:
            return None

    def times(self):
        """
        Devuelve un iterador por la lista de tiempos.
        """

        class IterTimes:
```

```
            def __init__(self, st):
                self.c = 0
                self.st = st

            def __iter__(self):
                return self

            def next(self):
                t = self.st.time(self.c)
                if t == None or self.c == self.st.longitud:
                    raise StopIteration
                else:
                    self.c += 1
                    return t

        return IterTimes(self)

    def separa(self):
        t = []
        x = []
        for i in self.valores:
            if i == None:
                break
            t.append(i[0])
            x.append(i[1])
        return t, x

    def __call__(self, t):
        """
        Interpola linealmente el valor en el instante t
        """

        c = c1 = c2 = 0
        d = d1 = d2 = None
        while d1==None or d2==None or d<=d2:
            _t = self.time(c)
            if _t == None:
                break
            else:
                d = abs(t - _t)
            if d1 == None or d<d1:
                c1, d1 = c, d
            elif d2 == None or d<d2:
                c2, d2 = c, d
            c += 1
        t1, t2 = map(self.time, [c1, c2])
        v1, v2 = map(self.__getitem__, [c1, c2])
        # a lo mejor la lista solo contiene un elemento
        if t1 == t2:
            return v1
        else:
            # si contiene por lo menos dos, pues interpolacion lineal
            return (v2*(t - t1) - v1*(t - t2))/(t2 - t1)

class Reloj:
    """
    Permite controlar el tiempo mediante los metodos arrancar y parar.
    """

    def __init__(self, timeFunc=time.time):
        """
        timeFunc: funcion que cada vez que se la llama devuelve el tiempo en ese
```

```
        mismo momento. Puede ser el tiempo real o no. Por defecto si no se
        espefica nada consulta el reloj del sistema.
        """

        self.timeFunc = timeFunc
        self.t0 = None
        self.retraso = 0
        self.funciona = False

    def arranca(self):
        """
        Arranca el reloj, si ya estaba arrancado no pasa nada.
        """

        if self.t0 == None:
            self.t0 = self.timeFunc()
            self.funciona = True
        if not self.funciona:
            self.retraso += self.timeFunc() - self.parada
            self.funciona = True

    def para(self):
        """
        Para el reloj.
        """

        if self.funciona:
            self.parada = self.timeFunc()
            self.funciona = False

    def __call__(self):
        """
        Consulta el tiempo actual.
        """

        if self.funciona:
            t = self.timeFunc() - self.t0 - self.retraso
        elif self.t0!=None:
            t = self.parada - self.t0 - self.retraso
        else:
            t = None
        return t

################################################################################
#       TESTING                                                                #
################################################################################
if __name__=='__main__':
    st = SerieTemporal(10, time.time)
    st.append(1)
    time.sleep(0.1)
    st.append(2)
    for t in st.times():
        print "%f" % t
    t1, t2 = st.time(0), st.time(1)
    t = (t1 + t2)/2
    print st(t)
```

B.17. aero.py

```
# -*- coding: latin-1 -*-
"""
Define clases necesarias para el calculo de fuerzas y momentos aerodinámicos
para todo ángulo de ataque y resbalamiento.
"""

from numarray import *
from matematicas import *

class perfilA:
    """
    Modelo de perfil para todo rango de ángulos de ataque según el
    NASA Memorandum 84281
    """
    def __init__(self, AR, cLmax=None, a=None, cL0=0, de0=None, de1=None,
                 de2=None, De=0):
        """
        a:      pendiente de sustentación entre 0, pi/4
        cLmax: coeficiente de sustentación máximo
        AR:     alargamiento del ala (Aspect Ratio = b^2/S)
        De:     flecha del ala
        de0
        de1
        de2:    resistencia del perfil cD = de0 + de1*al + de1*al^2
        """

        self.AR = AR
        self.De = De

        # Pendiente de sustentacion, para ángulo de resbalamiento nulo
        # y sin flecha. Si no hay datos se estima
        if a == None:
            self.a = 2*pi/(1 + 2/self.AR)
        else:
            self.a = a
            # se supone que si se da la pendiente del ala ya se ha calculado
            # el efecto de la flecha
            self.De = 0

        # Si no se espefica se supone entrada en perdida en pi/4
        if cLmax == None:
            self.cLmax = self.a*pi/4.
        else:
            self.cLmax = cLmax

        # Angulo de ataque para sustentacion nula
        self.al0 = -cL0/self.a

        # Coeficientes de resistencia, midiendo el angulo de ataque a partir
        # de al0
        A = 0.8*pi*AR

        # Angulos pequeños
        if de0 == None:
            self.de0 = 0.009
        else:
            self.de0 = de0 - self.a**2/A*self.al0**2

        if de1 == None:
            self.de1 = 0.0
        else:
            self.de1 = de1 + 2*self.al0*self.a**2/A
```

```
        if de2 == None:
            self.de2 = 0.11
        else:
            self.de2 = de2 - self.a**2/A

        def la(al):
            return self.de0 + self.de1*al + self.de2*al**2

        def Dla(al):
            return self.de1 + 2*self.de2*al

        # Angulos grandes
        self.ep0 = -0.1254
        self.ep1 =  0.09415
        self.ep2 =  0.977525

        def ha(al):
            return self.ep0 + self.ep1*al + self.ep2*(sin(al))**2

        def Dha(al):
            return self.ep1 + 2*self.ep2*sin(al)*cos(al)

        # Interpolacion intermedia

        x1 = -0.60
        x2 = -0.55
        x3 = -0.35
        x4 =  0.35
        x5 =  0.55
        x6 =  0.60

        y5, y6 = map(ha, [x5, x6])
        y1, y2 = y6, y5
        y3, y4 = map(la, [x3, x4])

        D6 = Dha(x6)
        D1 = -D6
        D3, D4 = map(Dla, [x3, x4])

        self.ma1 = InterpoladorSpline(
                x = [ x1, x2, x3 ],
                y = [ y1, y2, y3 ],
                t0 = D1,
                t1 = D3 )

        self.ma2 = InterpoladorSpline(
                x = [ x4, x5, x6 ],
                y = [ y4, y5, y6 ],
                t0 = D4,
                t1 = D6 )

        self.la = la
        self.ha = ha


    def coefs(self, al, be):
        """
        Calcula (cL, cD) para el angulo de ataque al.
        """

        # Corregimos pendiente de sustentacion con el angulo
```

```
        # de resbalamiento y flecha del ala
        a = self.a*(cos(be + self.De))**2

        # Angulo de entrada en perdida. Hay que calcularlo aqui porque
        # varia con el angulo de resbalamiento
        als = self.cLmax/a

        # Angulo de ataque auxiliar: de als a al1 se produce el cambio
        # brusco de sustentacion y se interpola linealmente
        al1 = 1.2*als

        # Util para funciones definidas por trozos
        def pick(f, D):
            """
            Devuelvo el primer valor de D tal que f aplicada a su
            llave de verdadero.
            """
            for k,v in D.iteritems():
                if f(k):
                    return v
            return None

        # Se nos puede colar algun error numerico:
        if al<-pi:
            al = -pi
        elif al>pi:
            al = pi
        if be<-pi:
            be = -pi
        elif be>pi:
            be = pi

        ali = pick(
        lambda (x, y): x<=al and al<=y,
        {
            ( -pi,  -pi/2 ): lambda x: pi + x,
            ( -pi/2, 0    ): lambda x: -x,
            (  0,    pi/2 ): lambda x: x,
            (  pi/2, pi   ): lambda x: pi - x
        })(al)

        # Damos opcion a que la resistencia no sea simetrica
        aliD = pick(
        lambda (x, y): x<=al and al<=y,
        {
            ( -pi,   -pi/2 ): lambda x: pi + x,
            ( -pi/2, -0.60 ): lambda x: -x,
            ( -0.60,  pi/2 ): lambda x: x,
            (  pi/2,  pi   ): lambda x: pi - x
        })(al)

        cL0 = pick(
        lambda (x, y): x <= ali and ali<= y,
        {
            ( 0,   als  ): lambda x: a*x,
            ( als, al1  ): lambda x: self.cLmax - a*(x - als),
            ( al1, pi/2 ): lambda x:  0.8*self.cLmax*
                                      ( 1 - ((x - al1)/(pi/2 - al1))**2 )
        })(ali)

        cL = pick(
        lambda (x, y): x<=al and al<=y,
```

```
        {
            ( -pi,  -pi/2 ):  0.8*cL0,
            ( -pi/2, 0    ): -cL0,
            (  0,    pi/2 ):  cL0,
            (  pi/2, pi   ): -0.8*cL0
        })

        cDp = pick(
        lambda (x, y): x<=aliD and aliD<=y,
        {
            (-0.60, -0.35): self.ma1,
            (-0.35,  0.35): self.la,
            ( 0.35,  0.60): self.ma2,
            ( 0.60,  pi/2): self.ha
        })(aliD)

        cD = cDp + cL**2/(0.8*pi*self.AR)

        return (cL, cD)

class perfilB:
    """
    Modelo de perfil para todo rango de ángulos de ataque según el
    NACA TN 3241
    """

    def __init__(self, a, cLmax, cDmax=1.18, d0=0.09, d2=0.21):
        """
        a: Pendiente de sustentación
        cLmax: máximo coeficiente de sustentación
        cDmax: máximo coeficiente de resistencia
        d0: primer término del coeficiente de resistencia
        d2: segundo término del coeficiente de resistencia
        """

        self.a = a
        self.cLmax = cLmax
        self.d0 = d0
        self.d2 = d2
        self.cDmax = cDmax

    def coefs(self, al, be):
        cD = self.d0 + self.d2*(al)**2
        cL = self.a*sin(al)

        if abs(cL)>self.cLmax:
            cL = self.cLmax*sign(cL)

        if abs(cD)>self.cDmax:
            cD = self.cDmax*abs(sin(al))

        return cL, cD

class perfilC:
    """
    Modelo de perfil para todo rango de ángulos de ataque basado en
    tabla para todo ángulo de ataque y resbalamiento
    """
    def __init__(self, cL, cD):
        self.cL = cL
        self.cD = cD
```

```
    def coefs(self, al, be):
        return cL(al, be), cD(al, be)


class FuselajeA:
    def __init__(self, cDa90, cDb90, cm90, cl90, cn90,
            cDa, cDb, cL, cY, cl, cm, cn,
            al1, al2, be1, be2, unidades = 'rad'):
        """
        Valor de los coeficientes para ángulo de ataque 90°:
            cDa90, cm90
        Valor de los coeficientes para ángulo de resbalamiento 90°:
            cDb90, cl90, cn90

        Coeficientes para pequeños ángulos de ataque:
            cDa, cL, cm
        Definidios entre al1, al2

        Coeficientes para pequeños ángulos de resbalamiento:
            cY, cl, cn
        Definidos entre be1, be2

        """

        # Angulos de ataque pequeños
        def la(al):
            if unidades == 'deg':
                al = deg(al)

            return array([
                cDa(al), cL(al), cm(al) ])

        # Angulos de resbalamiento pequeños
        def lb(be):
            if unidades == 'deg':
                be = deg(be)
            return array([
                cDb(be), cY(be), cl(be), cn(be) ])

        # Angulos de ataque grandes
        def ha(al):
            Sal, Cal = sin(al), cos(al)

            cDa = cDa90*abs(Sal)*(Sal**2)
            cL  = cDa90*abs(Sal)*Sal*Cal
            cm  = cm90*abs(Sal)*Sal

            return array([
                cDa, cL, cm ])

        # Angulos de resbalamiento grandes
        def hb(be):
            Sbe, Cbe = sin(be), cos(be)

            cDb =  cDb90*abs(Sbe)*(Sbe**2)
            cY  = -cDb90*abs(Sbe)*Sbe*Cbe
            cl  =  cl90*abs(Sbe)*Sbe
            cn  =  cn90*abs(Sbe)*Sbe

            return array([
                cDb, cY, cl, cn ])
```

```
        # Para ángulos de ataque intermedios aproximamos
        # entre los dos casos
        if unidades == 'deg':
            al1, al2 = rad(al1), rad(al2)
            be1, be2 = rad(be1), rad(be2)

        xa0 = al1 - rad(50)
        xa1 = al1 - rad(45)
        xa2 = al1
        xa3 = al2
        xa4 = al2 + rad(45)
        xa5 = al2 + rad(50)

        ya0 = ha(xa0)
        ya1 = ha(xa1)
        ya2 = la(xa2)
        ya3 = la(xa3)
        ya4 = ha(xa4)
        ya5 = ha(xa5)

        da0 = deriv_1(ha, xa0)
        da2 = deriv_1(la, xa2)
        da3 = deriv_1(la, xa3)
        da5 = deriv_1(ha, xa5)

        ca0 = []
        for i in range(3):
            ca0.append(
                InterpoladorSpline(
                    x = [xa0, xa1, xa2],
                    y = [ya0[i], ya1[i], ya2[i]],
                    t0 = da0[i],
                    t1 = da2[i])
            )
        ca1 = []
        for i in range(3):
            ca1.append(
                InterpoladorSpline(
                    x = [xa3, xa4, xa5],
                    y = [ya3[i], ya4[i], ya5[i]],
                    t0 = da3[i],
                    t1 = da5[i])
            )

        def ma0(al):
            return array(
                    [ca0[0](al), ca0[1](al), ca0[2](al)],
                    type='Float64')

        def ma1(al):
            return array(
                    [ca1[0](al), ca1[1](al), ca1[2](al)],
                    type='Float64')



        # Interpola linealmente entre (x1, y1) y (x2, y2)
        def lin(x1, y1, x2, y2, x):
            return (y1*(x2 - x) + y2*(x - x1))/(x2 - x1)

        # Coeficientes en funcion del angulo de ataque,
        # valido entre -pi, pi
```

```
def ca(al):
    if xa2<=al and al<=xa3:
        return la(al)
    elif xa0<=al and al<=xa2:
        #return lin(xa1, ya1, xa2, ya2, al)
        return ma0(al)
    elif xa3<=al and al<=xa5:
        #return lin(xa3, ya3, xa4, ya4, al)
        return ma1(al)
    else: # al<x1 or al>x4
        return ha(al)

xb0 = be1 - rad(50)
xb1 = be1 - rad(45)
xb2 = be1
xb3 = be2
xb4 = be2 + rad(45)
xb5 = be2 + rad(50)

yb0 = hb(xb0)
yb1 = hb(xb1)
yb2 = lb(xb2)
yb3 = lb(xb3)
yb4 = hb(xb4)
yb5 = hb(xb5)

db0 = deriv_1(hb, xb0)
db2 = deriv_1(lb, xb2)
db3 = deriv_1(lb, xb3)
db5 = deriv_1(hb, xb5)

cb0 = []
for i in range(4):
    cb0.append(
        InterpoladorSpline(
            x = [xb0, xb1, xb2],
            y = [yb0[i], yb1[i], yb2[i]],
            t0 = db0[i],
            t1 = db2[i])
    )
cb1 = []
for i in range(4):
    cb1.append(
        InterpoladorSpline(
            x = [xb3, xb4, xb5],
            y = [yb3[i], yb4[i], yb5[i]],
            t0 = db3[i],
            t1 = db5[i])
    )

def mb0(be):
    return array(
            [cb0[0](be), cb0[1](be), cb0[2](be), cb0[3](be)],
            type='Float64')

def mb1(be):
    return array(
            [cb1[0](be), cb1[1](be), cb1[2](be), cb1[3](be)],
            type='Float64')
```

```
        # Coeficientes en funcion del angulo de resbalamiento,
        # valido entre -pi, pi
        def cb(be):
            if xb2<=be and be<=xb3:
                return lb(be)
            elif xb0<=be and be<=xb2:
                return mb0(be)
                #return lin(xb1, yb1, xb2, yb2, be)
            elif xb3<=be and be<=xb5:
                return mb1(be)
                #return lin(xb3, yb3, xb4, yb4, be)
            else:
                return hb(be)

        self.call_a = ca
        self.call_b = cb

    def coefs(self, al, be):
        ca = self.call_a(al)
        cb = self.call_b(be)

        cD = ca[0] + cb[0]
        cL = ca[1]
        cY = cb[1]
        cl = cb[2]
        cm = ca[2]
        cn = cb[3]
        return cD, cL, cY, cl, cm, cn

class FuselajeB:
    def __init__(self, f):
        """
        Wrapper alrededor de la funcion f, que devuelve el valor de
        los coeficientes aerodinamicos para todo angulo de ataque y
        resbalamiento:
        cD, cL, cY, cl, cm, cn = f(al, be)
        """

        self.f = f

    def coefs(self, al, be):
        return self.f(al, be)
```

B.18. input.py

```
# -*- coding: latin-1 -*-
"""
Clases para facilitar la entrada de datos en forma de tablas y listas de
valores.
"""

from numarray import *
from matematicas import *

class Tabla:
    def __init__(self, v):
        """
        Crea una tabla
```

```
        v: lista de elementos. La primera fila  contiene las coordenadas x de
           cada columna menos el primer elemento de la fila, cuyo valor se
           ignora, y el primer elemento de cada fila la coordenada de la fila.
           Por ejemplo:

           sum123 = Tabla([
            [ None,      1,  2,  3 ],

            [ 1,         2,  3,  4 ],
            [ 2,         3,  4,  5 ],
            [ 3,         4,  5,  6 ]])

        """

        self.interpolador =  InterpoladorLineal(
                [ v[0][1:], [ x[0] for x in v[1:] ] ],
                transpose(array([ x[1:] for x in v[1:]])))

    def __call__(self, x, y):
        """
        Calcula el valor interpolado en el punto x, y.
        """

        return self.interpolador([x,y])

class Lista:
    def __init__(self, *v):
        """
        Crea una lista a partir de un numero variable de arguementos
        con el formato, por ejemplo:
        senos = Lista(
                -pi,    0,
                -pi/2, -1,
                 0,     0,
                 pi/2,  1,
                 pi,    0
            )
        Devuelve una instancia que se puede llamar y que utiliza
        interpolación mediante splines cúbicas para calcular los
        valores.
        """

        x = []
        y = []
        for i in range(len(v)/2):
            x.append(v[2*i])
            y.append(v[2*i + 1])

        self.interpolador = InterpoladorSpline(x, y)

    def __call__(self, x):
        """
        Calcula el valor interpolado en el punto x.
        """

        return self.interpolador(x)
```

B.19. protocolo.py

```
# -*- coding: latin-1 -*-
"""
Protocol nativo tal como viene especificado en el codigo de FlightGear

Para mas detalles el protocolo viene descrito en:
    FlightGear/src/Newtork/net_ctrls.hxx
    FlightGear/src/Newtork/net_fdm.hxx

El protocolo de comandos aunque se encuentra en la clase HTTPClient definida e
implementada en:
    FlightGear/src/FDM/ExternalNet/ExternalNet.hxx
"""

import struct
import re

# formato de de llegada de los controles
# los valores de 64 bits, como los doubles requieren padding para
# ir en bytes multiplos de 8
ctrlFmt = ">I 4x 9d 2I I 16I 4x 12d 4I 4d 8I 24I I 8I 5I I 4x 5d I I 4d 3d 2d \
2d I 2I 100x"

# tamano de llegada de controles
ctrlBufSize = struct.calcsize(ctrlFmt)

# formato de salida del estado del FDM
fdmFmt = ">2I 3d 6f 11f 3f 2f 5I 36f I 4f 4I 9f I i f 10f"

# tamano de salida
fdmBufSize = struct.calcsize(fdmFmt)

# formato de linea de comandos
reCmd = re.compile('GET /(?P<prop>\w+-?\w*)\?value=(?P<value>.*) HTTP/1.0')

# hasta que haya generadores de python con parametros de llamada utilizamos esta
# clase

class SequenceGet:
    """
    Obtiene los siguientes n elementos de una secuencia
    """

    def __init__(self, tuple):
        self.t = tuple
        self.c = 0

    def reset(self):
        self.c = 0

    def __call__(self, n):
        if n == 1:
            r = self.t[self.c]
        else:
            r = self.t[self.c:self.c + n]
        self.c += n
        return r

def ctrlUnpack(str):
    """
    Convierte la cadena de llegada de controles en un diccionario
    """
```

```
    # desenpaqueta la cadena en una tupla
    t = struct.unpack(ctrlFmt, str)

    # transformamos la tupla en un diccionario
    get = SequenceGet(t)

    return {
        'version':              get(1),

        # control aerodinamico
        'aileron':              get(1),
        'elevator':             get(1),
        'rudder':               get(1),
        'aileron_trim':         get(1),
        'elevator_trim':        get(1),
        'rudder_trim':          get(1),
        'flaps':                get(1),
        'spoilers':             get(1),
        'speedbrake':           get(1),

        # fallos de controles aerodinamicos
        'flaps_power':          get(1),
        'flap_motor_ok':        get(1),

        # controles de motor
        'num_engines':          get(1),
        'master_bat':           get(4),
        'master_alt':           get(4),
        'magnetos':             get(4),
        'starter_power':        get(4),
        'throttle':             get(4),
        'mixture':              get(4),
        'condition':            get(4),
        'fuel_pump_power':      get(4),
        'prop_advance':         get(4),
        'feed_tank':            get(4),
        'reverse':              get(4),

        # fallos de motor
        'engine_ok':            get(4),
        'mag_left_ok':          get(4),
        'mag_right_ok':         get(4),
        'spark_plugs_ok':       get(4),
        'oil_press_status':     get(4),
        'fuel_pump_ok':         get(4),

        # combustible
        'num_tanks':            get(1),
        'fuel_selector':        get(8),
        'xfer_pump':            get(5),

        'cross_feed':           get(1),

        # frenos
        'brake_left':           get(1),
        'brake_right':          get(1),
        'copilot_brake_left':   get(1),
        'copilot_brake_right':  get(1),
        'brake_parking':        get(1),

        # tren de aterrizaje
        'gear_handle':          get(1),
```

```
        # interruptores
        'master_avionics':      get(1),

        # navegacion y comunicaciones
        'comm_1':               get(1),
        'comm_2':               get(1),
        'nav_1':                get(1),
        'nav_2':                get(1),

        # viento y turbulencia
        'wind_speed':           get(1),
        'wind_dir_deg':         get(1),
        'turbulence_norm':      get(1),

        # temperatura y presion
        'temp_c':               get(1),
        'press_inhg':           get(1),

        # otra informacion de entorno
        'hground':              get(1),
        'magvar':               get(1),

        # hielo
        'icing':                get(1),

        # control de la simulacion
        'speedup':              get(1),
        'freeze':               get(1)
    }


# propiedades del FDM en el orden correcto para enpaquetar
fdmProps = [
        'version',
        'padding',

        # Posiciones
        'longitude',
        'latitude',
        'altitude',
        'agl',
        'phi',
        'theta',
        'psi',
        'alpha',
        'beta',

        # Velocidades
        'phidot',
        'thetadot',
        'psidot',
        'vcas',
        'climb_rate',
        'v_north',
        'v_east',
        'v_down',
        'v_wind_body_north',
        'v_wind_body_east',
        'v_wind_body_down',

        # Aceleraciones
```

```
        'A_X_pilot',
        'A_Y_pilot',
        'A_Z_pilot',

        # Entrada en perdida
        'stall_warning',
        'slip_deg',

        # Estatus de motor
        'num_engines',
        'eng_state',
        'rpm',
        'fuel_flow',
        'fuel_px',
        'egt',
        'cht',
        'mp_osi',
        'tit',
        'oil_temp',
        'oil_px',

        # Consumibles
        'num_tanks',
        'fuel_quantity',

        # Estado del tren de aterrizaje
        'num_wheels',
        'wow',
        'gear_pos',
        'gear_steer',
        'gear_compression',

        # Entorno
        'cur_time',
        'warp',
        'visibility',

        # Posicion de las superficies de control
        'elevator',
        'elevator_trim_tab',
        'left_flap',
        'right_flap',
        'left_aileron',
        'right_aileron',
        'rudder',
        'nose_wheel',
        'speedbrake',
        'spoilers'
]


def fdmPack(dict):
    """
    Transforma el diccionario de propiedades en una cadena lista para
    transmitir
    """

    # Chequea si la variable es un diccionario o lista
    # FIX: deberiamos chequear si implementa un protocolo de contenedor
    def isContainer(con):
        if type(con) in map(type, [[],{}]):
            return True
```

```
        else:
            return False

    t = []
    for prop in fdmProps:
        val = dict[prop]
        if isContainer(val):
            t.extend(val)
        else:
            t.append(val)

    return struct.pack(fdmFmt, *t)
```

B.20. matematicas.py

```
# -*- coding: latin-1 -*-
"""
Funciones matematicas de todo tipo
"""

from math import asin, atan
import cmath
from numarray import *
import numarray.linear_algebra as la
import copy
import logging
import pylab

#############################################################################
#                UTILIDADES                                                 #
#############################################################################

def escalon(a=0.0, b=1.0, A = 0.0, B=1.0):
    """
    Devuelve una funcion escalon:
     B |    ........
       |    .       .
     A |....        ........
    ---------------------------------
            a       b

    Si b = None --> b = inf
    Si a = None --> a = inf

    Valores por defecto:
        a = A = 0.0
        b = B = 1.0
    """

    if a == None:
        def r(t):
            if t<=b:
                return B
            else:
                return A
    elif b == None:
        def r(t):
            if t>=a:
                return B
```

```
            else:
                return A
    else:
        def r(t):
            if t>=a and t<=b:
                return B
            else:
                return A
    return r

def sign(x):
    """
    Devuelve el signo de x (0 se interpreta con signo positivo)
        x>=0 --> +1
        x <0 --> -1
    """

    if x<0:
        return -1
    elif x>=0:
        return +1

def cross(a, b):
    """
    Calcula el producto vectorial de dos vectores.
    Los vectores pueden ser cualquier elemento iterable, pero el resultado
    es un array.
    """

    return array([
         a[1]*b[2] - a[2]*b[1],
         a[2]*b[0] - a[0]*b[2],
         a[0]*b[1] - a[1]*b[0]
    ])

def mod(x):
    """
    Devuelve el modulo del vector x
    """
    s = 0
    for c in x:
        s += c**2
    return sqrt(s)

def rlen(x):
    """
    Alias para range(len(x))
    """

    return range(len(x))

def reverse(x):
    """
    Ordena al reves una lista.
    """

    y = []
    for i in rlen(x):
        y.append(x[-1 - i])
    return y

def rad(x):
```

```
    """
    Convierte de grados a radianes
    """

    return x*pi/180.

def deg(x):
    """
    Convierte de radianes a grados
    """

    return x*180./pi


def localiza(x, L):
    """
    Dado un contenedor L ordenado de menor a mayor , y un valor x, devuelve el
    indice del array tal que L[i]<=x<=L[i+1] mediante el metodo de la
    biseccion. Si el elemento no esta en la lista devuelve alguno de los
    extremos, de forma que el interpolador lineal se convierte automaticamente
    en un extrapolador.
    """

    a=0
    b=len(L)-1
    while not ( (L[a+(b-a-1)/2]<=x) and (L[a+(b-a-1)/2+1]>=x) ):
        if x<L[a+(b-a-1)/2]:
            b=a+(b-a-1)/2
        else:
            a=a+(b-a-1)/2+1
        if a==len(L)-1:
            return len(L)-2
        elif b==0:
            return 0

    return a+(b-a-1)/2

def ang(x, y):
    """
    Calcula el angulo que forma el punto (x,y).
    Util porque distingue porque vale para cualquier cuadrante y porque trata
    sin problemas los casos +-pi/2.
    """

    if x == 0:
        if y>0:
            a = pi/2
        elif y<0:
            a = -pi/2
        else:
            a = 0.0
    else:
        a = atan(y/x)

    if  x<0:
        if y<0:
            a = a - pi
        else:
            a = a + pi
    if a>=pi:
        a -= 2*pi
    elif a<=-pi:
```

```
        a += 2*pi
    return a

##############################################################################
#                 SUAVIZADO                                                  #
##############################################################################
def smooth(x, cparada=None, cfijo=None):
    """
    Suaviza los datos x, hasta que se cumpla una condicion de parada.
    cparada(n, x): devuelve verdadero cuando se cumpla la condicion de parada,
    para n pasadas y datos x
    cfijo(i) : devuelve verdadero si el dato en el indice i no es modificable
    """

    if cfijo == None:
        def cfijo(i):
            if i<2 or i>len(x)-3:
                return True
            else:
                return False

    if cparada == None:
        def cparada(n, y):
            if n==1:
                return True
            else:
                return False

    # Arrays temporales
    y = x.copy()
    z = x.copy()
    # numero de pasadas
    n = 0

    # kernel gaussiano para 4 vecinos
    p = 2
    c = [0.1174, 0.1975, 0.2349, 0.1975, 0.1174]

    # Aplicamos sucesivas paradas mientras no se la condicion de parada
    while not cparada(n, y):
        for i in rlen(y):
            if not cfijo(i):
                z[i] = 0.0
                for k in range(-p, p +1):
                    z[i] += c[k]*y[i+k]
        y, z = z, y
        n += 1
    return y

##############################################################################
#                 CALCULO                                                    #
##############################################################################

def solveNewton(h, ix, Nmax, eps):
    """
    Resuelve por el metodo de Newton.
    En realidad aplica el metodo iterativo:
        Xn+1 = Xn + h(Xn)
    Mientras sea:
        |Xn+1 - Xn|>eps y n<Nmax
    Se supone que la funcion h(x) representa:
        h(x) = -f(x)/f'(x)
```

```
    para que el metodo calcule f(x)=0
    """

    n, error = 0, eps + 1
    x0=ix
    while error>eps:
        x1=x0 + h(x0)
        error=abs(x1-x0)
        x0=x1
        if n>Nmax:
            logging.error("Metodo de Newton no converge")
            break
        n += 1

    return x1

def solveNewtonMulti(f, ix, Nmax, eps):
    """
    Metodo de Newton para la ecuacion vectorial f(x) = 0.
    ix es el punto inicial
    Nmax maximo de iteraciones
    eps error numerico
    """
    n, error, x0 = 0, eps + 1, ix
    s = len(ix)
    J = array(shape=(s,s), type='Float64')
    while error>eps:
        for i in range(s):
                J[i,:] = dpart_1(f, x0, i, eps)
        x1 = x0 - dot(la.inverse(J), f(x0))
        error = mod(x1 - x0)
        x0 = x1
        if n>Nmax:
            logging.error("Metodo de Newton(multi) no converge")
            break
        n += 1
    return x1


def solveSecante(f, xa, xb, Nmax, eps):
    """
    Resuelve f=0 mediante el metodo de la secante.

    xa y xb son los puntos iniciales para calcular la secante, de forma ideal
    deberia ser signo(f(xa))!=signo(f(xb)) y  xa<xb
    """
    error = eps + 1
    n = 0
    xa = float(xa)
    xb = float(xb)
    ya = f(xa)
    yb = f(xb)
    while error>eps:
        # Si tenemos la mala suerte de una horizontal
        # probamos con otro punto
        if yb == ya:
            xb = (xb + xa)/2.
            yb = f(xb)
            continue

        # Interseccion con cero de la recta que une A y B
        x = (xa*yb - ya*xb)/(yb - ya)
```

```
        y = f(x)

        # A lo mejor ha habido suerte
        if x == xa or x == xb:
            return x

        # Decidimos que punto se marcha, si A o B
        # Supongamos que se marcha B
        if y!=ya:
            x1 = (xa*y - ya*x)/(y-ya)
        else:
            # x1 es infinito
            x1 == None
        # Supongamos que se marcha A
        if y!=yb:
            x2 = (x*yb - y*xb)/(yb-y)
        else:
            # x2 es infinito
            x2 = None

        # Alguno de los 2 puntos no es posible
        if x1 == None:
            xa = x
            ya = y
            continue

        if x2 == None:
            xb = x
            yb = y
            continue

        # Hay que decidir entre dos puntos
        if sign(ya)!=sign(yb):
            # La solucion esta entre xa y xb, nos aseguramos
            # de que el nuevo punto caiga en este intervalo
            # si x1==None o x2==None fallan los test de intervalo
            # como debe ser
            if xa<x1 and x1<xb:
                # 1 cae dentro
                if xa<x2 and x2<xb:
                    # Ambos caen dentro, nos quedamos con el mejor
                    y1 = f(x1)
                    y2 = f(x2)
                    if abs(y1)<abs(y2):
                        xb = x
                        yb = y
                    else:
                        xa = x
                        ya = y
                else:
                    # 1 cae dentro, pero 2 fuera
                    xb = x
                    yb = y
            else:
                # 1 cae fuera, al menos uno tiene que caer dentro
                # por ser distinto signo ya, yb, luego 2 cae dentro
                xa = x
                ya = y
        else:
            # La solucion no tiene por que estar dentro del intervalo
            # Nos limitamos a coger el mejor
            y1 = f(x1)
```

```
            y2 = f(x2)
            if abs(y1)<abs(y2):
                xb = x
                yb = y
            else:
                xa = x
                ya = y
            if xa>xb:
                xa, xb = xb, xa
                ya, yb = yb, ya

        error = abs(y)
        n += 1
        if n>Nmax:
            logging.error("Secante no converge")
            return None
    return x

def deriv_1(f, x, h=0.001):
    """
    Calcula la primera derivada de f en el punto x
    """

    return (f(x+h) - f(x-h))/(2*h)

def dpart_1(f, x, p, h=0.001):
    """
    Calcula la derivada parcial de f en el punto x, en la coordenada p
    """

    x1 = copy.deepcopy(x)
    x2 = copy.deepcopy(x)
    x1[p] -= h
    x2[p] += h
    return (f(x2) - f(x1))/(2*h)



################################################################################
#            AJUSTE POR MINIMOS CUADRADOS                                      #
################################################################################

def lsq_fit(x, y, f):
    """
    Calcula un ajuste lineal del tipo:
        a0*f[0](x) + a1*f[1](x) + ... + aM-1*f[M-1](x)
    para los N datos "y" definidos en las N posiciones "x".
    Devuelve los coeficientes "a" y la desviacion estimadas en cada punto.
    """

    M = len(f)
    b = array(shape=(M), type='Float64')
    c = array(shape=(M, M), type='Float64')

    for j in range(M):
        for k in range(M):
            c[k, j] = sum([dot(f[j](x[i]),f[k](x[i])) for i in rlen(x)])
        b[j] = sum([dot(y[i], f[j](x[i])) for i in rlen(x)])
    d = la.inverse(c)
    return dot(d, b), map(sqrt, diagonal(d))

def f_pol(n):
```

```
    """
    Devuelve una funcion polinomica de grado n
    """

    return (lambda x: x**n)

def f_cos(n):
    """
    Devuelve una funcion coseno de frecuencia n
    """

    return (lambda x: cos(n*x))

def f_sin(n):
    """
    Devuelve una funcion seno de frecuencia n
    """

    return (lambda x: sin(n*x))

def f_lin(c, f):
    """
    Devuelve la funcion c[i]*f[i]
    """

    def r(x):
        return sum([c[i]*f[i](x) for i in rlen(f)])

    return r

################################################################################
#            INTERPOLADORES                                                    #
################################################################################

class InterpoladorLineal:
    """
    Interpolador lineal para tablas multidimensionales
    """

    def __init__(self, x, y):
        # nos aseguramos de que los tipos son correctos
        if not hasattr(x[0], '__getitem__'):
            self.x = [x]
        else:
            self.x = x
        self.y = array(y)

        # la dimension de cada punto de la tabla
        self.dim_x = len(self.x)

    def __call__(self, ix):
        """
        Interpola el valor en el punto ix
        """

        if not  hasattr(ix, '__getitem__'):
            x = [ix]
        else:
            x = ix

        y0 = self.y
        for d in range(self.dim_x):
```

```
            i = localiza(x[d], self.x[d])
            x0, x1, x2 = x[d], self.x[d][i], self.x[d][i+1]
            y1 = ((x2 - x0)*y0[i] + (x0 -x1)*y0[i+1])/(x2-x1)
            y0 = y1
        return y1

class InterpoladorSpline:
    """
    Interpola mediante splines cubicas datos unidimensionale.
    """

    def __init__(self, x, y, tipo = 'Clamp', t0 = None, t1 = None):
        """
        x: vector con las coordenadas x
        y: vector con las coordenadas y
        t0: tangente en x[0],y[0]
        t1: tangente en x[n],y[n] (len(x) == len(y) == n + 1)
        """

        self.x = x
        self.y = y

        if t0 == None:
            t0 = (y[1] - y[0])/(x[1] - x[0])

        if t1 == None:
            t1 = (y[-1] - y[-2])/(x[-1] - x[-2])

        # Construimos el sistema de ecuaciones que nos determinan los
        # coeficientes de las splines
        n = len(x) - 1
        C = zeros( shape = (n+1, n+1), type='Float64' )
        b = array( shape =(n+1,), type='Float64')

        self.h = h = array( [x[i+1] - x[i] for i in range(n)], type='Float64')

        # Condiciones en los extremos
        if tipo == 'Clamp':
            C[0,:2] = [2, 1]
            b[0] = 6/(h[0]**2)*(y[1] - y[0] - t0*h[0])
            C[n,-2:] = [1,2]
            b[n] = 6/(h[n-1]**2)*(y[n-1] - y[n] + t1*h[n-1])
        else:
            C[0,0] = 1.0
            b[0] = 0.0
            C[n,n] = 1.0
            b[n] = 0.0

        # Grueso de la matriz
        for i in range(1,n):
            C[i][i-1:i+2] = [h[i-1], 2*(h[i-1] + h[i]), h[i]]
            b[i] = 6*((y[i+1] - y[i])/h[i] + (y[i-1] - y[i])/h[i-1])

        # Salvamos los coeficientes
        self.a = la.solve_linear_equations(C, b)

        # Para poder consultarlo
        self.t0 = t0
        self.t1 = t1

    def __call__(self, ix):
        """
```

```
        Calcula el valor interpolado en el punto ix.
        """

        x, y = self.x, self.y
        i = localiza(ix, x)
        dx0, dx1 = ix - x[i], x[i+1] -ix
        a0, a1 = self.a[i], self.a[i+1]
        h = self.h[i]

        return (
                (a0*dx1**3 + a1*dx0**3)/(6*h)
                + (y[i] - a0*h**2/6)*dx1/h +
                (y[i+1] - a1*h**2/6)*dx0/h
        )



##############################################################################
#            QUATERNIOS                                                      #
##############################################################################

def mul_quat(qA, qB):
    """
    Multiplica dos cuaternios
    """

    qAv = array(qA[1:])
    qBv = array(qB[1:])
    q0 = qA[0]*qB[0] - dot(qAv, qBv)
    q1, q2, q3 = qA[0]*qBv + qB[0]*qAv + cross(qAv, qBv)
    return q0, q1, q2, q3

class Quaternion:
    def __init__(self, q0=1., q1=0., q2=0., q3=0.):
        """
        Inicializa el quaternio como (q0, (q1,q2,q3))
        q0 es la parte escalar
        q1,q2,q3 la parte vectorial
        """

        self.q0, self.q1, self.q2, self.q3 = q0, q1, q2, q3

    def escalar(self):
        """
        Devuelve la parte escalar del quaternio.
        """

        return self.q0

    def vector(self):
        """
        Devuelve la parte vectorial del quaternio.
        """

        return array([self.q1, self.q2, self.q3], type='Float64')

    @classmethod
    def rot(cls, ang, x, y, z):
        """
        Devuelve un quaternio de la rotacion ang por el vector
        (x,y,z)
```

```
        """

        q0 = cos(ang/2.)
        q1, q2, q3 = (
                sin(ang/2.)*array([x, y, z], type='Float64')/
                sqrt(x**2 + y**2 + z**2) )
        return Quaternion(q0, q1, q2, q3)

    @classmethod
    def euler(cls, ch, th, fi):
        """
        Devuelve el quaternio de rotacion equivalente a los angulos
        de euler dados
        """

        return Quaternion.rotacion(
                [ (ch, 0, 0, 1), (th, 0, 1, 0), (fi, 1, 0, 0) ])

    def conj(self):
        """
        Devuelve el quaternio conjugado
        """

        return Quaternion(self.q0, -self.q1, -self.q2, -self.q3)

    def toMatrix(self):
        """
        Devuelve la matriz equivalente de rotacion
        """

        q0, q1, q2, q3 = self.q0, self.q1, self.q2, self.q3
        M = array(shape = (3, 3), type='Float64')

        return array([
            [
              1. - 2.*(q2**2 + q3**2),
              2.*(-q0*q3 + q1*q2),
              2.*(q0*q2 + q1*q3)
            ],
            [
              2.*(q0*q3 + q1*q2),
              1. - 2.*(q1**2 + q3**2),
              2.*(-q0*q1 + q2*q3)
            ],
            [
              2.*(-q0*q2 + q1*q3),
              2.*(q0*q1 + q2*q3),
              1. - 2.*(q1**2 + q2**2)
            ]], type='Float64')

    def toEuler(self):
        """
        Devuelve los angulos de euler en la tupla (ch, th, fi).
        """

        q0, q1, q2, q3 = self.q0, self.q1, self.q2, self.q3

        # normalizamos
        m = sqrt(q0**2 + q1**2 + q2**2 + q3**2)
        q0 /= m
        q1 /= m
        q2 /= m
```

```
        q3 /= m

        Sth = -2*(q1*q3 - q0*q2)
        # Como -pi/2<=th<=pi/2 podemos hacer esto tranquilos
        # Para th = +-pi/2 es posible que por error numerico se Sth>1
        if Sth>1.0:
            Sth = 1.0
        th = asin(Sth)

        # si no hay singularidad (th != +-pi/2)
        if abs(Sth) != 1.0:
            ch = ang(1 - 2*(q2**2 + q3**2), 2*(q1*q2 + q0*q3))
            fi = ang(1 - 2*(q1**2 + q2**2), 2*(q0*q1 + q2*q3))

        # si no damos un valor cualquiera a ch, ya que hay varias posibilidades
        # para ch y th que nos dan los mismos ejes
        else:
            ch = 0
            fi = ang(q0*q2 + q1*q3, q1*q2 - q0*q3)

        return ch, th, fi

    def __add__(self, other):
        return Quaternion(
                self.q0 + other.q0,
                self.q1 + other.q1,
                self.q2 + other.q2,
                self.q3 + other.q3)

    def __mul__(self, other):
        if isinstance(other, Quaternion):
            q0 = self.q0*other.q0 - dot(self.vector(), other.vector())
            q1, q2, q3 = ( self.q0*other.vector() + other.q0*self.vector()
                    + cross(self.vector(), other.vector()) )
        else:
            q0, q1, q2, q3 = ( other*self.q0, other*self.q1,
                    other*self.q2, other*self.q3 )

        return Quaternion(q0, q1, q2, q3)

    def __rmul__(self, other):
        return self*other

    def __sub__(self, other):
        return Quaternion(
                self.q0 - other.q0,
                self.q1 - other.q1,
                self.q2 - other.q2,
                self.q3 - other.q3 )

    def __repr__(self):
        return "[%f, (%f, %f, %f)]" % (self.q0, self.q1, self.q2, self.q3)

    @classmethod
    def rotacion(cls, l_quat):
        """
        Devuelve el quaternio resultante de aplicar sucesivamente las
        rotaciones contenidas en l_quat. Cada rotacion es una tupla de formato:
            (angulo, x, y, z)
        """

        return reduce(lambda x,y: x*y,
```

```
                [Quaternion.rot(r[0], r[1], r[2], r[3])  for r in l_quat])

    def __iter__(self):
        return [self.q0, self.q1, self.q2, self.q3].__iter__()

################################################################################
#           POLINOMIOS                                                         #
################################################################################
def polyeval(p, x):
    """
    Evalua el polinomio en x
    """
    n = len(p) - 1
    r = p[n]
    for i in range(n-1, -1, -1):
        r = p[i] + r*x
    return r

def polydiv(p, a):
    """
    Divide el polinomio p por el monomio x-a
    """
    n = len(p) - 1
    q = array(shape=(n,), type='Complex64')

    r = p[n]
    for i in range(n-1, -1, -1):
        s = p[i]
        q[i] = r
        r = s + a*r
    return q

def psolve(p, eps = 0.001, x0 = 0., x1 = 1., x2 = 2., Nmax=100):
    """
    Encuentra una raiz mediante el metodo de Muller
    """

    P0 = polyeval(p, x0)
    P1 = polyeval(p, x1)
    P2 = polyeval(p, x2)

    error = eps + 1.
    n = 0
    while error>eps:
        if n>Nmax:
            print "polysolve: n=%d>Nmax=%d" % (n, Nmax)
            return None

        q = (x0 - x1)/(x1 - x2)
        A = q*P0 - q*(1 + q)*P1 + q*q*P2
        B = (2*q + 1)*P0 - (1+q)*(1+q)*P1 + q*q*P2
        C = (1+q)*P0
        # Si tenemos suerte es C = 0, en cuyo caso
        # interrumpimos YA, porque entonces es E = 0
        # y dividiriamos por cero
        if C == 0:
            return x0

        D = cmath.sqrt(B*B  - 4*A*C)
        E1 = B - D
        E2 = B + D
        if abs(E1)>abs(E2):
```

```
            E = E1
        else:
            E = E2
        x = x0 - (x0 - x1)*2*C/E

        error = abs(x-x0)
        x0, x1, x2 = x, x0, x1
        P0, P1, P2 = polyeval(p, x0), P0, P1

    return x0

def polysolve(p, eps = 0.001, x0 = 0., x1 = 1., x2 = 2., Nmax=100):
    """
    Encuentra todas las raices del polinomio, aplicando el metodo
    de Muller y dividiendo por la raiz resultante
    """

    r = []
    q = p
    while len(q)>1:
        s = psolve(q, eps, x0, x1, x2, Nmax)
        r.append(s)
        q = polydiv(q, s)
    return r

def estabilidad(a, f, xa=-4, ya=-4, xb=0.5, yb=4, Nx=100, Ny=100):
    """
    Dibuja la region de estabilidad.
    El polinomio de estabilidad tiene la forma:

    q(r) = sum( (a[j] - w*f[j](w))*r^(p-j), j, 0, p)
    donde p es el numero de pasos del esquema

    xa, ya, xb, yb forma el rectangulo donde se realiza el calculo.
    Nx, Ny son la resolucion que se utiliza para la malla.
    """

    def pol(x, y):
        """
        Calcula el polinomio caracterÃstico en un punto generico x, y.
        """

        # Si un coeficiente no esta presente, vale cero
        la = len(a)
        lf = len(f)
        # lt - 1 = numero de pasos-->lt coeficientes
        # generalmente lf>la
        lt = max(la, lf)
        def geta(i):
            if i>=la:
                return 0.0
            else:
                return a[i]

        def getf(i):
            if i>=lf:
                return lambda x: 0.0 + 0.0j
            else:
                return f[i]

        # Punto del plano complejo
        w = x + y*1j
```

```
        # Lista con los coeficientes
        p = zeros(shape=(lt,), type='Complex64')

        # Finalmente construimos el polinomio
        for j in range(lt):
            p[lt - 1 - j] = geta(j) - w*getf(j)(w)

        return p

    Z = array(shape=(Nx, Ny), type='Float64')
    dx = float(xb - xa)/(Nx-1)
    dy = float(yb - ya)/(Ny-1)
    X = arange(xa, xb + dx, dx)
    Y = arange(ya, yb + dy, dy)

    mini = Nx - 1
    maxi = 0
    minj = Ny - 1
    maxj = 0
    for i in range(0, Nx):
        for j in range(0, Ny):
            p = pol(X[i], Y[j])
            # Los ceros del polinomio
            z = polysolve(p)
            # Nos quedamos con la raiz de mayor modulo
            r = max(map(abs, z))
            if r<=1:
                if i<mini:
                    mini = i
                if i>maxi:
                    maxi = i
                if j<minj:
                    minj = j
                if j>maxj:
                    maxj = j

            Z[j, i] = r

    # Damos un margen
    mini -= 3
    maxi += 3
    minj -= 3
    maxj += 3
    # Evitamos salirnos de limites
    if mini<0:
        mini = 0
    if maxi>Nx - 1:
        maxi = Nx - 1
    if minj<0:
        minj = 0
    if maxj>Ny - 1:
        maxj = Ny - 1

    # Por algun motivo solo funciona con copias
    X0 = X[mini:maxi].copy()
    Y0 = Y[minj:maxj].copy()
    Z0 = Z[minj:maxj, mini:maxi].copy()

    pylab.contour(X0, Y0, Z0, arange(0, 1.05, 0.05), colors='k')
    pylab.show()

    return X, Y, Z
```

```
if __name__=="__main__":
    def f(x):
        A = array([1, 0, 0], type='Float64')
        B = array([
            [1, 1, 1],
            [1, 1, 0],
            [1, 0, 0]], type='Float64')
        return A + dot(B, x)
    def g(x):
        a, b = x
        return array([a + b - 2, (a - 1)*(b + 2)], type='Float64')

    print solveNewtonMulti(f, [0, 0, 0], 100, 1e-5)
    print solveNewtonMulti(g, [0, 0], 100, 1e-5)
```

B.21. geodesia.py

```
# -*- coding: latin-1 -*-
"""
Definicion del Geoide y utilidades de conversion de coordenadas
"""

from numarray import *
from math import *
from matematicas import *

class Geoide:
    """
    Contiene la definicion del elipsoide y conversion de coordenadas
    """
    def __init__(self, a=6378137, invf=298.257223563, epsLat=1e-6):
        """
        Inicializa por defecto el geoide segun el WGS84
            a: Semieje mayor
            invf: el inverso del achatamiento
            epsLat: error numerico en las rutinas de cambio de cordenadas
        """
        self.a = a
        self.f = 1/invf
        self.e2 = self.f*(2 - self.f)
        self.epsLat = epsLat

        # la ultima tangente de latitud calculada por cart2geod
        self.Tlat = None
        # la ultima inversa de la tangente de latitud
        self.iTlat = None

    def geod2cart(self, lat, lon, alt):
        """
        Transforma de coordenadas geodesicas a cartesianas.
        Unidades en radianes y metros.
        """

        N = self.a/sqrt(1 - self.e2*sin(lat)**2)
        x = (N + alt)*cos(lat)*cos(lon)
        y = (N + alt)*cos(lat)*sin(lon)
```

```
        z = (N*(1 - self.e2) + alt)*sin(lat)

        return x, y, z

    def cart2geod(self, x, y, z):
        """
        Transforma de coordenadas cartesianas a geodesicas.
        Unidades en radianes y metros.
        """

        # error en latitud
        errLat = 100
        # minimo numero de iteraciones
        minN = 5
        # maximo numero de iteraciones
        maxN = 100
        # numero de iteraciones
        n = 0

        if (x**2 + y**2)<(self.a/2)**2:
            if self.Tlat != None:
                self.iTlat = 1./self.Tlat
                self.Tlat = None
            elif self.iTlat == None:
                self.iTlat = (1 - self.e2)*sqrt(x**2 + y**2)/z
            # Ya disponemos de una primera aproximacion para
            # la inversa de la tangente de la latitud
            iTlat1 = self.iTlat

            # Constantes del bucle
            c0 = sqrt(x**2 + y**2)/z
            c1 = self.e2*self.a/z
            c2 = (self.f-1)**2
            while n<minN or errLat>self.epsLat:
                iTlat2 = c0/(1. + c1/sqrt(iTlat1**2 + c2))
                errLat = abs(iTlat2 - iTlat1)
                iTlat1 = iTlat2
                n += 1

            lat = pi/2 - atan(iTlat2)

            lon = ang(x, y)
            Slat2 = 1./(1. + iTlat2**2)
            N = self.a/sqrt(1 - self.f*(2-self.f)*Slat2)
            alt = z/sqrt(Slat2) - N*(1. - self.e2)
        else:
            if self.iTlat != None:
                self.Tlat = 1./self.iTlat
                self.iTlat = None
            elif self.Tlat == None:
                self.Tlat = z/((1 - self.e2)*sqrt(x**2 + y**2))

            # Valor inicial para la tangente de la latitud
            Tlat1 = self.Tlat
            # constantes
            c0 = 1./sqrt(x**2 + y**2)
            c1 = self.e2*self.a
            c2 = (self.f-1)**2
            while n<minN or errLat>self.epsLat:
                Tlat2 = c0*(z + c1*Tlat1/sqrt(1 + c2*Tlat1**2))
                errLat = abs(Tlat2 - Tlat1)
                Tlat1 = Tlat2
```

```
            n += 1

        # latitud
        lat = atan(Tlat2)
        Slat1 = Slat = sin(lat)
        N2 = self.a/sqrt(1 - self.f*(2-self.f)*(Slat)**2)
        # longitud
        lon = ang(x, y)
        # altitud
        alt = sqrt(x**2 + y**2)/sqrt(1 - Slat1**2) - N2

    return lat, lon, alt

def ejesLocales(self, lat, lon, alt):
    """
    Coloca los ejes locales con origen en lat, lon, alt. Con el
    eje x apuntando al sur, el eje y al este y el z en la vertical.
    """

    Slat, Clat = sin(lat), cos(lat)
    Slon, Clon = sin(lon), cos(lon)

    # matriz ejes locales-cartesianos geocentricos
    self.Llc = array( [
        [  Slat*Clon,  Slat*Slon, -Clat ],
        [ -Slon,       Clon,       0    ],
        [  Clat*Clon,  Clat*Slon,  Slat ]] ,type='Float64')
    # matriz ejes cartesianos geocentricos-locales
    self.Lcl = transpose(self.Llc)
    # origen de los ejes locales
    self.Ol = array(self.geod2cart(lat, lon, alt), type='Float64')


def loc2cart(self, dx, dy, dz):
    """
    Calcula las coordenadas cartesianas a partir de los
    desplazamientos respecto a los ejes locales.
    """
    return self.Ol + dot(self.Lcl, [dx, dy, dz])

def cart2loc(self, x, y, z):
    """
    Calcula los desplazamientos en ejes locales
    a partir de las coordenadas cartesianas.
    """
    return dot(self.Llc, [x,y,z] - self.Ol)

def loc2geod(self, dx, dy, dz):
    """
    Calcula las coordenadas geodesicas a partir de los
    desplazamientos respecto a los ejes locales.
    """

    return self.cart2geod(*self.loc2cart(dx, dy, dz))

def geod2loc(self, lat, lon, alt):
    """
    Calcula las coordenadas locales a partir de las geodesicas
    """

    # Calculamos los cartesianos geocentricos
    cart = self.geod2cart(lat, lon, alt)
```

```
        # Pasamos a locales
        return dot(self.Llc, cart - self.Ol)

##############################################################################
#       TESTING                                                              #
##############################################################################
if __name__ == '__main__':
    lat0, lon0, alt0 = 0.0, 0.5, 1000
    wgs84 = Geoide()
    x, y, z = wgs84.geod2cart(lat0, lon0, alt0)
    lat1, lon1, alt1 = wgs84.cart2geod(x, y, z)
    error = abs(lat1 - lat0) + abs(lon1 - lon0) + abs(alt1 - alt0)
    print "error = %.12f" % error
```

B.22. Lynx.py

```
# -*- coding: latin-1 -*-
#!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!#
#           NO TOCAR                                #
from modelo import Modelo
from matematicas import *
from numarray import *
from aero import *
from input import *
from integrador import *

import logging

modelo = Modelo()
#!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!#

##############################################################################
# A PARTIR DE ESTE PUNTO INTRODUCIR LOS PARAMETROS CORRECTOS DEL HELICOPTERO   #
##############################################################################

##############################################################################
# MISCELANEO                                                                 #
##############################################################################
modelo.nombre = "Lynx"
modelo.integrador = ABM2()

##############################################################################
# DATOS MASICOS                                                              #
##############################################################################
modelo.M = 4313.7        # Masa del helicoptero
modelo.Ixx = 2767.1      # Momentos de inercia
modelo.Iyy = 13904.5     #      [  Ixx    0   -Ixz ]
modelo.Izz = 12208.8     # I = [   0    Iyy    0  ]
modelo.Ixz = 2034.8      #      [ -Ixz    0    Izz ]

##############################################################################
# PARAMETROS DEL ROTOR                                                       #
##############################################################################
modelo.rotor.gas = rad(4)    # Inclinacion del rotor, positivo hacia adelante
modelo.rotor.hR = 1.274      # Distancia vertical del CM al rotor, positivo
                             # hacia arriba
modelo.rotor.xcg = -0.0198   # Distancia horizontal, positivo hacia atras
modelo.rotor.h0 = 2.946      # Distancia del rotor al suelo
```

```
modelo.rotor.Nb = 4          # Numero de palas
modelo.rotor.R = 6.4         # Radio del rotor
modelo.rotor.c = 0.391       # Cuerda de las palas
modelo.rotor.Ibe = 678.14    # Momento de inercia de la pala
modelo.rotor.Kbe = 166352    # Constante elastica del muelle equivalente
modelo.rotor.tht = -0.14     # Torsion lineal de la pala

modelo.rotor.a0 = 6.0        # Pendiente de sustenacion
modelo.rotor.de0 = 0.009     # Coeficiente de resistencia parasita
modelo.rotor.de1 = 0.0
modelo.rotor.de2 = 37.983    # Coeficiente de resistencia inducida

##############################################################################
# PARAMETROS DEL ROTOR DE COLA                                               #
##############################################################################
modelo.rotor_cola.hT = 1.146         # Distancia vertical de la cola
modelo.rotor_cola.lT = 7.66          # Distancia horizontal de la cola
modelo.rotor_cola.K = 0.0            # Inclinacion respecto al plano vertical

modelo.rotor_cola.RT = 1.106         # Radio del rotor de cola
modelo.rotor_cola.cT = 0.18001       # Cuerda de la cola
modelo.rotor_cola.IbeT = 1.08926     # Momento de inercia de la pala de cola
modelo.rotor_cola.KbeT = 2511.24     # Muelle equivalente
modelo.rotor_cola.k3 = -1.           # Acoplamiento paso-batimiento = tan(de3)
modelo.rotor_cola.tht = 0.0          # Torsion lineal
modelo.rotor_cola.gT = 5.8           # Reduccion transmision cola-rotor principal

modelo.rotor_cola.a0T = 6.0          # Pendiente de sustentacion
modelo.rotor_cola.de0T = 0.008       # Coeficiente de resistencia parasita
modelo.rotor_cola.de1T = 0.0         #
modelo.rotor_cola.de2T = 5.334       # Coeficiente de resistencia inducida

##############################################################################
# PARAMETROS DEL MOTOR                                                       #
##############################################################################
modelo.motor.Omi = 35.63         # Velocidad de rotacion del rotor para par motor
                                 # cero
modelo.motor.Wto_0 = 746000      # Potencia de despegue de un motor a nivel del mar
modelo.motor.Wmc_0 = 664000      # Potencia maxima continua de un motor a nivel del mar
modelo.motor.x     = 0.85        # Variacion de la potencia disponible
modelo.motor.Wid   = 1000        # Potencia consumida para motor sin carga

modelo.motor.Irt = [             # Inercias referidas al rotor del sistema
                                 # de la transmision
                3085.069,        # Ningun motor activo
                3344.716,        # 1 motor activo
                3344.716]        # 2 motores activo

# Primera constante de tiempos
modelo.motor.tae1 = 0.1
# Segunda constante de tiempos
modelo.motor.a2 = 0.1
modelo.motor.b2 = 0.1
modelo.motor.c2 = 0.1

# Tercera constante de tiempos
modelo.motor.a3 = 0.1
modelo.motor.b3 = 0.1
modelo.motor.c3 = 0.1

modelo.motor.K3 = 65e3           # "Rigidez" del sistema motor
```

```
modelo.motor.n = 2              # numero de motores
modelo.motor.P  = 0.1           # Perdidas de transmision conjuntas de rotor
                                # y cola

################################################################################
# PARAMETROS DEL FUSELAJE                                                      #
################################################################################
modelo.fuselaje.Ss = 32.        # Superficie de adimensionalizacion para
                                # fuerzas/momentos laterales
modelo.fuselaje.Sp = 24.        # Superficie de adimensionalizacion para
                                # fuerzas/momentos longitudinales
modelo.fuselaje.lf = 12.        # Longitud de adimensionalizacion
modelo.fuselaje.xca = 0.139     # Posicion del centro aerodinamico respecto al
                                # Centro de Masas en ejes cuerpo
modelo.fuselaje.zca = 0.190
                                # Distancia tren de aterrizaje-centro de masas,
                                # positivo hacia abajo
modelo.fuselaje.ht = 0.762

modelo.fuselaje.aero = FuselajeA(  # Modelo aerodinamico de fuselaje
    unidades = 'deg',       # Unidades:  grados:   deg
                            #            radianes: rad
    cDa90 = 0.3480,         # Coeficiente de resistencia para al = 90Â°
    cDb90 = 0.4784,         # Coeficiente de resistencia para be = 90Â°
    cm90 =  0.03,           # Coeficiente de cabeceo para al = 90Â°
    cl90 =  0.01,           # Coeficiente de balance para be = 90Â°
    cn90 =  0.1,            # Coeficiente de guinada para be = 90Â°

    # cDa, cL, cm definidos entre al1, al2
    al1 = -21,
    al2 =  21,
    # cDb, cY, cl, cn definidos entre be1, be2
    be1 = -21,
    be2 =  21,

    # Coeficientes para angulos pequenos
    # Coeficiente de resistencia en funcion de angulo de ataque
    cDa = Lista(
        -21, 0.06661,
        -18, 0.06010,
        -15, 0.05531,
        -12, 0.04991,
         -9, 0.04707,
         -6, 0.04561,
         -3, 0.04421,
          0, 0.04233,
          3, 0.04245,
          6, 0.04372,
          9, 0.04486,
         12, 0.04685,
         15, 0.04926,
         18, 0.05170,
         21, 0.05613
    ),

    # Coeficiente de resistencia en funcion de angulo de resbalamiento
    cDb = Lista(
        -21, 0.05532,
        -18, 0.04096,
        -15, 0.02863,
        -12, 0.01842,
```

```
      -9, 0.01040,
      -6, 0.00464,
      -3, 0.00116,
       0, 0.00000,
       3, 0.00116,
       6, 0.00464,
       9, 0.01040,
      12, 0.01842,
      15, 0.02863,
      18, 0.04096,
      21, 0.05532
),

# Coeficiente de cabeceo en funcion de angulo de ataque
cm = Lista(
     -21, -0.02415,
     -18, -0.02210,
     -15, -0.02015,
     -12, -0.01803,
      -9, -0.01588,
      -6, -0.01351,
      -3, -0.01074,
       0, -0.00747,
       3, -0.00421,
       6, -0.00085,
       9,  0.00237,
      12,  0.00573,
      15,  0.00875,
      18,  0.01197,
      21,  0.01433
),

# Coeficiente de sustentacion en funcion de angulo de ataque
cL = Lista(
     -21, -0.02415,
     -18, -0.02210,
     -15, -0.02015,
     -12, -0.01803,
      -9, -0.01588,
      -6, -0.01351,
      -3, -0.01074,
       0, -0.00747,
       3, -0.00421,
       6, -0.00085,
       9,  0.00237,
      12,  0.00573,
      15,  0.00875,
      18,  0.01197,
      21,  0.01433
),

# Coeficiente de fuerza lateral en funcion de angulo de resbalamiento
cY = Lista(
     -21,  0.17547,
     -18,  0.15138,
     -15,  0.12677,
     -12,  0.10178,
      -9,  0.07653,
      -6,  0.05110,
      -3,  0.02557,
       0,  0.00000,
       3, -0.02557,
```

```
          6, -0.05110,
          9, -0.07653,
         12, -0.10178,
         15, -0.12677,
         18, -0.15138,
         21, -0.17547
    ),

    # Coeficiente de balance en funcion de angulo de resbalamiento
    cl = Lista(
        -21, -0.00697,
        -18, -0.00472,
        -15, -0.00270,
        -12, -0.00097,
         -9,  0.00000,
         -6,  0.00000,
         -3,  0.00000,
          0,  0.00000,
          3,  0.00000,
          6,  0.00000,
          9,  0.00000,
         12,  0.00097,
         15,  0.00270,
         18,  0.00472,
         21,  0.00697
    ),

    # Coeficiente de guinada en funcion de angulo de resbalamiento
    cn = Lista(
        -21, -0.01704,
        -18, -0.01461,
        -15, -0.01217,
        -12, -0.00974,
         -9, -0.00730,
         -6, -0.00487,
         -3, -0.00243,
          0,  0.00000,
          3,  0.00243,
          6,  0.00487,
          9,  0.00730,
         12,  0.00974,
         15,  0.01217,
         18,  0.01461,
         21,  0.01704
    ),
)


################################################################################
# PARAMETROS DEL ESTABILIZADOR HORIZONTAL                                      #
################################################################################
# Distancia horizontal del estabilizador horizontal
modelo.estabilizador_horizontal.ltp = 7.66
# Distancia vertical del estabilizador horizontal
modelo.estabilizador_horizontal.htp = 1.146
# Area del estabilizador horizontal
modelo.estabilizador_horizontal.Stp = 1.197
# Angulo de ataque del estabilizador horizontal
modelo.estabilizador_horizontal.altp0 = rad(-1.0)
# Modelo de estabilizador horizontal
modelo.estabilizador_horizontal.aero = perfilA(
        AR  =  2.7,         # Alargamiento
```

```
        a    =  2.3663,      # Pendiente de sustentacion
        cLmax = 0.6,         # Coeficiente de sustentacion maximo
        de0 =  1.065e-3,     # Coeficientes de resistencia:
        de1 = -8.4703e-2,    # de = de0 + de1*al + de2*al^2
        de2 =  1.46981
)

#############################################################################
# PARAMETROS DEL ESTABILIZADOR VERTICAL                                     #
#############################################################################
# Distancia horizontal del estabilizador vertical
modelo.estabilizador_vertical.lfn = 7.48
# Distancia vertical del estabilizador vertical
modelo.estabilizador_vertical.hfn = 0.57
# Area del estabilizador vertical
modelo.estabilizador_vertical.Sfn = 1.107
# Angulo de ataque del estabilizador vertical
modelo.estabilizador_vertical.befn0 = 0.0

modelo.estabilizador_vertical.aero = perfilA( # Modelo de estab. horizontal
        AR =   2.7,          # Alargamiento
        a =    2.5,          # Pendiente de sustentacion
        cL0 =  0.,           # Sustentacion para angulo de ataque nulo
        cLmax = 0.9,         # Coeficiente de sustentacion maximo
        de0 =  1.065e-3,     # Coeficientes de resistencia:
        de1 = -8.4703e-2,    # de = de0 + de1*al + de2*al^2
        de2 =  1.46981
)

#############################################################################
# CONTROLES                                                                 #
#############################################################################
# { th0  }              { et0  }
# { th1s } =  c0 + c1*  { et1s }
# { th1c }              { et1c }
# { th0T }              { etp  }

modelo.controles.c0 = array([
              0.059,
              0.000,
              0.000,
             -0.061,
            ], type='Float64')


modelo.controles.c1 = array([
            [ 0.300, 0.000,  0.000,  0.000 ],
            [ 0.000, 0.200,  0.000,  0.000 ],
            [ 0.000, 0.000,  0.200,  0.000 ],
            [ 0.442, 0.000,  0.000,  0.100 ],
            ], type='Float64')

# { et0  }       { et0p  }       { p }      { th - th_0}
# { et1s } = S0* { et1sp } + S1* { q } + S2*{ fi - fi_0}
# { et1c }       { et1cp }       { r }      { ch - ch_0}
# { etp  }       { etpp  }

modelo.controles.S0 = array([
    [ 1.0000, 0.0000, 0.0000, 0.0000 ],
    [ 0.0000, 0.0000, 0.0000, 0.0000 ],
    [ 0.0000, 0.0000, 0.0000, 0.0000 ],
    [ 0.0000, 0.0000, 0.0000, 1.0000 ]], type='Float64')
```

```
modelo.controles.S1 = array([
    [  0.0000,  0.0000, 0.0000 ],
    [  0.0000, -1.0000, 0.0000 ],
    [ +0.0000,  0.0000, 0.0000 ],
    [  0.0000,  0.0000, 3.0000 ]], type='Float64')

modelo.controles.S2 = array([
    [  0.0000,  0.0000, 0.0000 ],
    [ -2.5000,  0.0000, 0.0000 ],
    [  0.0000, +2.5000, 0.0000 ],
    [  0.0000,  0.0000, 0.0000 ]], type='Float64')

modelo.controles.th_0 =  lambda et1sp:  0.0603 + 1.4*et1sp
modelo.controles.fi_0 =  lambda et1cp: -0.0476 - 1.4*et1cp
modelo.controles.ch_0 =  0.0

# Limite para el control automÃ<tico
modelo.controles.L = 1.00

###############################################################################
# PARAMETROS DEL TREN                                                         #
###############################################################################
# x: posicion en ejes cuerpo del punto del tren
# n: coeficiente de rozamiento
# a, b: constante de rigidez y amortiguamiento para la fuerza normal
# c, d: constante de rigidez y amortiguamiento para la primera fuerza tangente
# e, f: constante de rigidez y amortiguamiento para la segunda fuerza tangente
modelo.tren.punto(
    x = [ 1.867, 1.01, 0.762],
    n = 1.0,
    a = 5e5, b = 1e5, c = 5e5, d = 1e5, e = 5e5, f = 1e5 )

modelo.tren.punto(
    x = [-1.147, 1.01, 0.762],
    n = 1.0,
    a = 5e5, b = 1e5, c = 5e5, d = 1e5, e = 5e5, f = 1e5 )

modelo.tren.punto(
    x = [ 1.867, -1.01, 0.762],
    n = 1.0,
    a = 5e5, b = 1e5, c = 5e5, d = 1e5, e = 5e5, f = 1e5 )

modelo.tren.punto(
    x = [-1.147, -1.01, 0.762],
    n = 1.0,
    a = 5e5, b = 1e5, c = 5e5, d = 1e5, e = 5e5, f = 1e5 )

modelo.tren.T0 = 1e-5

#!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!#
#   FIN DE LA CONFIGURACION                                                   #
#   El siguiente codigo no hace falta tocarlo, se utiliza solo                #
#   para funciones de debug                                                   #
#!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!#

###############################################################################
# TESTING                                                                     #
###############################################################################
if __name__ == "__main__":
    import time
    modelo.init()
```

```
# Propiedades menos vector de estado y controles
modelo.hG = 0
modelo.viento_vel = 0
modelo.viento_dir = 0
modelo.P = 103100
modelo.T = 283
modelo.motor.nmotores = 2

# Vector de estado
modelo.t = 0.0
modelo.x_i = modelo.y_i = 0.0
modelo.z_i = 1000
tr0 = modelo.trimado(10.0, 0.0, 0.0, 0.0, 1000, 1.225)

t0 = time.time()
T = 10.

# modelo.series se ocupa de los controles
s1 = modelo.series(T, 0, 0, 0, 0, 0.02,
        'u', 'v', 'w', 'p', 'q', 'r', 'la0','be0','be1c_w', 'be1s_w',
        'la1c_w', 'la1s_w', 'x_i', 'th', 'fi', 'ch')
t1 = time.time()
print "tiempo integrado = %f, tiempo real = %f, speedup=%f" %\
        (T, t1 - t0, T/(t1 -t0))

# De nuevo vector de estado
tr1 = modelo.trimado(10.0, 0.0, 0.0, 0.0, 1000., 1.225)
modelo.t = 0.0
modelo.x_i = modelo.y_i = 0.0
modelo.z_i = 1000

t0 = time.time()
T = 10.
s2 = modelo.series(T, 0, 0, 0, 0, 0.01,
        'u', 'v', 'w', 'p', 'q', 'r', 'la0','be0','be1c_w', 'be1s_w',
        'la1c_w', 'la1s_w', 'x_i', 'th', 'fi', 'ch')
t1 = time.time()

from pylab import *
plot(s1['t'], s1['u'])
plot(s2['t'], s2['u'])
xlabel('t')
ylabel('u')
show()

plot(s1['t'], s1['v'])
plot(s2['t'], s2['v'])
xlabel('t')
ylabel('v')
show()

plot(s1['t'], s1['w'])
plot(s2['t'], s2['w'])
xlabel('t')
ylabel('w')
show()

plot(s1['t'], s1['p'])
plot(s2['t'], s2['p'])
xlabel('t')
ylabel('p')
```

```
    show()

    plot(s1['t'], s1['q'])
    plot(s2['t'], s2['q'])
    xlabel('t')
    ylabel('q')
    show()

    plot(s1['t'], s1['r'])
    plot(s2['t'], s2['r'])
    xlabel('t')
    ylabel('r')
    show()

    plot(s1['t'], s1['th'])
    plot(s2['t'], s2['th'])
    xlabel('t')
    ylabel(r'$\theta$')
    show()

    plot(s1['t'], s1['fi'])
    plot(s2['t'], s2['fi'])
    xlabel('t')
    ylabel(r'$\phi$')
    show()
```

B.23. derivadas.py

```
#!/usr/bin/env python
# -*- coding: latin 1 -*-

# Las derivadas aqui calculadas siguen el convenio del Padfield:
# Las fuerzas estan divididas por la masa
#   X' = X/M
#   Y' = Y/M
#   Z' = Z/M
# Los momentos por las inercias:
#                  -1
#     { L' }   [  Ixx    0   -Ixz ]  { L }
#     { M' } = [   0    Iyy    0  ]  { M } =
#     { N' }   [ -Ixz    0    Izz ]  { N }
#              [ Izz/(IxxIzz - Ixz^2)     0     Ixz/(Ixx*Izz - Ixz^2) ] { L }
#              [           0          1/Iyy                          ]     { M }
#              [ Ixz/(IxxIzz - Ixz^2)     0     Ixx/(Ixz*Izz - Ixz^2) ] { N }


# Cargamos el modelo del helicoptero
import sys
import imp
nombre_modelo = sys.argv[1]
try:
    (file_modelo, path_modelo, desc_modelo) = imp.find_module(nombre_modelo)
except ImportError:
    sys.exit("Error: no se encontro el modelo")
modulo = imp.load_module(nombre_modelo, file_modelo, path_modelo, desc_modelo)
modelo = modulo.modelo
modelo.init()

from matematicas import *
```

```
from numarray import *
import numarray.linear_algebra as la

def extract(dic, lis):
    """
    Devuelve una lista con el valor que se encuentra en dic
    para cada elemento de lis
    """
    r = []
    for l in lis:
        r.append(dic[l])
    return r

class Derivadas:
    def __init__(self):
        # Variables a derivar
        self.a = ['X', 'Y', 'Z', 'L', 'M', 'N', 'Dq0',
                  'Dq1', 'Dq2', 'Dq3', 'D0m', 'DT2Q1', 'DTQ1']
        # Variables que derivan
        self.b = ['u', 'v', 'w', 'p', 'q', 'r', 'q0', 'q1', 'q2', 'q3',
                  '0m', 'DTQ1', 'Q1', 'th0', 'th1c', 'th1s', 'th0T']
        # Velocidades
        self.V = []
        # Las derivadas: la derivada X_u para todas las velocidades es una
        # lista que se encuentra en d['X'][u]
        self.d = {}
        for x in self.a:
            self.d[x] = {}
            for y in self.b:
                self.d[x][y] = []

    def asarray(self):
        """
        Devuelve el valor de las derivadas de estabilidad en forma de array:
            El primer indica indica la velocidad.
            El segundo indice la variable derivada.
            El tercer indice la variable que deriva.
        Por ejemplo:
            r = d_Modelo.asarray()
            print r[10, 2, 3]

        Como:
            d_Modelo.a[2] = 'Z'
            d_Modelo.b[3] = 'p'
        Suponiendo que fuese por ejemplo:
            d_Modelo.V[10] = 15

        Entonces mostraria por pantalla el valor de Z_p para la velocidad
        de 15 m/s
        """

        v = len(self.V)
        r = array(shape = (v, len(self.a), len(self.b)), type='Float64')
        for k in range(v):
            for i in range(len(self.a)):
                for j in range(len(self.b)):
                    r[k, i, j] = self.d[self.a[i]][self.b[j]][k]
        return r

    def append_b(self, n, l):
        """
        l es una lista que contiene las derivadas de todas las variables
```

```
        respecto a n, para una nueva velocidad. Esta funcion anade esta
        lista.
        """
        c = 0
        for a in self.a:
            self.d[a][n].append(l[c])
            c += 1

    def eigen(self, v):
        """
        Calcula los autovalores para la velocidad v
        """
        r = array(shape = (len(self.b)-4, len(self.a)), type='Float64')
        for i in range(len(self.a)):
            for j in range(len(self.b)):
                if self.b[j] not in ['th0', 'th1c', 'th1s', 'th0T']:
                    r[j, i] = self.d[self.a[i]][self.b[j]][v]
        return la.eigenvalues(r)


d_Modelo = Derivadas()

# Funcion a derivar
def f(x):
    modelo.trim = True
    modelo.controles.th0 =  x[16]
    modelo.controles.th1c = x[17]
    modelo.controles.th1s = x[18]
    modelo.controles.th0T = x[19]
    return modelo.paso(x[:16], 0)

# Velocidad en nudos, 1kn = 0.51444 m/s
V = concatenate([arange(0,2,0.2), arange(2, 162, 2.)])
for x in V:
    # Trimamos el modelo
    tr = modelo.trimado(x*0.51444, 0.0, 0.0, 0.0, 1000, 1.225)
    x0 = extract(tr,[
          "u", "v", "w", "p", "q", "r",
          "q0", "q1", "q2", "q3",
          "Om", "DTQ1", "Q1"
         ])
    x0.extend([0, 0.0, 1000.0])
    x0.extend([tr['th0'], tr['th1c'], tr['th1s'], tr['th0T']])

    # Evitamos que actuen los mandos
    modelo.trim = True

    # Empezamos a calcular derivadas
    print "V= ", x
    d_Modelo.V.append(x)
    for i in range(len(d_Modelo.b)):
        if d_Modelo.b[i] in ['th0', 'th1c', 'th1s', 'th0T']:
            k = i + 3
        else:
            k = i
        d_Modelo.append_b(d_Modelo.b[i], dpart_1(f, x0, k))

def pop_prox(x, l):
    """
    Extrae de la lista l el elemento que se encuentre mas proximo
    a x.
    """
```

```
    def prox(a, b):
        return abs(b-a)
    c = 0
    p = prox(x, l[0])
    for i in range(1, len(l)):
        if prox(x, l[i])<p:
            c = i
            p = prox(x, l[i])
    return l.pop(c)

eigen_x = [[] for i in range(13)]
eigen_y = [[] for i in range(13)]

e0 = d_Modelo.eigen(0).tolist()
for v in range(len(e0)):
    eigen_x[v].append(complex(e0[v]).real)
    eigen_y[v].append(complex(e0[v]).imag)

for x in range(1, len(V)):
    e = d_Modelo.eigen(x).tolist()
    for v in range(len(e0)):
        b = pop_prox(e0[v], e)
        eigen_x[v].append(complex(b).real)
        eigen_y[v].append(complex(b).imag)
    e0 = [ eigen_x[i][x] + eigen_y[i][x]*1j for i in range(len(eigen_x)) ]

from pylab import *
rc('figure', figsize=(4.72, 3.15))
rc('text', usetex=True)

c = ['b', 'g', 'r', 'c', 'm', 'y', 'k' ]
c *= 3
for i in (5, 0, 1, 2, 3, 4, 6, 7, 8, 9, 10, 11, 12):
   plot([eigen_x[i][0]], [eigen_y[i][0]], c[i] + 'o')
   plot([eigen_x[i][-1]], [eigen_y[i][-1]], c[i] + '^')
   plot(eigen_x[i], eigen_y[i], c[i] + '-')
show()

def latex(s):
    """
    Transforma la cadena s en una cadena latex
    """
    try:
        r = {
                'th0':  r'\theta_0',
                'th1c': r'\theta_{1c}',
                'th1s': r'\theta_{1s}',
                'th0T': r'\theta_{0_T}'
            }[s]
    except KeyError:
        r = s
    return r

for a in ['X', 'Y', 'Z', 'L', 'M', 'N']:
    for b in ['u', 'v', 'w', 'p', 'q', 'r', 'th0', 'th1s', 'th1c', 'th0T']:
        plot(d_Modelo.V, d_Modelo.d[a][b])
        ylabel(r'$%s_{%s}$' % (a, latex(b)))
        savefig('%s_%s_%s.eps' % (modelo.nombre, a,b), orientation='landscape')
        close()

        print a,b
```

Bibliografía

[1] Alastair K. Cooke and Eric W. H. Fitzpatrick. *Helicopter Test and Evaluation*. Blackwell Science, 2002.

[2] Joseph M. Cooke, Michael J. Zyda, David R. Pratt, and Robert B. McGhee. Npsnet: Flight simulation dynamic modeling using quaternions. *Presence*, 1(4):404–420, January 1994.

[3] Peter D.Talbot, Bruce E. Tinling, William A. Decker, and Robert T.N. Chen. A mathematical modelo of single main rotor helicopter for piloted simulation. Technical Report NASA TM 84281, NASA, September 1982.

[4] Juan A. Hernández. *Cálculo Numérico en Ecuaciones Diferenciales Ordinarias*. Aula Documental de Investigación, 2000.

[5] Kathryn B. Hilbert. A mathematical model of the uh-60 helicopter. Technical Report NASA TM 85890, NASA, April 1984.

[6] J.J. Howlett. Uh-60a black hawk engineering simulation program: Volume i - mathematical model. Technical Report NASA CONTRACTOR REPORT 166309, NASA, December 1981.

[7] I.C.Cheeseman and W.E. Bennett. The effect of the ground on a helicopter rotor in forward flight. Technical Report R&M 3021, Ministry of Supply, 1957.

[8] Wayne Johnson. *Helicopter Theory*. Princenton university press, 1980.

[9] Benton H. Lau, Alexander W. Louise, Nicholas Griffiths, and Costatinos P. Sotiriou. Performance and rotor loads measurements of the lynx xz170 helicopter with rectangular blades. Technical Report NASA-TM-104000, NASA, May 1993.

[10] Alfred Leick. *Gps Satellite Surveying*. John Wiley & Sons, 1990.

[11] NIMA. Department of defense world geodetic system 1984, its definition and relationships with local geodetic systems. Technical Report NIMA TR8350.2, National Imaginery and Mapping Agency, July 1997.

[12] Gareth D. Padfield. *Helicopter Flight Dynamics*. Blackwell Science Ltd, 1996.

[13] Alexander R. Perry. The flightgear flight simulator. Presentado en UseLinux 2004.

[14] David A. Peters and Ninh HaQuang. Dynamic inflow for practical applications. *Journal of the American Helicopter Society*, pages 64–68, October 1988.

[15] Dale M. Pitt and David A. Peters. Theoretical prediction of dynamic-inflow derivatives. *Vertica*, 5:21–34, 1981.

[16] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C*. Cambridge University Press, second edition, 2002.

[17] J.L. López Ruiz. *Helicópteros: Teoría y Diseño Conceptual*. Editorial E.T.S.I Aeronáuticos, 1993.