Universidad Politécnica de Madrid

MUSE

Ampliación de Matemáticas

Curso de programación

Autores:

Andújar Saltoratto, Gabriel
González Fariña, Álvaro
Ruiz Royo, Pablo

8 de febrero de 2019

Índice

0. Introducción

El objetivo principal de este informe es guiar al lector a través de todo el código realizado durante el curso, de tal forma que pueda entender los conceptos que hay detrás del mismo, así como analizar de manera crítica los resultados obtenidos. Dicho código ha sido realizado en dos lenguajes diferentes, **Fortran y MATLAB**; en aras de poder comparar las ventajas y desventajas entre un lenguaje compilado y uno interpretado. Para la realización de los proyectos en Fortran se utilizará el entorno de desarrollo **Visual Studio**, y el compilador **Intel Fortran**.

En primer lugar se van a explicar varios **conceptos clave** de la programación aprendidos durante el curso. Tras esto, se llega a la **metodología de programación**, donde se define la semántica que va a utilizarse a la hora de programar, así como errores que han tratado de evitarse. Estos dos puntos son fundamentales para entender el código realizado.

Entrando en materia, se han estudiado básicamente **3 problemas**. Por un lado se tiene el oscilador armónico, el cual se ha estudiado para tres esquemas numéricos distintos (Euler, Euler inverso y RK-4). Por otro lado se tiene el problema de los N cuerpos, analizando resultados para 2 y 3 cuerpos. Por último, se ha estudiado un problema desde el inicio, planteando una necesidad, y como es lógico una solución para dicha necesidad. Se trata del estudio de la dinámica de una nave espacial que se acerca a un agujero negro, considerando a la nave como un sólido rígido plano.

1. Conceptos clave

En esta sección se presentan una serie de conceptos que han ido aprendiéndose durante el curso, y que son necesarios a la hora de entender el porqué se realiza el código de una forma u otra:

1.1. Lenguajes de programación

Dentro de los lenguajes de programación es posible realizar la siguiente clasificación: **compilados** o **interpretados**. Tanto compiladores como interpretadores son programas que convierten el código escrito a lenguaje máquina (código binario).

La gran diferencia es que el compilado necesita de un paso adicional para realizar dicha conversión, la compilación; mientras que el interpretado realiza la conversión mientras se va ejecutando. Ejemplos de lenguajes compilados son **Fortran,C$_{++}$ o Java**. Ejemplos de lenguajes interpretados son **MATLAB, Python o JavaScript**.

Un lenguaje compilado está optimizado para el momento de la ejecución; mientras que un lenguaje interpretado está optimizado para facilitarle la vida al programador.

Los programas que han sido realizados por el grupo no tienen un grado de complejidad elevado, por lo que usar un tipo de lenguaje u otro, no supondrá mayor diferencia. Aun así, se van a realizar los programas de manera redundante: por un lado en el **lenguaje compilado Fortran**; y por otro en el **lenguaje interpretado MATLAB**. Según se vayan obteniendo los resultados se irán comentando las diferencias observadas entre ambos tipos.

1.2. Paradigmas de programación

Existen múltiples paradigmas de programación: imperativa, orientada a objetos, dirigida por eventos, funcional, lógica, con restricciones... Cada una de ellas ha sido propuesta por la comunidad de programadores y desarrolladores con el objetivo de resolver un problema en concreto. Todos los programas han sido realizados con el **paradigma funcional**, el más cercano a las matemáticas de todos. Con el objetivo de ver el asunto con perspectiva, se ha propuesto realizar uno de los programas con el **paradigma de programación orientada a objetos**, en el cual se encapsulan variables y funciones en elementos denominados objetos. Como se verá más adelante, el programa elegido para ello será el de los N-cuerpos.

1.3. Esquema de trabajo

El siguiente esquema representa la dinámica de trabajo que ha de seguirse en la resolución de problemas desde su concepción, hasta su finalización:

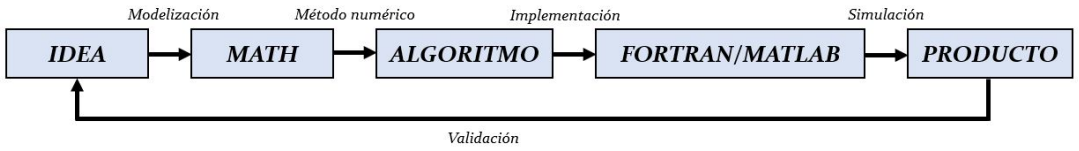

Figura 1: *Esquema de trabajo*

La modelización es el proceso que ha de llevarse a cabo para traducir una **idea al lenguaje de las matemáticas**. Esto se ha estudiado en numerosas asignaturas a lo largo de la carrera (mecánica clásica, mecánica de fluidos, termodinámica...). Para pasar del **lenguaje de las matemáticas a un algoritmo** que pueda resolver la máquina es necesario aplicarle un método numérico (Euler, Runge-Kutta...). Este paso también se estudió en profundidad en la carrera, por lo que el paso importante para este curso será el siguiente: la **implementación**. Por lo tanto, el grupo se centrará en realizar una implementación estructurada y profesional de los problemas que desean estudiarse en Fortran y MATLAB.

1.4. Extreme programming

Se trata de una metodología de desarrollo de la ingeniería de software formulada en 1999, basada en los siguientes valores: simplicidad, comunicación, retroalimentación, coraje y respeto.

Para que dichos valores se cumplan, es necesario seguir una serie de pautas; tales como realizar un desarrollo iterativo e incremental, pruebas unitarias continuas, refactorización del código y programación por parejas. Así pues, se recomienda que las tareas de desarrollo se lleven a cabo por dos personas en un mismo puesto, de tal manera que uno de ellos sea el **driver** (encargado de la escritura del código), y el otro el **reviewer** (encargado de corregir al driver si se equivoca). La mayor calidad del código escrito compensa con creces la posible pérdida de productividad inmediata. Por tanto, el grupo ha tomado este método para la realización del código, con el objetivo de comprobar su efectividad al finalizar el proyecto.

1.5. Test-Driven Development

El desarrollo guiado por pruebas es un proceso de desarrollo de software que se basa en la repetición continua de un ciclo, en el que se convierten los requisitos en test específicos que el programa creado debe pasar. El código realizado no tiene porqué estar escrito de manera perfecta, ya que únicamente se le exige que pase el test. Una vez conseguido este paso, se refactoriza el código, de tal manera que puedan cumplirse los valores de la programación extrema. Este ciclo es más conocido como **Red-Green-Refactoring**.

1.6. Métodos numéricos

Como bien se ha explicado anteriormente, para pasar del lenguaje matemático al algoritmo que pueda resolver la máquina es necesario implementar diversos métodos numéricos.

Una primera clasificación que puede realizarse es la de métodos:

- **Implícitos:** $G(U^{n+1}, U^n, U^{n-1}, ...)$, generalmente incondicionalmente estables
- **Explícitos:** $U^{n+1} = G(U^n, U^{n-1}, U^{n-2}, ...)$, condicionalmente estables

Otra clasificación es la de métodos:

- **Multietapa**, los más usados son los Runge Kutta. Entre paso y paso realiza evaluaciones intermedias. El Euler se considera unietapa
- **Multipaso**, los más usados son los Adam-Bashford o los Moulton. Realiza varios pasos anteriores y extrapola. El Euler se considera unipaso.

2. Metodología de programación

Para cumplir con los valores de la programación extrema es necesario cuidar la semántica, de tal manera que puedan diferenciarse las constantes, las variables, las funciones y los módulos de un simple vistazo:

- **Constantes**
- **Variables**
- **Funciones**
- **Módulos**

Es realmente importante usar el lenguaje matemático y variables locales (no datapool); así como evitar los nombres verbosos en la escritura.

3. Oscilador armónico

Este trata la resolución de la siguiente ecuación diferencial ordinaria:

$$\ddot{x} + x = 0 \tag{1}$$

Esta ecuación describe un movimiento armónico unidimensional ideal, como la que describiría un punto con masa sujeto con un muelle.

A continuación se describen tanto en Fortran como en MatLab la resolución de la ecuación con condiciones iniciales, es decir, un problema de Cauchy. Presentaremos distintos métodos de integración temporal, sus soluciones y sus errores respecto la solución analítica: Euler directo, Euler inverso, y Runge-Kutta de cuarto orden.

La condición inicial será posición inicial unidad y velocidad nula. La solución analítica en este caso será un simple coseno de t:

$$x = \cos t \tag{2}$$

Para adaptarlo al plano de la programación, se define un vector de estado U:

$$U = \left\{ \begin{array}{c} x \\ \dot{x} \end{array} \right\} \tag{3}$$

Y la ecuación (1) queda:

$$\frac{d\mathbf{U}}{dt} = \left[\begin{array}{cc} 0 & 1 \\ -1 & 0 \end{array} \right] \mathbf{U} \tag{4}$$

3.1. Euler

El método de Euler es el esquema temporal más simple que existe. Consiste en hallar una solución aproximada a partir de un punto inicial y la función derivada, que nos viene dada por la naturaleza del problema, e iterar para hallar la solución en un número discreto de puntos:

$$U^{n+1} = U^n + F(U^n) \cdot \Delta t \tag{5}$$

Siendo F la derivada del vector de estado en el instante n, y Δt el incremento temporal.

Por la naturaleza del método se induce un error de primer orden, proporcional al paso de tiempo. Por otro lado, también existirá un error debido al truncamiento y redondeo numérico.

La aplicación en Fortran es muy simple, con solo una linea de código se aplica la ecuación (4):

```
U = U + time_step * F(U)
```

A continuación se muestra el resultado de este método comparándolo con el analítico y su error (para Fortran):

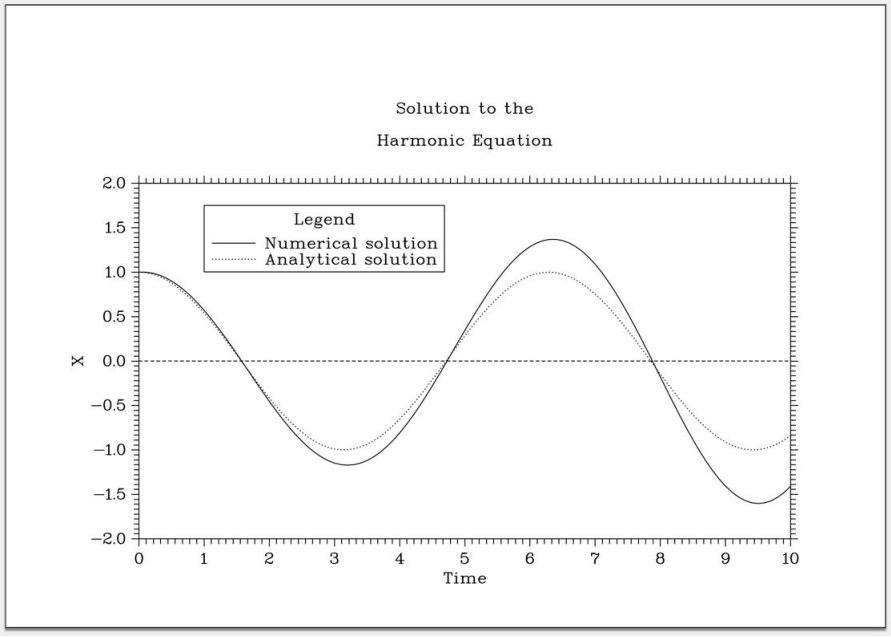

Figura 2: *Solución numérica (método de Euler) y analítica. Graficado con DISLIN.*

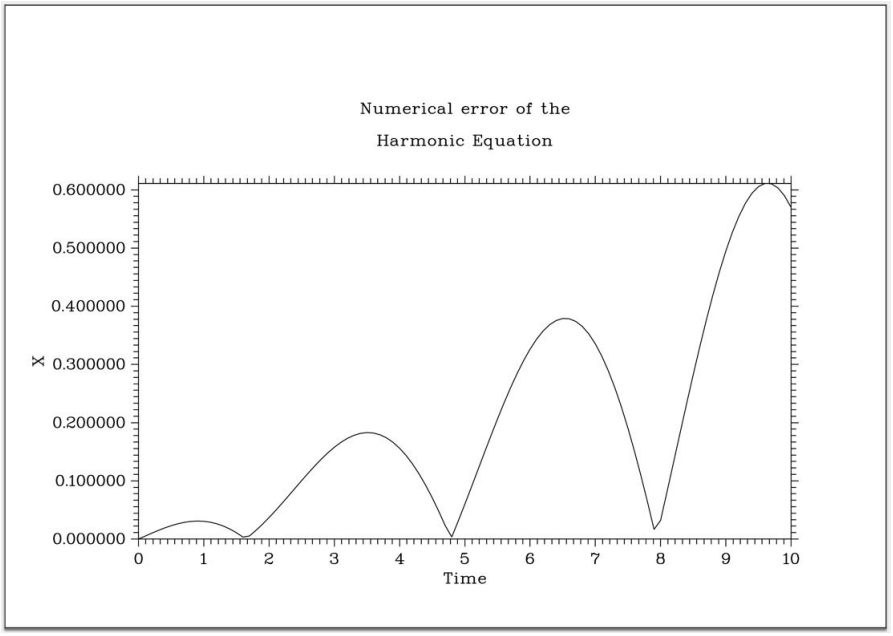

Figura 3: *Error cometido al utilizar Euler. Graficado con DISLIN.*

A continuación se muestra el resultado de este método comparándolo con el analítico y su error, pero habiéndo realizado el código en el lenguaje interpretado MATLAB:

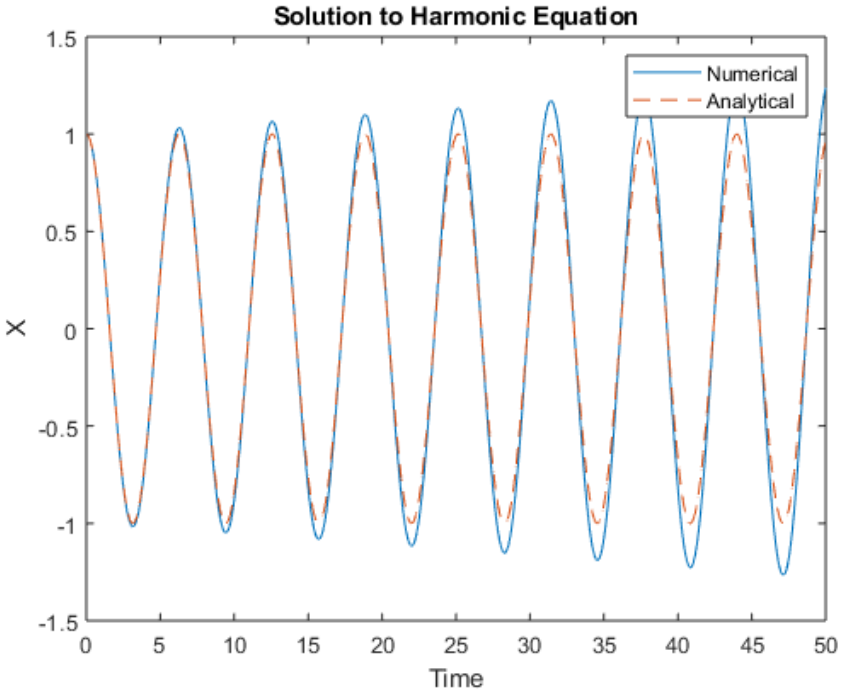

Figura 4: *Solución numérica (método de Euler) y analítica. Graficado con MATLAB.*

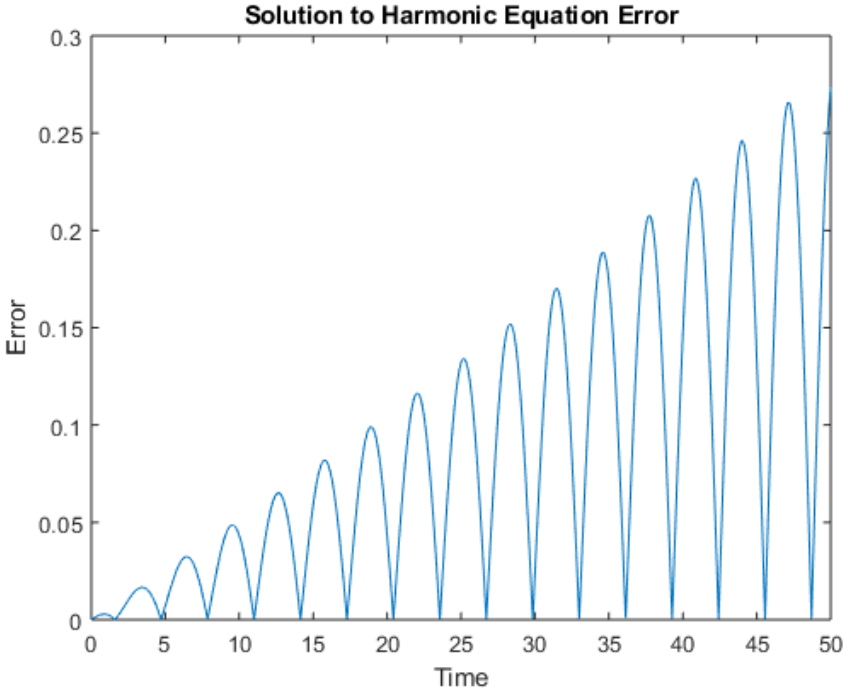

Figura 5: *Error cometido al utilizar Euler. Graficado con MATLAB.*

Aunque las representaciones están realizadas para un tiempo mayor en MATLAB, se observa que los resultados son idénticos en ambos programas. También puede apreciarse que el error va aumentando con el tiempo, debido a la acumulación.

3.2. Euler inverso

El esquema temporal Euler inverso se diferencia con el Euler directo en que la información de la derivada la toma en el instante que se quiere resolver:

$$U^{n+1} = U^n + F(U^{n+1})\Delta t \tag{6}$$

Para implantar la ecuación numéricamente, es necesario linealizar F en el caso de que no sea lineal, y despejar U^{n+1}:

$$F(U_0) = A \cdot U_0 \tag{7}$$

$$U^{n+1} = [I - \Delta t \cdot A]^{-1} U^n \tag{8}$$

En este problema la matriz queda:

$$A = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \tag{9}$$

Y los resultados de la simulación quedan (para Fortran):

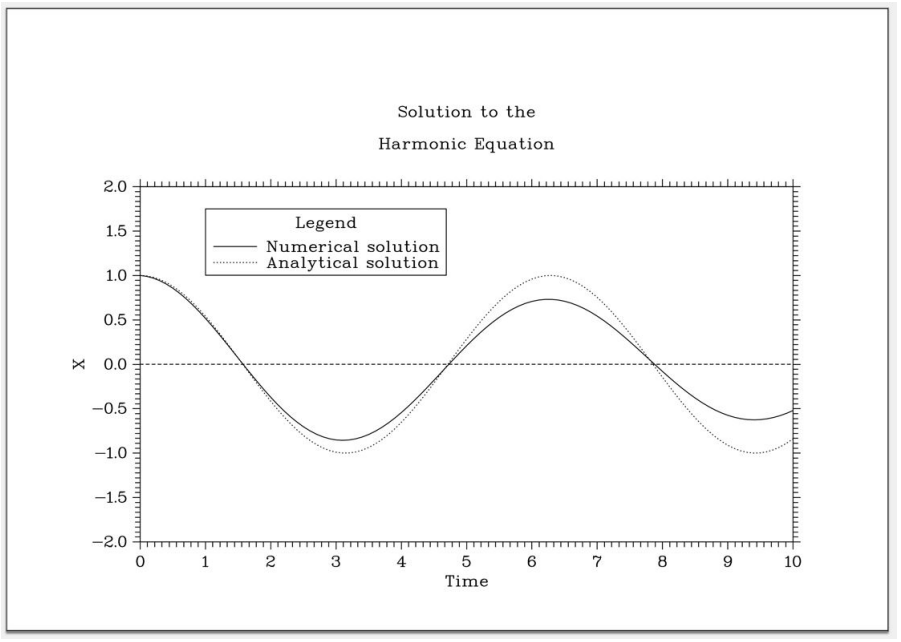

Figura 6: *Solución numérica (Euler inverso) y analítica . Graficado con DISLIN.*

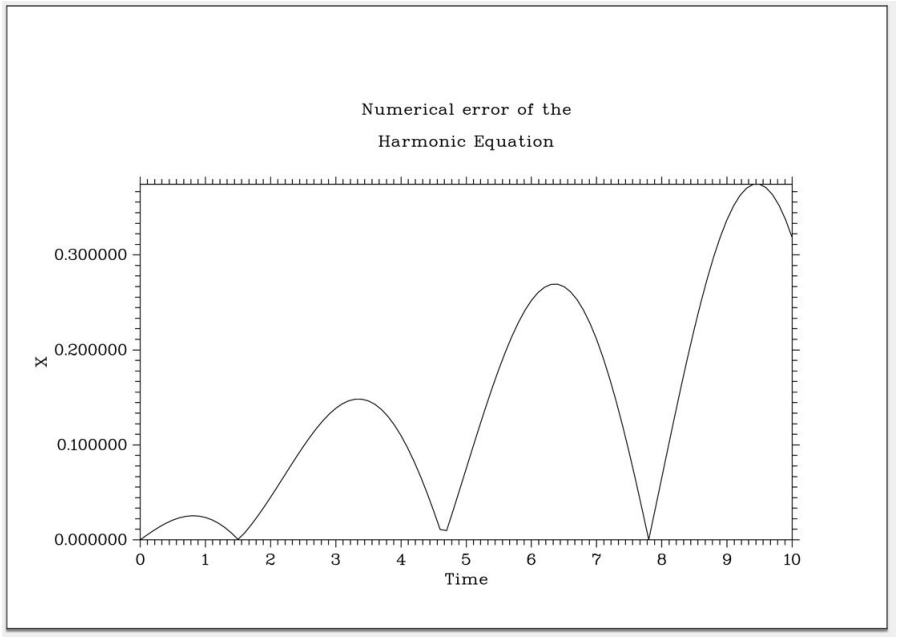

Figura 7: *Error cometido al utilizar Euler inverso. Graficado con DISLIN.*

Y los resultados de la simulación quedan (para Matlab):

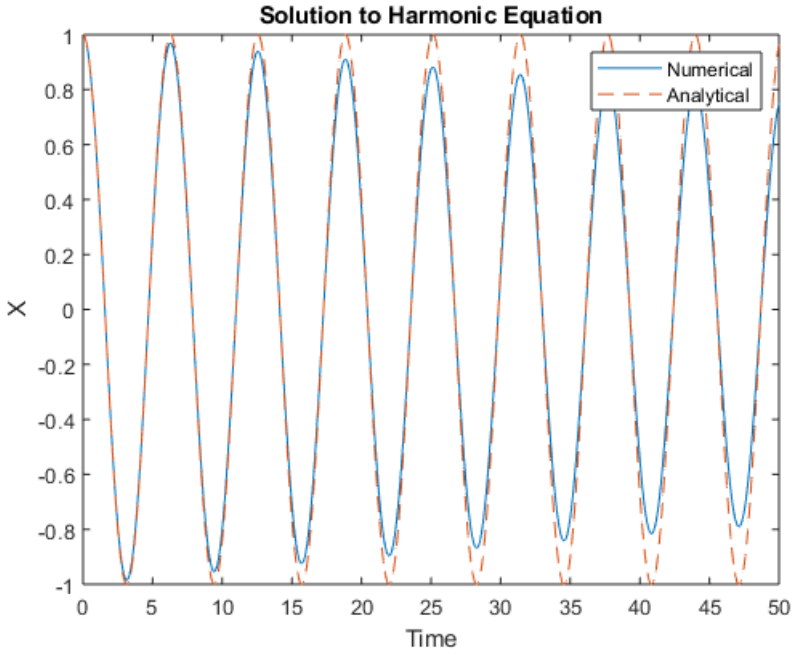

Figura 8: *Solución numérica (Euler inverso) y analítica . Graficado con MATLAB.*

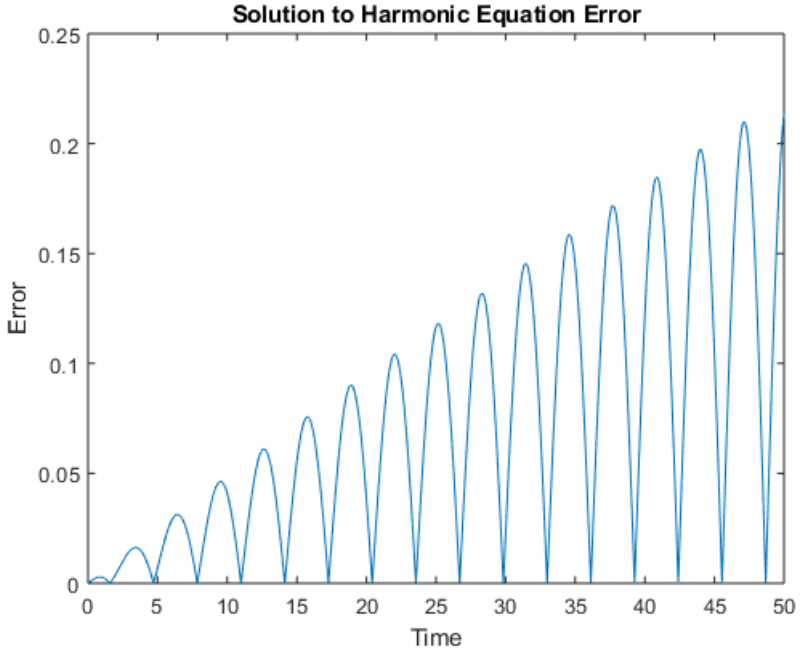

Figura 9: *Error cometido al utilizar Euler inverso. Graficado con MATLAB.*

Al igual que en el caso anterior, los resultados obtenidos con ambos programas son idénticos. Es de notable importancia la disminución del error cometido respecto al integrador anterior.

3.3. Runge-Kutta 4

La integración temporal de Runge-Kutta es un método multipaso que ofrece en general errores mucho más pequeños que los Euler, es varios órdenes de magnitud más preciso, pero también aumenta el coste computacional.

Se ha usado la librería NumericalHUB, la cual incluye útiles herramientas para la resolución numérica de problemas matemáticos. En concreto el módulo del problema de Cauchy *Cauchy_problem*, que da libertad en la elección de distintos esquemas temporales, entre ellos el RK de orden 4.

El Runge-Kutta de orden 4 presenta el siguiente aspecto:

$$U^{n+1} = U^n + \frac{1}{6}\Delta t(k_1 + 2k_2 + 2k_3 + k_4) \tag{10}$$

Siendo:

$$k_1 = F(t, U^n) \tag{11}$$

$$k_2 = F(t + \frac{1}{2}\Delta t, U^n + \frac{1}{2}k_1\Delta t) \tag{12}$$

$$k_3 = F(t + \frac{1}{2}\Delta t, U^n + \frac{1}{2}k_2\Delta t) \tag{13}$$

$$k_4 = F(t + \Delta t, U^n + k_3\Delta t) \tag{14}$$

A continuación se muestra el resultado del problema por medio de este método, y su error respecto al analítico, en el lenguaje de programación Fortran y MATLAB.

Como en los dos casos anteriores, los resultados obtenidos con ambos programas son idénticos. Se aprecia que utilizando un esquema de alto orden, como es el Runge-Kutta4, el error disminuye muchísimo, en torno a tres órdenes de magnitud.

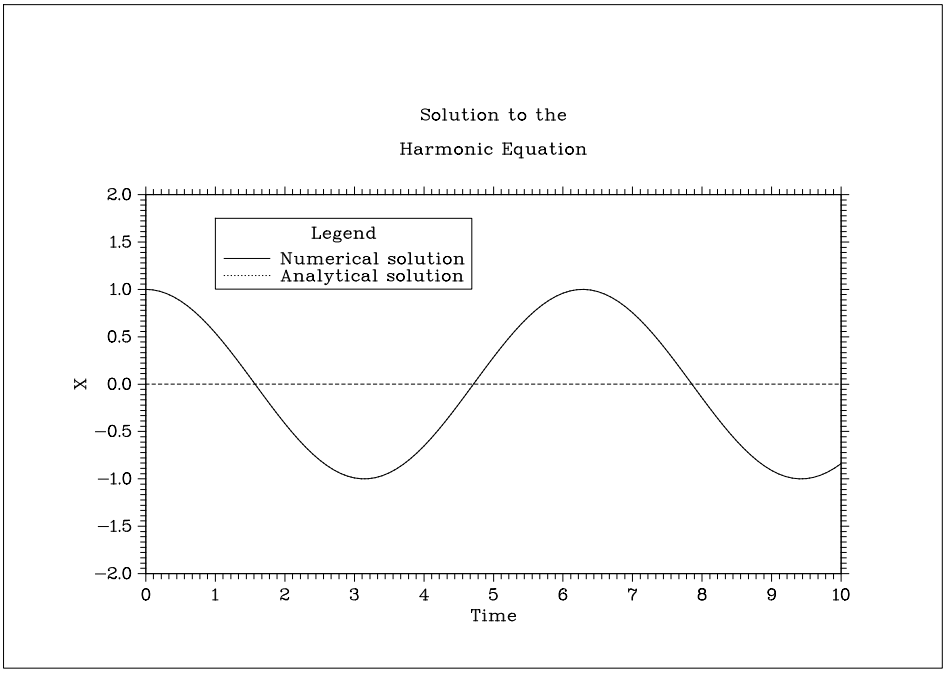

Figura 10: *Solución numérica (RK4) y analítica. Graficado con DISLIN.*

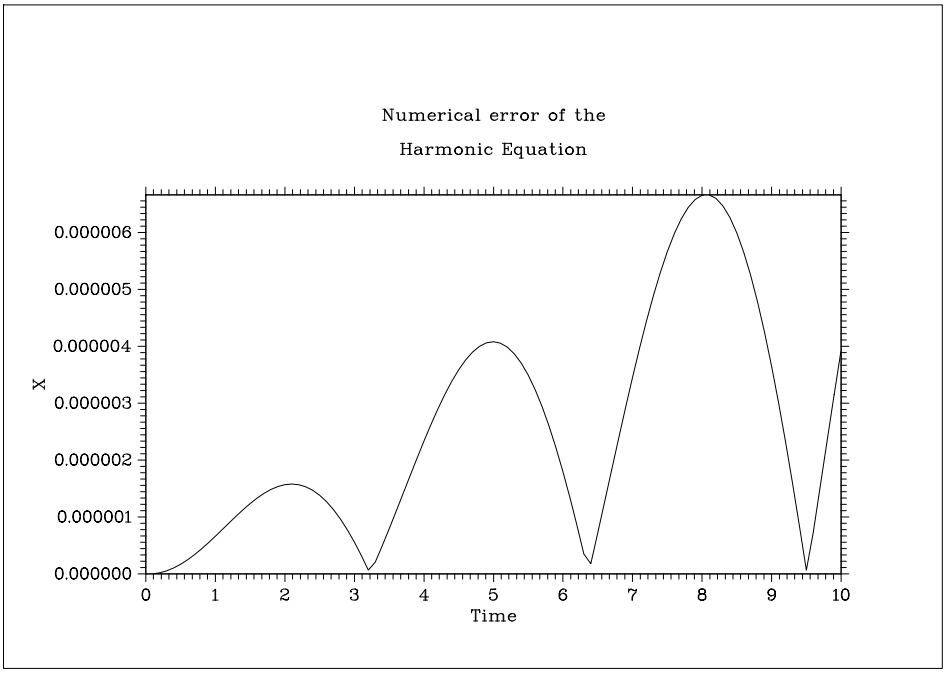

Figura 11: *Error absoluto cometido al utilizar Runge Kutta de orden 4. DISLIN.*

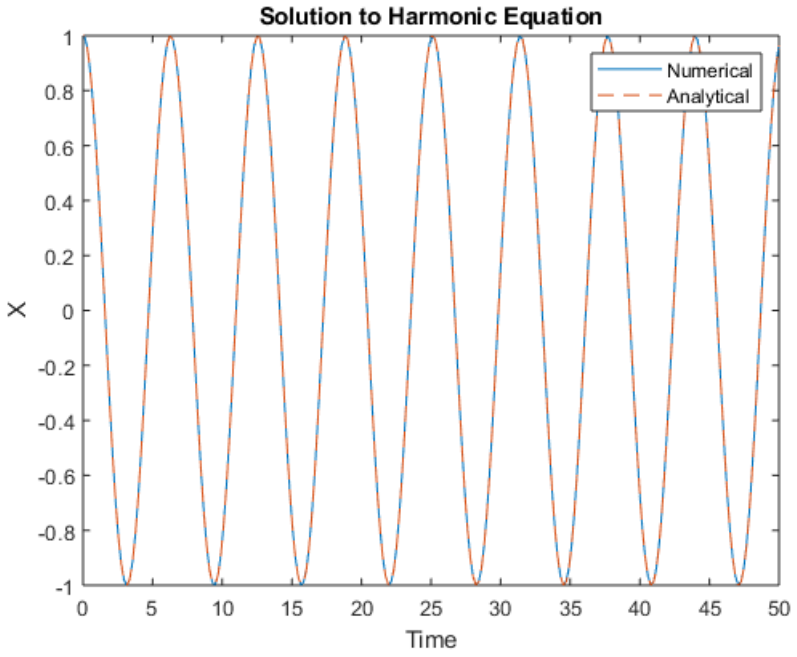

Figura 12: *Solución numérica (RK4) y analítica. Graficado con MATLAB.*

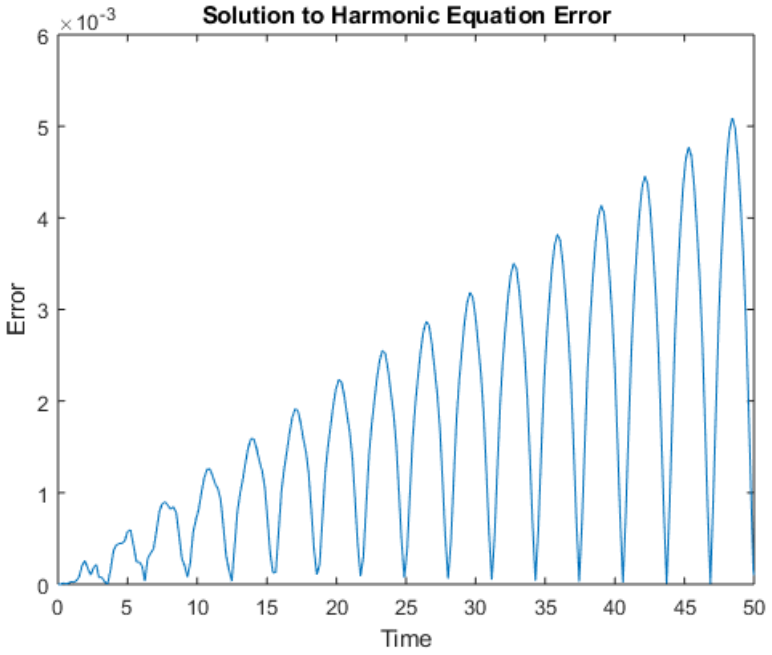

Figura 13: *Error absoluto cometido al utilizar Runge Kutta de orden 4. MATLAB.*

4. Problema de los n cuerpos

El problema de los n cuerpos consiste en la aplicación de la ley de gravitación universas en un sistema de masas puntuales. Este problema tiene solución analítica para un sistema de dos cuerpos, no así en un sistema de más cuerpos. Por ello, la resolución numérica se abre como la única vía hallar una solución mas o menos precisa (aunque bajo ciertas hipótesis existan métodos para estimación e incluso resolución del problema).

La ley de gravitación universal, como ya sabemos, tiene el siguiente aspecto:

$$F_g = G\frac{m_A m_B}{|\mathbf{r}^3|}\mathbf{r} \tag{15}$$

Donde la fuerza es producida por la masa B y es aplicada en la masa A, y G es la constante gravitacional.

Una característica de los sistemas de n cuerpos para $n > 2$ es que son inestables. Esto quiere decir que son muy sensibles a las condiciones iniciales, y que los errores numéricos se van amplificando.

4.1. Problema de los dos cuerpos

Es el sistema más simple que permite el problema y tiene solución analítica (las leyes de Kepler). Este puede ser utilizado como simplificación de los sistemas Sol-Tierra, Tierra-Luna, o Tierra-Satélite. Al existir solución analítica, es fácil la comparación para hallar el error cometido.

En Fortran se ha usado el paradigma de programación orientada a **objetos**, con un esquema temporal tipo **Euler**

Respecto a las condiciones iniciales en Fortran, la Tierra se posiciona en el origen y la luna a 400.000 km en el eje x. Sus velocidades son tales que describen orbitas circulares y su cantidad de movimiento es nula, y así su centro de masas no se mueve.

Con MATLAB se han obtenido resultados para el integrador de Euler y para el ODE45 (se trata de un Runge-Kutta de cuarto orden).

En MATLAB, las condiciones iniciales son tales que el centro de masas describe un movimiento rectilineo uniforme.

A continuación se presentan los resultados obtenidos:

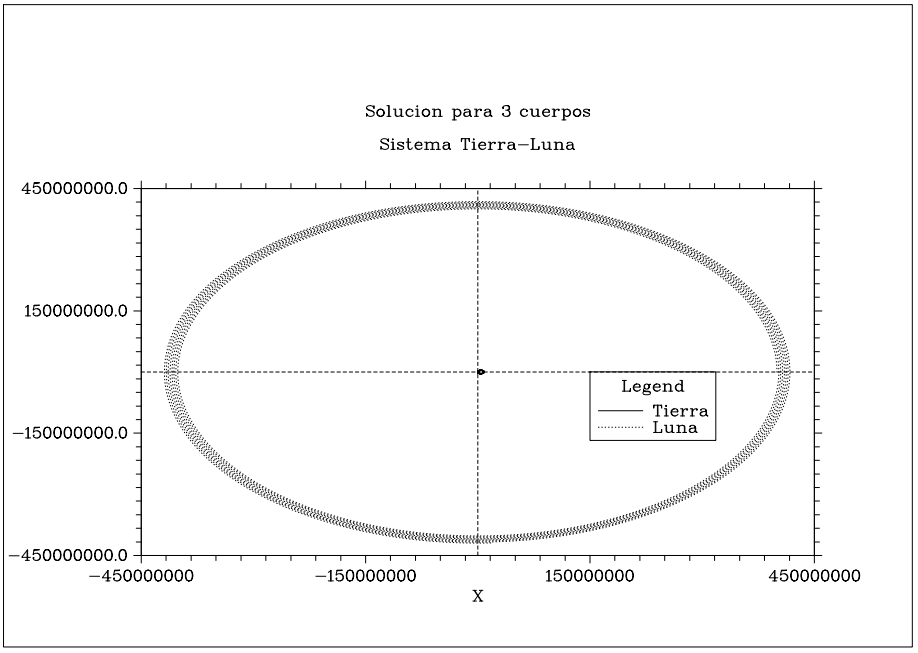

Figura 14: *Órbitas de la Tierra y la Luna respectoa un sistema inercial. En lenguaje Fortran con el método de Euler.*

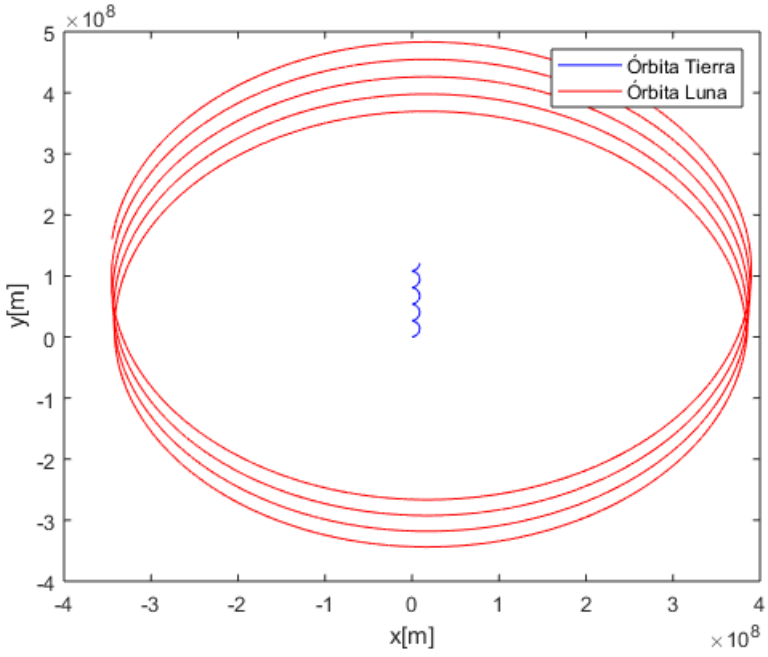

Figura 15: *Problema de los dos cuerpos con Euler. Graficado con MATLAB.*

4.2. Problema de los tres cuerpos

Se trata de un problema genérico, con la inestabilidad característica comentada anteriormente. La solución del problema numérico ya no se puede comparar con una solución analítica, aunque hay métodos para estimar el error. Las trayectorias se han realizado para un sistema inercial.

En el código Fortran se ha usado la librería *NumericalHUB* para aplicar distintos esquemas temporales. El tercer cuerpo en este caso es de una masa no despreciable respecto a las otras dos (100 veces menos que la de la luna), y su posición inicial está en el eje x, más allá de la Luna.

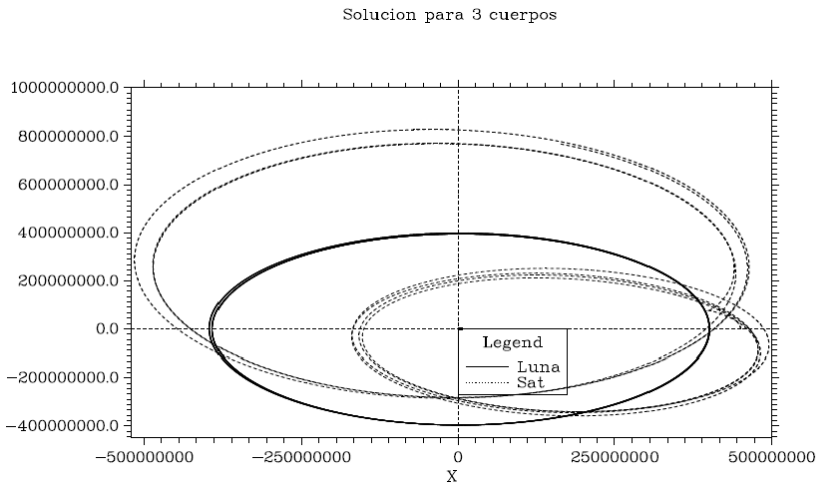

Figura 16: *Problema de los tres cuerpos con Fortran, método de RK4. Graficado con DISLIN.*

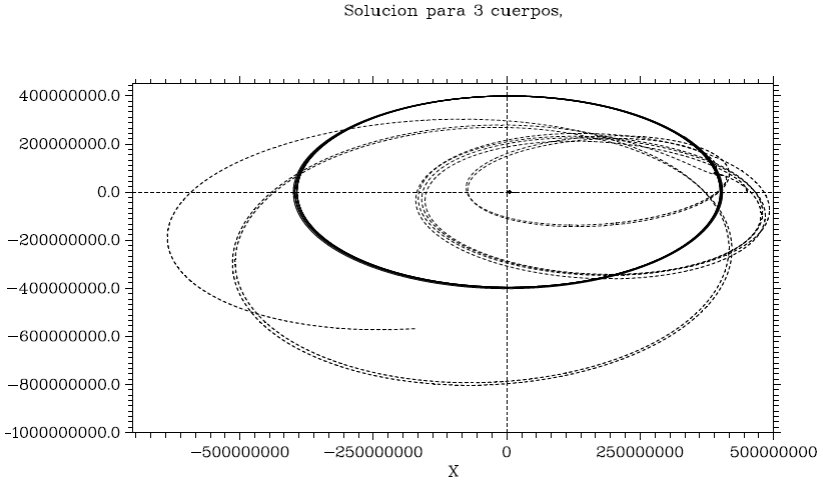

Figura 17: *Problema de los tres cuerpos con Fortran, método de Euler. Graficado con DISLIN.*

Se observa en las gráficas que el resultado es totalmente distinto, y esto es debido al carácter caótico del problema.

En MATLAB se ha realizado el análisis de la Tierra, Luna y un satélite con una masa de 100 kg, con el integrador de Euler y el ODE45.

En las gráficas mostradas a continuación se puede observar la mayor precisión del integrador ODE45 cuando se deja correr un tiempo considerable. Aplicando el integrador temporal de Euler la órbita diverge muy rápidamente:

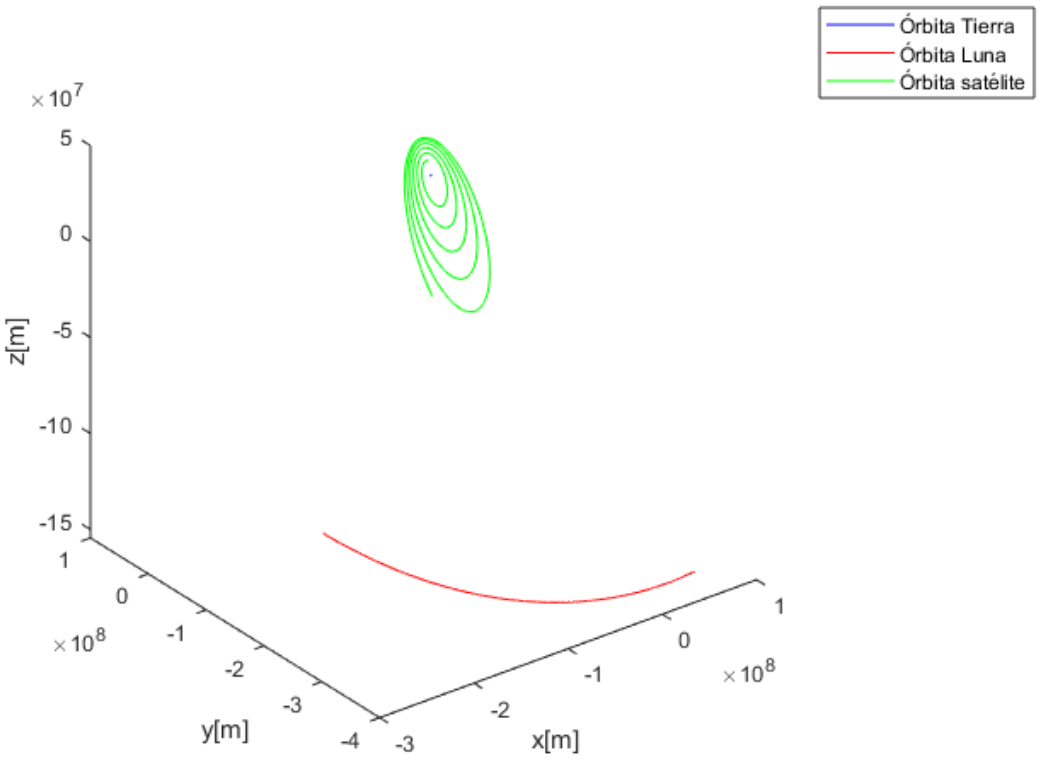

Figura 18: *Problema de los tres cuerpos con Euler. Graficado con MATLAB.*

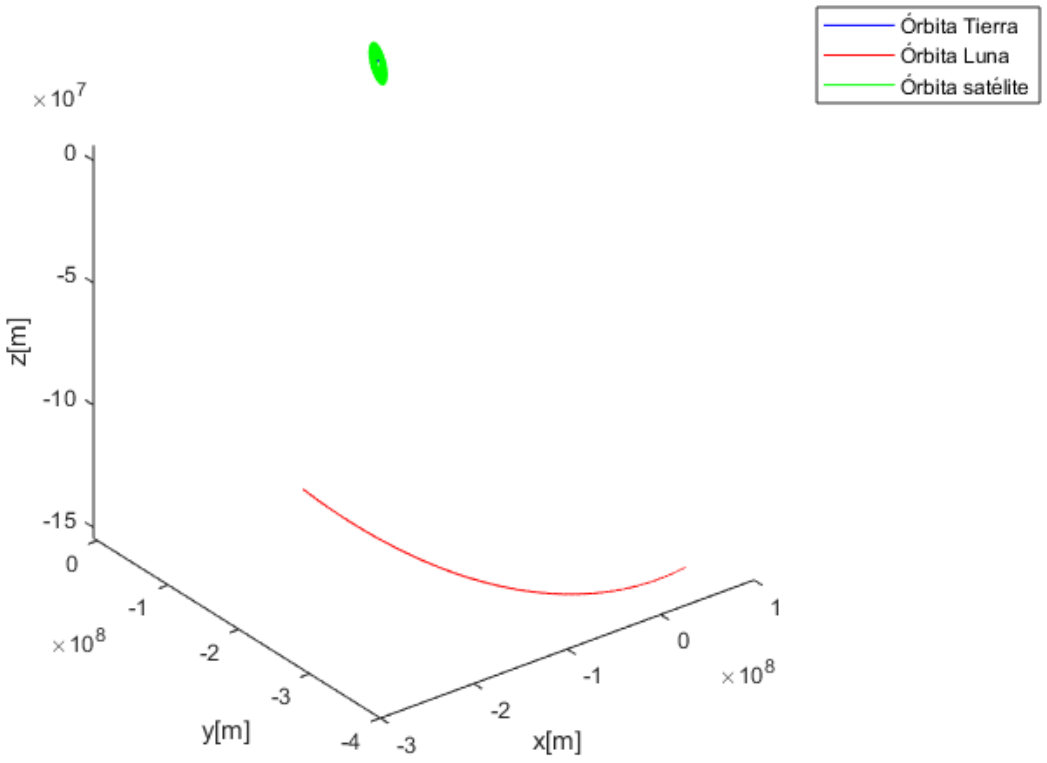

Figura 19: *Problema de los tres cuerpos con ODE45. Graficado con MATLAB.*

4.3. Puntos de Lagrange

Este es un caso específico del sistema de tres cuerpos, en la una de las tres masas es despreciada respecto a las otras dos. Existe una peculiaridad si tomamos como referencia el plano orbital de las masas no despreciadas y la recta que las une. En este sistema de referencia no inercial existen unos puntos críticos en el que la suma de fuerzas es nula (teniendo en cuenta también la fuerza centrífuga). Estos cinco puntos son los puntos de Lagrange, y de ellos unos serán estables y otros inestables.

El objetivo de este código es hallar el comportamiento de esta masa despreciable en esos puntos. Debido a las perturbaciones numéricas, si la masa está exactamente en uno de esos puntos, ésta se desplazará se desplazará sola, mostrando su comportamiento estable o inestable.

Se muestra el punto L4 (estable) en Fortran, tanto con el integrador de Euler, como el Runge-Kutta4. Las gráficas están referenciadas al centro de la Tierra, y contiene siempre el centro de la Luna.

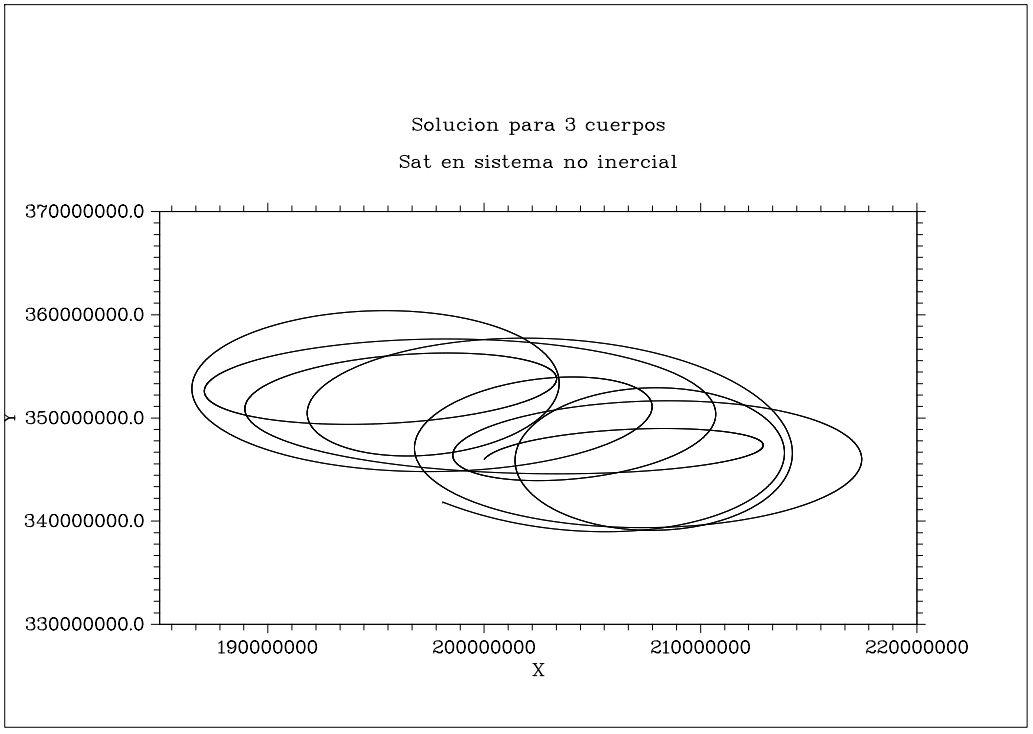

Figura 20: *Masa despreciable cerca del punto L4, método de Euler. Graficado con DISLIN.*

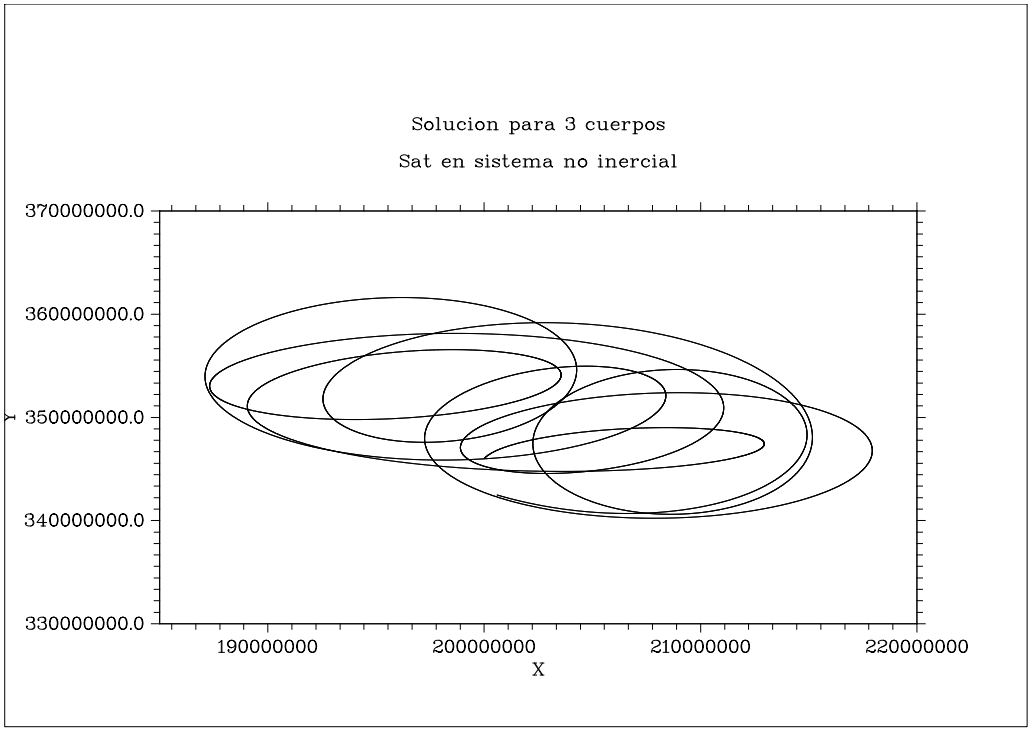

Figura 21: *Masa despreciable cerca del punto L4, lenguaje Fortran con el método de Euler. Graficado con DISLIN.*

Se puede observar su carácter estable, ya que los puntos inestables se alejan rapidamente del punto en este sistema de referencia, chocando frecuentemente con los otros cuerpos.

A continuación se muestran los resultados obtenidos con MATLAB integrando con el ODE45. Se puede observar que el punto L4 y L5 son estables, mientras que los otros tres puntos, el L1, L2 y L3, son inestables. En el L1 y L2 se observa el movimiento anómalo del satélite, mientras que en L3 la inestabilidad viene dada por el cierre continuado de la órbita. Sin embargo, puede verse en las figuras (21) y (22), cómo el satélite permanece en la órbita tras pasar el tiempo (contando con el movimiento del sistema debido a que la cantidad de movimiento es diferente de 0)

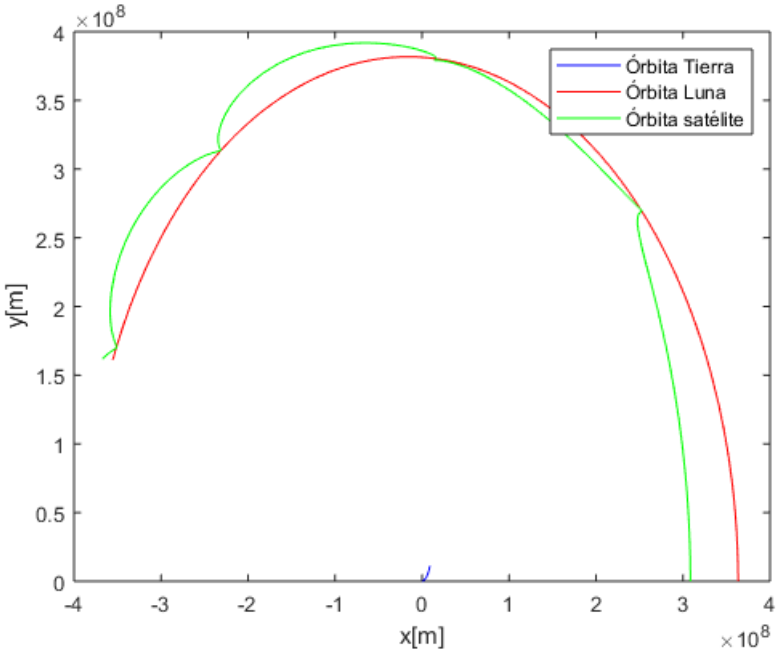

Figura 22: *Punto de Lagrange L1 (inestable). Graficado con MATLAB.*

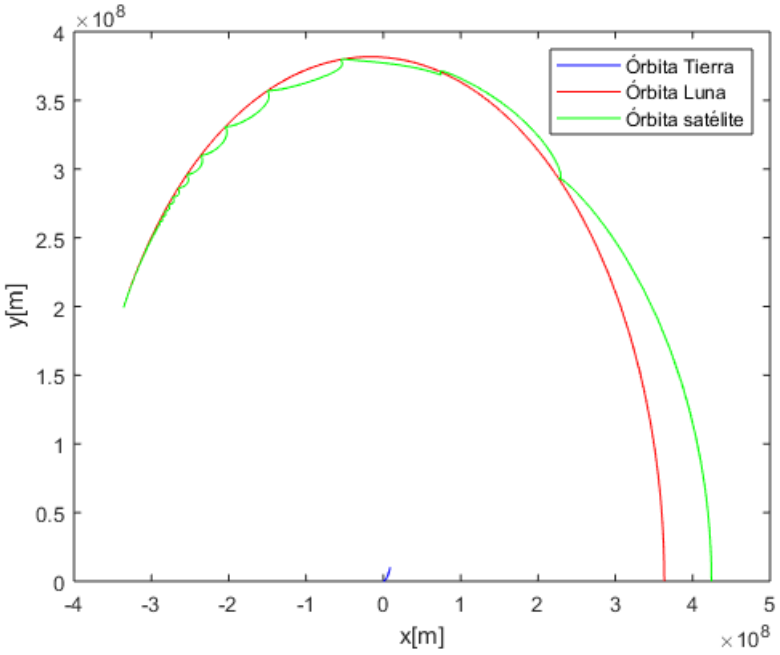

Figura 23: *Punto de Lagrange L2 (inestable). Graficado con MATLAB.*

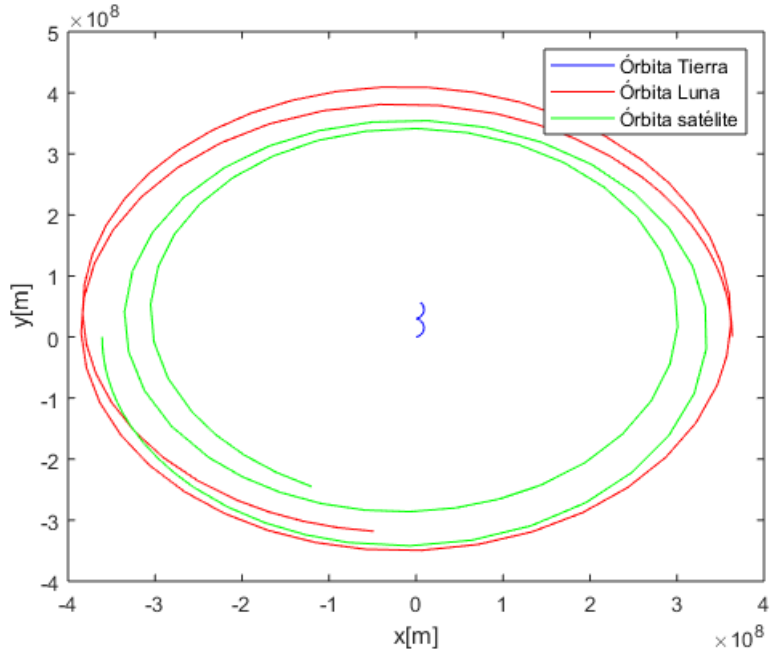

Figura 24: *Punto de Lagrange L3 (inestable). Graficado con MATLAB.*

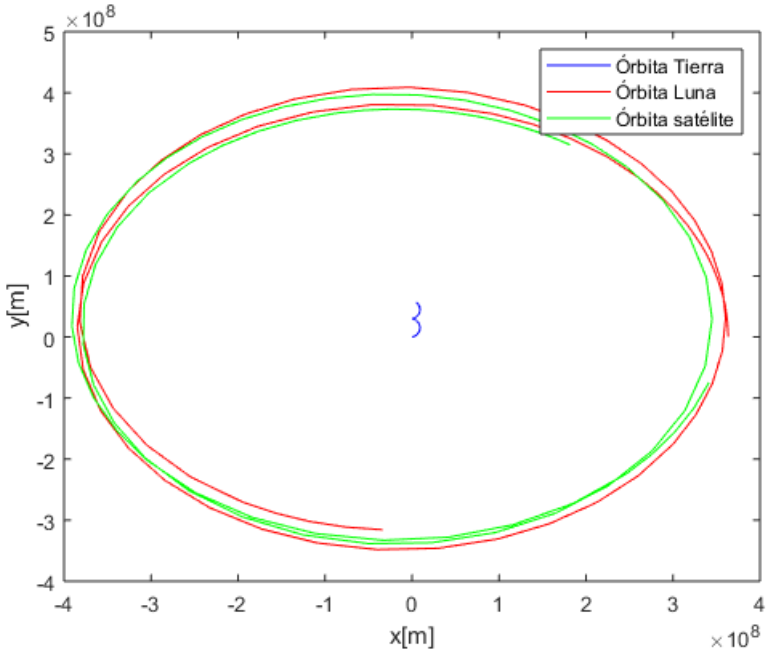

Figura 25: *Punto de Lagrange L4 (estable). Graficado con MATLAB.*

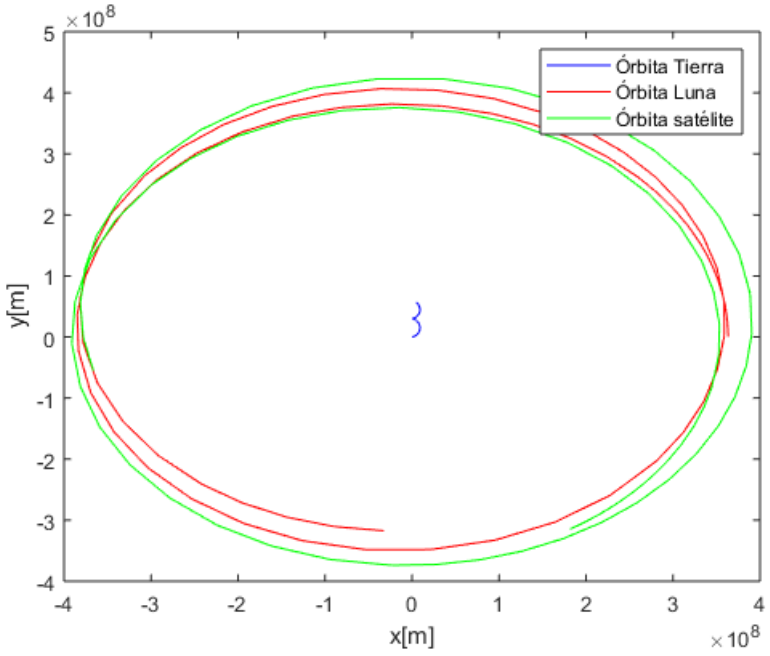

Figura 26: *Punto de Lagrange L5 (estable). Graficado con MATLAB.*

5. GMAT

GMAT es la herramienta utilizada por la NASA para el análisis de misiones espaciales. Para la resolución de las ecuaciones diferenciales utiliza integradores de alto orden, tales como el RungeKutta89, el PrinceDormand78 o el AdamsBashforthMoulton.

Se ha diseñado una misión en GMAT, en la que un satélite orbita la Tierra (considerando la influencia gravitatoria de la Luna). Por otro lado, se ha utilizado el código realizado en MATLAB para el problema de los tres cuerpos, considerando las mismas condiciones (distancia, época, ...). A continuación se muestran las gráficas que muestran la comparación de ambos casos.

Utilizando el integrador de Euler, se observa cómo la solución diverge conforme pasa el tiempo. Por otro lado, utilizando el integrador RungeKutta45, la solución aproxima mejor que la anterior, pero aún así converge con el tiempo. Con esto puede verse la necesidad de usar un integrador de muy alto orden para el cálculo de órbitas, de tal forma que la acumulación de errores sea mínima.

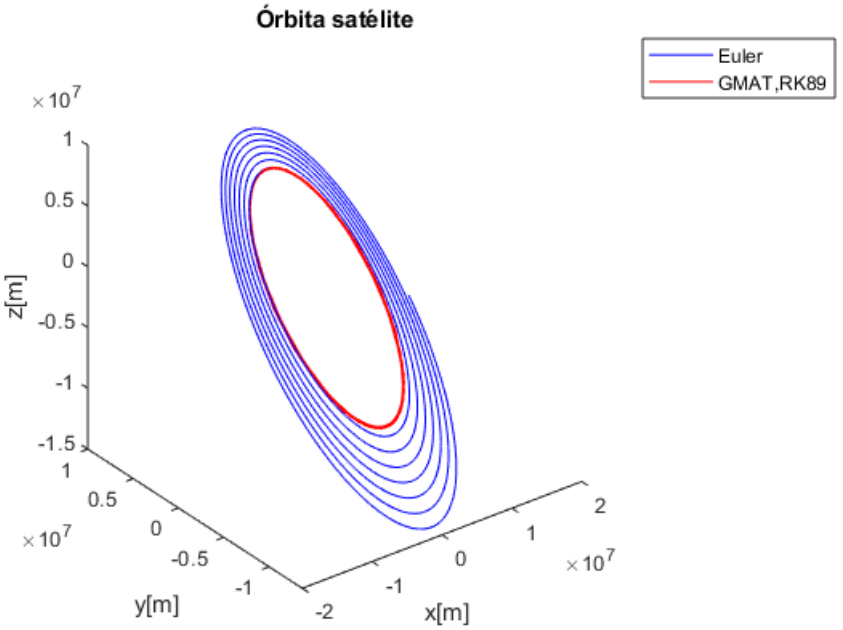

Figura 27: *Comparación resultados GMAT RK89, con código nuestro (Euler). Graficado con MATLAB.*

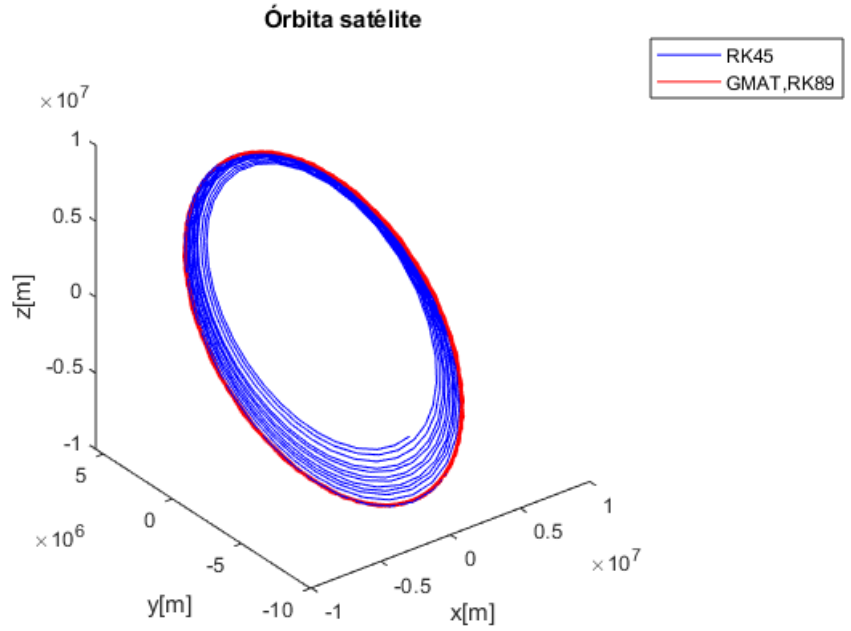

Figura 28: *Comparación resultados GMAT RK89, con código nuestro (RK45). Graficado con MATLAB.*

6. Proyecto *Black Hope* preliminar

Este problema se encuentra aún en proceso, y se ha realizado únicamente en Fortran. Aun así, a continuación se muestran los resultados obtenidos hasta el momento. Es necesario comentar que en este problema se han intentado aplicar todos los conceptos aprendidos durante este semestre.

6.1. Introducción

Una nave espacial intergaláctica de considerables dimensiones se encuentra en las proximidades de un agujero negro. Esta nave se supone similar al *Halcón Milenario* de Star Wars (es decir, con forma de disco). Se pretende estudiar el comportamiento de la nave al realizar un flyby a una cierta distancia del objeto masivo. En estas condiciones, se espera obtener los resultados de la posición del centro de masas y la rotación de la nave con el tiempo.

6.2. Desarrollo

Para simular la nave como un sólido, es necesario definir una distribución de puntos que imiten la geometría, y dotar a cada punto de una determinada masa. La distribución de la masa es fundamental a la hora de simular la rotación de la nave, ya que la fuerza de la gravedad ejercida por el agujero negro será dependiente de la posición y la masa de cada punto de la nave, generando un momento.

A continuación se describe la estructura del código realizado:

- **Program** BlackHope
  - **Module:** initialization
  - **Module:** integration
  - **Module:** gravity
  - **Module:** palomo_milenario
  - **Module:** graphic

El programa principal utiliza cinco módulos para la resolución del problema. Cada módulo a su vez, se compone de subrutinas; y estas últimas de funciones. A continuación se muestra la estructura de cada uno de los módulos, y su explicación:

- **Module:** Inicialization
  - **Subrutine:** geometry
    - **Function:** points_in_line
    - **Function:** circle_line
  - **Subrutine:** center_of_mass

El módulo *inicialización* contiene dos subrutinas. La primera de ellas se ha llamado **geometry**, y en ella se calculan los puntos que definen el sólido, y la masa que se otorga a cada uno de ellos. Para el cálculo de los puntos se han utilizado dos funciones, que aparecen en el mismo módulo, y se llaman **circle_line** (calcula la función que define el contorno del disco) y **points_in_line** (establece los puntos por filas a partir del límite establecido en la función anterior). En la segunda subrutina **center_of_mass**, se calcula el centro de masas de la nave, para posteriormente realizar la ecuación de momentos.

- **Module:** gravity
  - **Subrutine:** forces
    - **Function:** prod_vectorial

El módulo *gravity* contiene la subrutina **forces**. En ella se calcula la gravedad que tiene cada punto en función de la distancia y la masa, y se calcula la fuerza total y el momento que experimenta la nave. Se define la función **prod_vectorial** para realizar la operación matemática producto vectorial de dos vectores cualesquiera.

- **Module:** palomo_milenario
  - **Subrutine:** update_vars

El módulo *palomo_milenario* contiene la subrutina **update_vars**, en la que, una vez calculada la posición del centro de masas en el siguiente instante de tiempo, se calcula la posición de cada punto de la nave para saber la rotación que ha sufrido.

- **Module:** integration
  - **Subrutine:** accelerations
  - **Subrutine:** U_converter

El módulo *integration* contiene dos subrutinas. La primera de ellas se ha llamado **accelerations**, y en ella se escriben los vectores y las funciones de estado correspondientes. La segunda de ellas, **U_converter**, es la encargada de pasar del vector de estado obtenido a las variables físicas del problema.

- **Module:** graphic
  - **Subrutine:** write_data

El módulo *graphic* contiene la subrutina **write_data**, encargada de la escritura de los resultados obtenidos (posición del los puntos en función del tiempo) en un archivo de texto.

6.3. Resultados

Aún no se han obtenido resultados para este problema, pero en un futuro se espera obtener el movimiento del centro de masas con el tiempo, así como la rotación que sufre la nave (con una cierta distribución de masas) debido a la gravedad.

7. Conclusiones

- La metodología *Extreme programming* es realmente útil a la hora de desarrollar código, aumentando en gran medida la eficiencia del proceso.
- Es fundamental realizar el ciclo *Red-Green-Refactoring* en el desarrollo de software, ya que de esta forma se desacoplan la resolución del problema y la organización del código; lo cual permite ahorrar mucho tiempo.

- En programas no complejos, como los realizados en este trabajo, la realización del código en un lenguaje interpretado (como MATLAB) requiere menos tiempo que en uno compilado (como Fortran). Sin embargo, realizar un proyecto de grandes dimensiones en un lenguaje como MATLAB es prácticamente imposible.
- En dinámica orbital es imprescindible utilizar integradores de alto orden para conseguir soluciones estables con el tiempo.
-