UNIVERSIDAD CARLOS III DE MADRID

Proyecto fin de carrera

Interfaz gráfica de usuario y herramientas de visualización para un simulador de vuelo orbital

Alumno:
Daniel Díaz Granizo

Tutor:
Manuel Sanjurjo Rivo

Contenido

I. Introducción

1. Descripción

El proyecto "Interfaz gráfica de usuario y herramientas de visualización para un simulador de vuelo orbital" está dirigido desde el Departamento de Ingeniería Aeroespacial de la Universidad Carlos III y pretende desarrollar una interfaz de usuario con la que poder realizar simulaciones de movimientos orbitales con el propósito de generar interés para futuros estudiantes de esta especialidad. Mediante el desarrollo de este proyecto vamos a proporcionar al usuario las herramientas y funciones necesarias para configurar todo tipo de movimientos orbitales para un gran abanico de cuerpos celestes –astros, planetas, lunas y asteroides/cometas etc- y vehículos espaciales que se mueven por nuestro Sistema Solar. El objetivo es poder analizar parámetros del movimiento como la velocidad o posición y su movimiento relativo con respecto a otros astros.

En el desarrollo del proyecto también se tendrá en cuenta el hecho de que éste pueda ser modificado y mejorado por futuros estudiantes que quieran añadir nuevas funciones o quieran tomar elementos del mismo que puedan ser relevantes para otros campos de desarrollo. Así pues, se describirán ampliamente todas las funciones incluidas en el mismo con el fin de facilitar al máximo el trabajo a posibles desarrolladores futuros. La aproximación al trabajo será por lo tanto modular, de manera que el entorno de desarrollo sea flexible.

Esperamos que este, además de cumplir con las expectativas requeridas, sea una herramienta de investigación para todos los alumnos que deseen profundizar en el campo de la astrodinámica así como un elemento de apoyo para los profesores que deseen mostrar visualmente las interrelaciones de los elementos que orbitan por nuestro sistema solar.

El proyecto se propone construir una Interfaz Gráfica de Usuario que permita interactuar con el software de movimientos orbitales. Una Interfaz Gráfica de Usuario –también denominada GUI por sus siglas en inglés- es una capa superior que se le da a un programa informático para que el usuario interactúe visualmente con el software que pretende.

2. Objetivos

El desarrollo de la Interfaz Gráfica de Usuario –GUI en adelante- marca la línea de objetivos para este proyecto. Esta capa se sitúa pues entre el usuario final y el software de cálculo, brindando información gráfica al usuario de los resultados. La información resultante puede ser de cualquier naturaleza: ya sean listas ordenadas de datos, gráficos de datos, botones de acción, cuadros de texto para introducir valores, imágenes, gráficos dinámicos etc. Las posibilidades son infinitas. Así mismo, la interfaz de usuario debe, además de mostrar información, interactuar con el usuario mediante cualquiera de los métodos de entrada típicos de un ordenador –ratón, teclado, ficheros de entrada, etc- permitiendo con estos inputs modificar parámetros del software y obtener resultados acordes a lo que el usuario desea.

La interfaz gráfica de usuario que se busca en el proyecto debe de cumplir las siguientes características:

- Sencillez
- Entendible
- Configurable
- Comunicación usuario-programa

Como todas las interfaces de usuario, la principal característica debe ser la sencillez y la claridad. Este es un aspecto fundamental puesto que la mejor interfaz de usuario es aquella que no hace falta explicarse. El usuario debe ser capaz de entender las posibilidades que le brinda el software mediante una correcta organización de la información y de las interacciones entre usuario y software. Conseguir una interfaz de usuario sencilla no es a menudo fácil debido a que en general las opciones que ofrece un software son muy grandes por lo que se deberá seleccionar a conciencia qué información se muestra y en qué momentos con el fin de facilitar la compresión de los datos al usuario.

Conseguir una interfaz de usuario simple pasa primero por organizar los datos de modo entendible para el usuario. Debemos de tener en cuenta que el usuario final no tiene por qué saber qué función se está utilizando en cada momento y mucho menos comprenderla por lo que deberemos "traducir" el lenguaje del código a lenguaje de usuario. El objetivo de esto es obtener una salida entendible y sencilla que el usuario pueda comprender rápidamente y que resuma todo aquello que el software está realizando por detrás. Del mismo modo buscaremos esa sencillez para las acciones que el usuario pueda realizar.

Otro aspecto fundamental de una buena interfaz de usuario es la capacidad de obtener resultados del modo que el usuario prefiera. Esto se consigue proporcionando a la interfaz herramientas para modificar los parámetros de entrada o lo que es lo mismo, dotándola de configuración. Como configuración entendemos la capacidad del usuario de modificar todos los parámetros de ajuste que el programa admite a la entrada y según los cuales va a realizar la simulación. Dentro de la configuración aparecen parámetros que se refieren a la propia simulación como el cuerpo celeste, posición de ejes, campo gravitatorio, posición del cuerpo u otros, y parámetros que atañen al campo de la generación de la vista en 3D de la simulación numérica como resolución, capacidad de grabar el contenido en un archivo de video, etc. En este proyecto no sería difícil encontrar decenas de parámetros configurables antes de la simulación, de modo que la clave en este aspecto de la interfaz es conseguir el equilibrio entre lo que el usuario puede llegar a necesitar configurar, y el hecho de que un programa con multitud de parámetros a configurar no es utilizable por el usuario. Debemos dar libertad al usuario para que modifique datos de entrada a su antojo pero sin abrumarle debido a la gran cantidad de parámetros configurables que hay. Este aspecto de la configuración es clave y guarda mucha relación con que la interfaz sea sencilla y entendible.

Una vez que la GUI se ha entendido y se ha configurado acorde al usuario que la utiliza, debemos de dotar al software de capacidad necesaria para responder a lo que el usuario demanda intentando devolver información del modo que mejor precise éste. Una vez que la GUI se ha configurado con los parámetros de la simulación, el usuario deberá iniciar la misma. La simulación es cerrada; se ejecuta de inicio a fin abarcando el tiempo que se le haya determinado en la GUI y al final de la misma se devuelve la información que el usuario haya demandado en la configuración. Por lo tanto conseguir distintos modos de comunicación entre software y usuario es imprescindible, con el fin de mejorar la compresión de los resultados que simulará el núcleo. Dentro de este campo entraría por ejemplo, la capacidad de devolver el resultado calculado en forma de tablas de datos en archivos planos, lo cual sería desde un punto de vista científico suficiente, o bien, con el objetivo de mejorar la comunicación usuario-software, intentar darle formato a esos datos para facilitarle el trabajo al usuario final. Algunos de los formatos serían por ejemplo los utilizados como estándares por programas como Matlab. También esto es aplicable a otras salidas como son gráficas o sencillamente datos por pantalla.

La segunda parte del proyecto se centra en el desarrollo de las interfaces entre los distintos módulos del proyecto así como desarrollar en la medida que sea posible el software que hay por debajo. Para ello se realizará un trabajo de investigación primero con el que definir qué es lo que esperamos que nuestro software consiga, qué partes están ya desarrolladas por otros investigadores, qué partes no, y de todas esas partes que no están desarrolladas, cuáles merece la pena desarrollar en relación al trabajo que conllevan y el rendimiento que obtendríamos en caso de desarrollarlas.

Por debajo de la interfaz de usuario se ejecutará el software de simulación. Este será el núcleo del programa y contendrá las leyes físicas correspondientes a las interacciones entre los objetos que se hayan seleccionado desde la interfaz. Este núcleo será el "universo" de este proyecto.

Otro objetivo que se persigue será la fiabilidad. Se intentará cubrir todo el espectro de posibilidades que se puedan dar en cualquier de los cuatro bloques –de los que hablaremos a continuación- con el fin de prevenir cualquier tipo de error en la lectura de datos por éstos. Como se dice, ningún programa informático está exento de errores y a mayor complejidad del mismo, el número de éstos crece de modo exponencial. No obstante se intentará cubrir con la mayor eficacia posible todas las posibilidades que generen algún tipo de error en cualquiera de los bloques que componen el programa.

Una vez que tengamos una interfaz sencilla y limitados los errores al máximo, el último objetivo que nos marcamos es la exactitud en los resultados. Cómo de exactos serán estos resultados comparados con los reales. Para ello, una vez realizado todo el trabajo y con la GUI funcionando se llevará a cabo un trabajo de validación de resultados apoyándonos en herramientas ya existentes para comparar los datos calculados.

3. Estructura del proyecto

El software que se pretende desarrollar consta de cuatro grandes bloques que interactúan entre si intercambiando información y modificando la respuesta de los objetos que entran en juego dentro de la simulación.

- Interfaz de usuario: ya definida en párrafos anteriores, se trataría del software de más alto nivel para el usuario, y con el que interactuaría estableciendo los parámetros de la simulación. La selección de lenguaje de programación para este bloque ha sido de .NET de Microsoft Visual Studio. Las razones de la elección son su facilidad para crear aplicaciones visualmente agradables, la sencillez para lanzar otro tipo de programas mediante comando y la simplicidad de código.

- Simulador: se trata del "universo" de nuestro proyecto. En él se establecerán unos parámetros iniciales –cuerpos, posiciones, ejes de referencia etc- con los que se iniciará la simulación de la órbita. Este bloque no ofrece visualización alguna, simplemente el cálculo secuencial de las interacciones gravitacionales entre todos los objetos que se seleccionen desde la interfaz de usuario. Ya veremos más adelante que el núcleo de cálculo más profundo es una función Matlab. A pesar de ello y como explicaremos más adelante, el lenguaje utilizado para el simulador será el Visual C/C++ de Microsoft Visual Studio por el hecho de que junto con el Fortran, es el lenguaje que se recomienda para el uso del interpretador Matlab Engine.

- Integrador: se trataría del núcleo de cálculo. Una vez que desde el simulador se han organizado las ecuaciones diferenciales que representarán el movimiento, éstas son pasadas al integrador que las resolverá mediante funciones nativas de Matlab. Se trata por tanto del núcleo del programa con el que se deberán realizar miles de "steps" de tiempo para cada simulación. Una vez terminados los cálculos, la información será devuelta al simulador que gestionará las salidas en función de lo solicitado por el usuario en la GUI.

- Visualización 3D: este último bloque trabajará generando un entorno tridimensional con los objetos seleccionados desde la interfaz de usuario. Mediante el intercambio de información con el simulador, este bloque

generará la simulación en tiempo real de la nave sobre el cuerpo que le atrae. Para esta labor hemos seleccionado como lenguaje Java debido a su versatilidad de plataformas. Además, utilizaremos las librerías open source de JMonkey integradas en el IDE Eclipse para la creación de entornos 3D.

Vamos a dedicar unas líneas a intentar argumentar el hecho de que mezclemos tantos lenguajes y entornos, y si esto perjudica o beneficia al resultado final del proyecto.

Desde el punto de vista de desarrollo, para la creación de los cuatro módulos necesitamos tres grandes software de programación: Matlab, Visual Studio y IDE Eclipse (JMonkey). Son más tecnologías de las que en principio serían recomendadas. Todas ellas de distinta sintaxis y organización del código. La programación se hará en cuatro lenguajes –Matlab script, .NET, C/C++ y Java- y por tanto se deberán crear interfaces comunes de nexo entre programas. No obstante parece más recomendable priorizar el uso de tecnologías específicas para cada uno de los bloques que la intercomunicación entre ellos. Recordemos que la estructuración del proyecto en bloques fue establecida antes de comenzar la investigación sobre qué tecnología era mejor usar en cada bloque. Además la parte de las interfaces es una parte importante del proyecto. La idea de tomar para cada bloque la tecnología que se adecua más al mismo es una idea acertada para simplificar el trabajo de dicho bloque. También una parte del trabajo se basará en hacer cada uno de esos bloques independientes y consistentes con las informaciones y datos que otros bloques puedan enviarle consiguiendo una independencia casi total y por tanto facilitándole a cada uno de los bloques la capacidad de ser mejorado en el futuro de manera autónoma sin tener por qué conocer los cuatro lenguajes de programación.

A todo esto hay que sumarle el hecho de que cada uno de los bloques realiza funciones totalmente distintas a las del resto, de modo que tomando en cada caso la tecnología que más se adecue a lo que buscamos conseguiremos una

optimización del rendimiento del código además de mejores resultados. Poniendo en una balanza la simplicidad de programar cada bloque en un lenguaje específico para obtener el mejor rendimiento en ese bloque frente a la simplicidad en las interfaces –o canales de comunicación entre bloques- pesa más la primera siendo por tanto este tipo de organización la que se ha elegido para el proyecto.

Cada bloque se comunicará con el resto mediante interfaces que ocuparán un capitulo en este proyecto. Por lo tanto, podemos suponer por el momento que todos los bloques poseen la capacidad de enviar información al resto y éstos la capacidad de entenderla y procesarla.

La secuencia de ejecución de cada bloque, así como el intercambio de información entre los mismos será la siguiente:

1. El usuario inicia el software, cargándose como primera ventana la interfaz creada en .NET.
2. El usuario, una vez seleccionados los parámetros de operación necesarios, comenzará la simulación pulsando en el botón Iniciar.
3. La interfaz de usuario se cerrará e iniciará el simulador con parámetros de entrada, los seleccionados por el usuario, y éste comenzará a realizar los cálculos.
4. El simulador ejecutará cálculos hasta que se cumpla una condición predefinida en la GUI como podría ser tiempo transcurrido, posición del vehículo espacial o cualquier otra.
5. Mientras que la simulación se está desarrollando, el bloque del simulador creará un canal de información bidireccional con el bloque del integrador.
6. Una vez que termine el simulador por haber alcanzado la condición preestablecida, se cerrará el simulador y volverá a abrirse la GUI en modo Output mostrando todas las salidas y los archivos generados.
7. Si el usuario lo desea, en el modo Output de la GUI podrá ejecutar la simulación en 3D.

Este sería el esquema fundamental del proyecto. En los párrafos sucesivos intentaremos explicar y exprimir todos los puntos de interés para que el proyecto se lleve a cabo. Así mismo realizaremos trabajos de investigación con el fin de minimizar la carga del proyecto y utilizar herramientas ya diseñadas por otros miembros de la universidad.

II. Flujo de trabajo

El primer punto que se realizará será el de documentación de qué es lo que esperamos obtener como resultado de nuestro proyecto. Para ello revisaremos otros programas que realicen tareas similares. Finalmente intentaremos definir las necesidades que debe de cubrir nuestro proyecto.

1. Revisión de software similar

Un buen punto de partida para lo que se desea lograr lo podemos encontrar en la multitud de simuladores ya existentes que distintas empresas y organizaciones bridan gratuitamente para el estudio del campo que nos atañe. La lista de software similar propuesta por el director del proyecto incluye los siguientes nombres: GMAT, STK, Orbiter.

A continuación realizamos un pequeño análisis de cada uno de ellos intentando extraer pequeñas trazas de lo que se espera que que sea nuestro proyecto.

General Mission Analysis Tool (GMAT)

Se trata de un software creado por la NASA. Es por tanto el paradigma de lo que nos gustaría conseguir en nuestro simulador. Comenzaremos por describir la GUI.

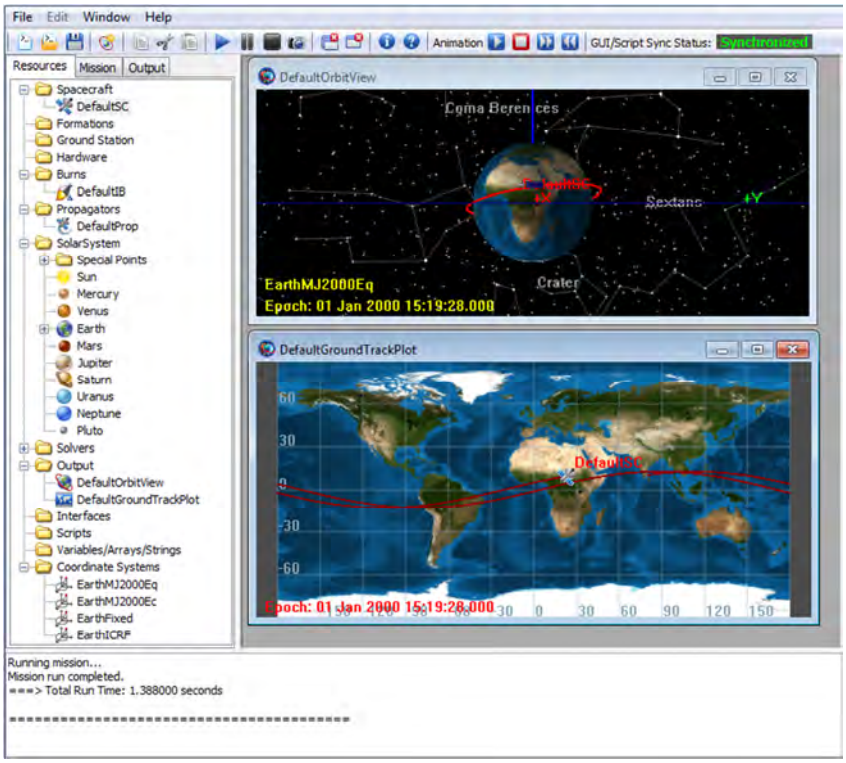


Figura II.1. Vista previa de interfaz de usuario de GMAT

Como podemos observar, se trata de una interfaz de usuario muy bien organizada. Comenzando por la parte superior, observamos a la izquierda un menú de edición de misiones, seguido de un menú de control de la simulación así como de la animación. En la parte izquierda tenemos un menú con tres pestañas. En la primera de ellas denominada "Resources" vemos una lista de carpetas que contienen distintos elementos que pueden intervenir en la simulación. Cada elemento de esas carpetas es totalmente configurable y el resultado de la simulación dependerá pues de los parámetros que se introduzcan. La organización de los elementos es impecable y no deja lugar a dudas de lo que contiene cada una de las carpetas. En la segunda pestaña denominada "Mission" tenemos el elemento que configuraría la misión. En este caso se trata de propagar el movimiento, pero del mismo modo podría configurarse para cualquier otro objetivo. Finalmente en la pestaña "Outputs" se organizan todas las salidas que genera el sistema, entre las que se encuentran la visualización 3D del movimiento y el mapa de la tierra con la proyección de la órbita en 2D, como se observa en la imagen. La simulación puede arrojar también otros datos que sólo deberían de especificarse antes de iniciar la simulación.

Sin duda alguna lo que esperamos obtener de nuestro proyecto se parece mucho a lo que encontramos aquí. Conociendo nuestras limitaciones por tiempo y por recursos intentaremos construir una versión simplificada incluyendo tan solo aquellas opciones totalmente necesarias.

Orbiter

En este caso, este programa se aleja un poco más de lo que buscamos puesto que Orbiter se trata de un software para simular en 3D misiones de cualquier índole. Es más un juego o un simulador con el que poder manejar vehículos espaciales en distintas misiones, incluso desde dentro de la propia cabina de control, como si fuésemos astronautas. Mediante el menú inicial que vemos en la imagen podemos configurar la misión y otros parámetros más de índole gráfica, así como los controles de entrada que usemos (teclado, joystick etc). El grado de realismo es enorme y basta con manipular una de las naves en distintas misiones para darse cuenta del nivel de detalle en el cálculo de movimientos orbitales en el espacio y ver el grado de dificultad que implica estar dentro de una nave.

Figura II.2. Vista previa de menú inicial de Orbiter

Alguna de las misiones que podemos controlar en este programa son: maniobra de aproximación a la ISS, desacoplamiento de la ISS, maniobra de despegue con una Atlantis o sencillamente la exploración de planetas como Saturno.

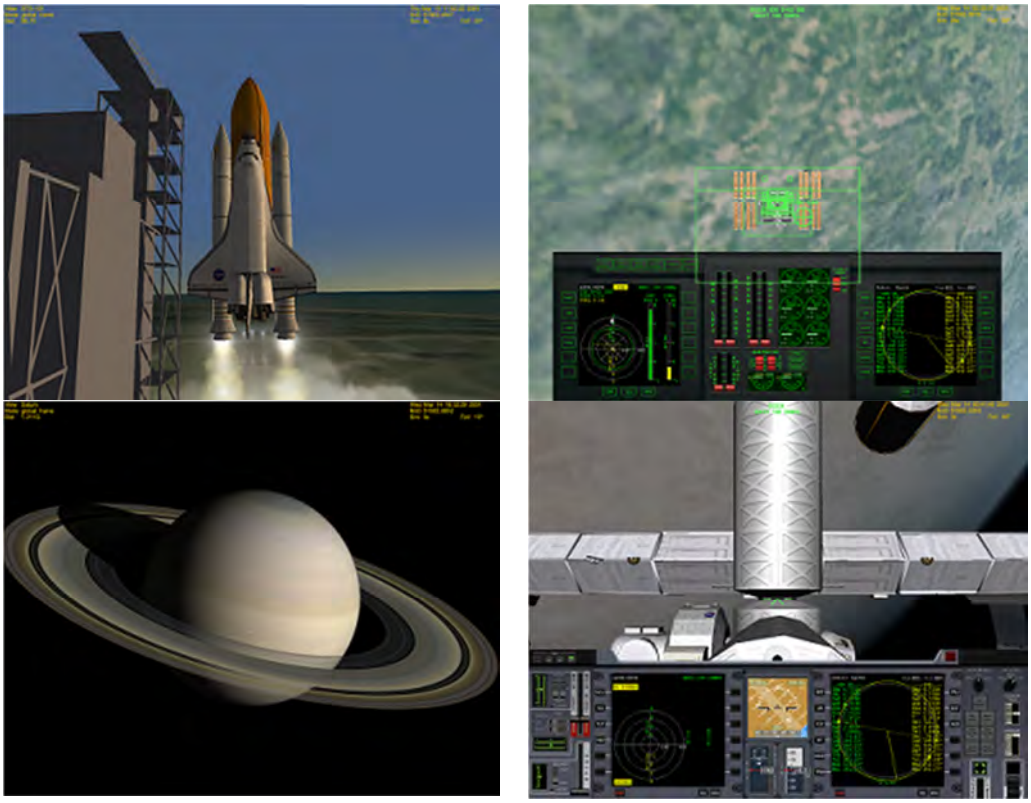


Figura II.3. Ejemplos de misiones en Orbiter

En las imágenes se puede observar el alto nivel de detalle en el modelado 3D.

STK de AGI

Software creado por la compañía AGI para la simulación de movimientos de satélites sobre la tierra con la posibilidad de configurar bases terrestres, navales, etc.

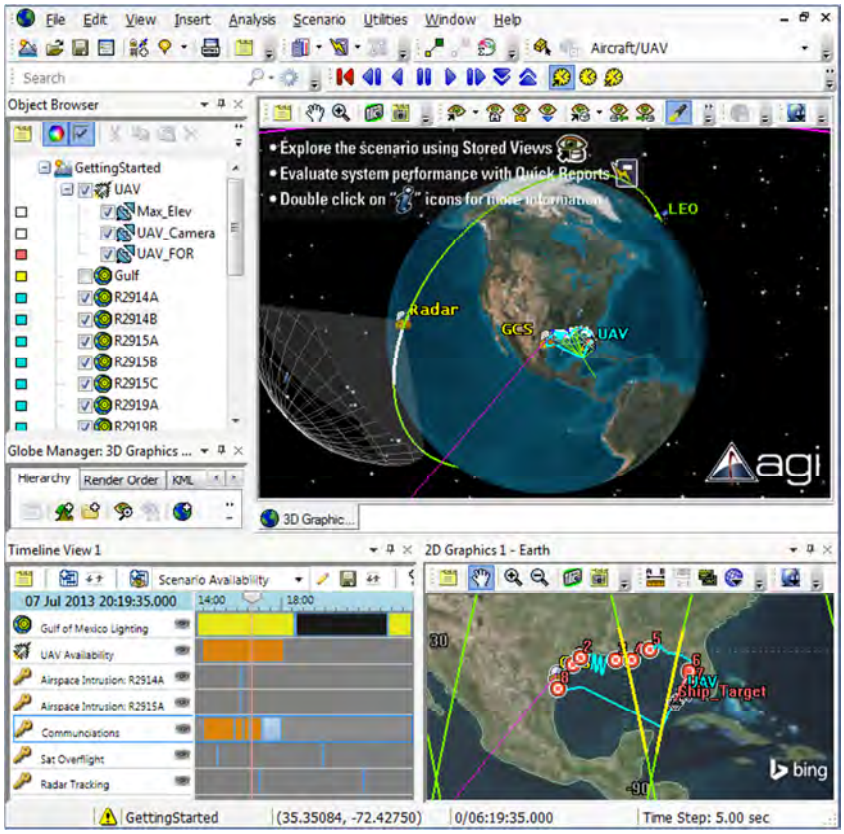


Figura II.4. Vista previa de interfaz de usuario en STK

Como observamos, la GUI es parecida a la de GMAT, aunque bastante más sobrecargada. No obstante, este aumento de elementos en la pantalla permite a priori una configuración más exacta de la misión que queremos realizar. En la parte superior vemos menús de edición de misiones, así como el control de tiempo de la simulación. Una diferencia con GMAT es que en éste la barra de *Control de tiempo* y la de *Control de simulación* están fundidas en una sola. De modo que ambos conceptos serían uno solo: la simulación 3D se genera en tiempo real a medida que los cálculos se van realizando.

A la izquierda tenemos el menú de inspector de objetos, donde podemos observar todos los elementos que existen en la visualización 3D. En la parte inferior tenemos el *Timeline* que a medida que avanza la simulación va

recorriendo todos los objetos. Esta función es importante además de útil porque se puede inspeccionar el estado de cada uno de los objetos para un instante de tiempo dado. Finalmente tenemos la visualización en 3D de los movimientos que el simulador va calculando. Se pueden observar con gran precisión los movimientos relativos de todos los elementos y la interacción entre ellos como podría ser la comunicación entre un satélite LEO y una base marina que se está moviendo.

El último punto a destacar del software es la integración nativa con Bing Maps que permite obtener en tiempo real imágenes del servicio de Microsoft para colocarlas sobre el globo terrestre.

2. Identificación de las necesidades y requisitos de la GUI. Necesidades de visualización

Tras el estudio de los simuladores propuestos se empiezan a definir todas las características que conforman nuestra GUI así como los parámetros que se podrán configurar en la misma.

Como se ha dicho con anterioridad debemos dejarnos en el tintero muchos elementos interesantes de estos programas por la complejidad que conllevan en la realización de este proyecto. Por lo tanto, intentaremos hacer una selección acertada de los mismos buscando maximizar los resultados.

Para empezar, nuestra GUI trabajará con *misiones*. Cada misión estará definida por una cantidad amplia parámetros que expondremos a continuación. Lo que la GUI nos permitirá será gestionar esa misión y modificar los parámetros de la misma. Además, mediante la barra superior podremos guardar la misión en un archivo, cargar misiones ya guardadas o comenzar una nueva misión. Todos los parámetros que introduzcamos en la interfaz tendrán un significado único y describirán de manera inequívoca los elementos que definirán tanto el entorno como la nave así como parámetros de integración u otros. Así pues la misión consistirá en un paquete con los siguientes parámetros:

- Nave. Paquete de datos que contendrá toda la información configurable acerca de la nave tanto a nivel de cálculo como a nivel de visualización. Algunos de los datos que incluirá este paquete de información serán las condiciones iniciales –posición, velocidad y tiempo- así como los parámetros físicos de la nave –masa o coeficiente de drag entre otros-. Se

habilitará, además, la posibilidad de elegir modelo tridimensional para la simulación.

- Sistemas de referencia. Se trata de un bloque de información fijo y no configurable con varios sistemas de referencia tanto inerciales como no inerciales. En posteriores puntos de la memoria definiremos aquellos ejes de referencia que se consideran los más más convenientes.
- Propagador. Paquete de información que contiene todas las características que definen al sistema de integración como los *steps* máximo y mínimo, el tipo de función integradora o la tolerancia. Además incluirá el modelo de fuerzas que se utilizará para la integración pudiéndose activar o desactivar perturbaciones que explicaremos más adelante. Se incorpora, además, en este apartado, una pestaña para seleccionar un motor de cohete y poder realizar maniobras.
- Motor de cohete. Se trata un elemento que añade fuerza al movimiento y se usa para realizar maniobras. Se basa en los motores utilizados en vehículos espaciales que contienen un propulsante químico que produce grandes empujes en periodos de tiempo muy cortos. Si únicamente se requiere la propagación de una órbita sin maniobras, no será necesario este elemento.
- Sistema solar. Contendrá toda la información relacionada con el sistema solar. Para cada elemento de interés –planetas o lunas- se dispondrá de los datos característicos como la constante planetaria, el achatamiento o el radio ecuatorial.
- Outputs. Quizás sea esta la parte de la misión más compleja y más interesante. Se trata de un gestor de salidas de información. Cada una de esas salidas se podrá realizar mediante archivo de datos, gráfica XY, representación de groundtrack, o simulación en 3D. Mediante el gestor podremos añadir y eliminar salidas y configurar cada una para que nos muestre las variables que deseamos de un modo preciso.
- Variables. Gestor de variables para añadir nuevas a las que ya hay y poder obtener gráficas u archivos de informe con las variables que el usuario prefiera.

Estas serán en principio las necesidades que deberá cubrir la GUI. Se trabajará para conseguir encajar en una interfaz de usuario todas estas características que podrán modificarse con arreglo a las preferencias del usuario.

Una vez tenemos las variables bien definidas y su organización, vamos a intentar definir las necesidades de visualización que deberá incorporar nuestro proyecto.

Dado que trabajamos con órbitas espaciales se hace imprescindible crear algún tipo de entorno tridimensional con el que poder interactuar y visualizar los resultados. Nuestro proyecto intenta cubrir esta necesidad mediante la creación de un entorno tridimensional con objetos sobre los cuales superpondremos una textura o imagen para así crear un planeta o una luna. La imagen principal que se ha elegido para La Tierra será una descargada de la web de la NASA que tiene resolución 5K.

Figura II.5. Imagen de la superficie terrestre en alta resolución

Una resolución de 5K nos garantiza buenos resultados para las distancias características que recorreremos en los cálculos; no obstante se queda algo corto cuando intentamos hacer demasiado zoom sobre algún punto en concreto. En la página web de la NASA hay otras imágenes con mayor resolución –de hasta 20K de resolución - pero el problema es que estas imágenes ocupan por si solas unos 250MB de memoria por lo que no se hace viable utilizarlas debido a que ralentizaría demasiado todos los procesos de renderizado.

Para subsanar este problema también se ha pensado en habilitar un botón de simulación sobre Google Earth, con imágenes reales del globo terrestre y con posibilidad de hacer zoom hasta zonas muy cercanas a la superficie terrestre.

El objetivo en este punto es implementar ambas opciones. La primera opción –la desarrollada en este proyecto- no necesitará ningún complemento extra para instalar. En cambio, para la segunda opción, la de Google Earth será necesario primeramente instalar este programa además de modificar algunas opciones dentro de sus menús, como explicamos en el Anexo 4.

Del mismo modo que haremos para La Tierra, lo repetiremos para La Luna y otros. La imagen que hemos usado de La Luna para el renderizado ha sido también tomada de la página web de la Nasa y tiene una resolución 4K.

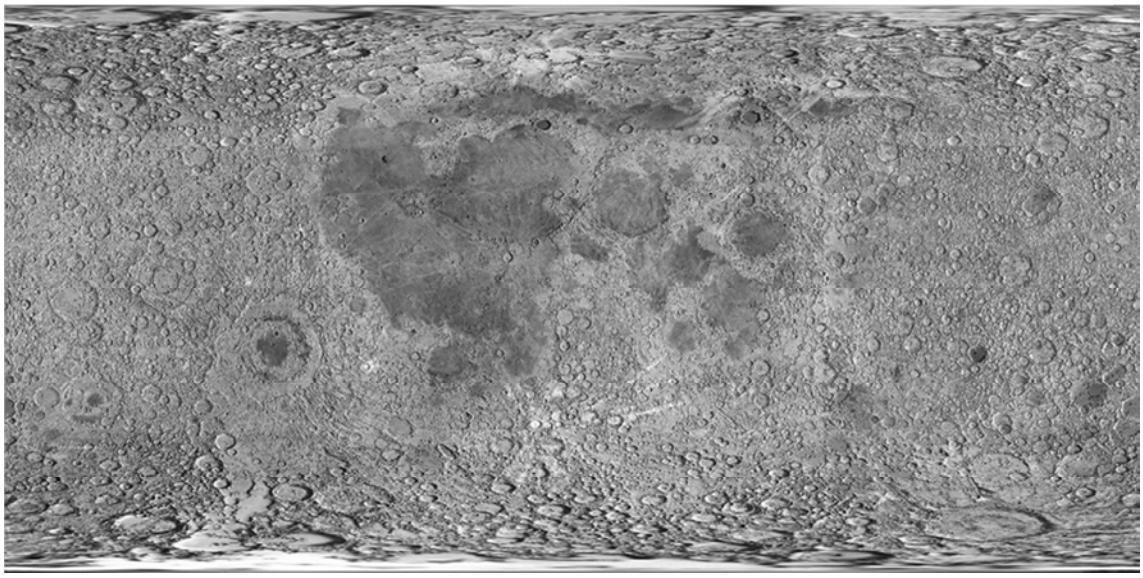


Figura II.6. Imagen de la superficie lunar en alta resolución

El proceder con respecto a Google Earth será el mismo y habilitaremos la opción de poder visualizar las órbitas mediante este software sobre la superficie lunar.

Para el resto de planetas se han tomado distintas imágenes descargadas de internet que intentan emular su superficie.

Además de todo esto, y para terminar con las necesidades de visualización, se dará la opción de crear mapas de tierra de latitud/longitud para poder observar el movimiento del satélite sobre la superficie, ya sea terrestre o lunar.

III. Cálculos

1. Mecánica orbital

Para obtener la órbita de un objeto que se mueve alrededor de un cuerpo celeste es necesario definir tanto la posición $\vec{r}$ como la velocidad $\vec{v}$, ambos vectores con tres coordenadas cada uno. Por lo tanto nos encontramos ante un problema de 6 variables.

La complicación la encontramos al intentar obtener estas 6 variables utilizando el método clásico de cálculo aplicando las leyes de Newton sobre las fuerzas, así como las de gravitación y las leyes de Kepler. Básicamente el cálculo se realizaría del siguiente modo:

Sean un cuerpo celeste de masa m_c sobre el que orbita otro cuerpo celeste de masa m, la fuerza de atracción que existe entre ambos cuerpos viene definido por la ley de gravitación universal de Newton:

$$\vec{F} = -G\frac{m_c m}{|r|^2}\overrightarrow{u_r} \qquad (III.1)$$

Donde G es la constante de gravitación universal de valor $G = 6.67348 \times 10^{-11}\frac{Nm^2}{kg^2}$, $|r|$ es el módulo del vector posición y $\overrightarrow{u_r}$ un vector unitario en la misma dirección y sentido que $\vec{r}$.

Realizamos el cambio de variable $\mu_p = G \cdot m_c$ siendo μ_p el conocido *parámetro de gravitación estándar* referenciado para un planeta con unidades $[\mu_p] = \frac{km^3}{s^2}$, utilizado comúnmente en el cálculo de órbitas alrededor de planetas, y cuyos valores se encuentran tabulados para los distintos planetas del sistema solar. Además, podemos expresar el vector unitario como $\overrightarrow{u_r} = \frac{\vec{r}}{|r|}$. Con esto llegamos a la expresión:

$$\vec{F} = -\mu_p\frac{m}{|r|^2}\cdot\frac{\vec{r}}{|r|}$$

Aplicando a continuación la segunda ley de Newton y teniendo en cuenta que la aceleración es la derivada segunda de la posición, obtenemos:

$$-\mu_p\frac{m}{|r|^2}\cdot\frac{\vec{r}}{|r|} = m\frac{d^2\vec{r}}{dt^2}$$

Expresión que requiere de ciertas hipótesis iniciales como que $m_c \gg m$, que los cuerpos son esféricamente simétricos y además que no existen perturbaciones de otros cuerpos. Para finalizar podemos eliminar la masa del problema obteniendo la siguiente ecuación diferencial:

$$\frac{d^2\vec{r}}{dt^2} + \frac{\mu_p}{|r|^2} \cdot \frac{\vec{r}}{|r|} = 0 \qquad (III.2)$$

Esta ecuación describe el movimiento restringido a dos cuerpos donde no se considera la atracción gravitatoria del cuerpo pequeño sobre el mayor y se trata de una ecuación diferencial vectorial de segundo orden no lineal. Es aquí donde encontramos el principal problema en el uso del sistema clásico de cálculo de movimientos. Para obtener las coordenadas para un tiempo dado se debe integrar la ecuación entera, no obstante podemos ahorrarnos esta integración acudiendo a los elementos orbitales keplerianos. Este planteamiento es igualmente valido en el estudio de movimientos orbitales y se basa en la definición de 6 parámetros (elementos orbitales) los cuales, junto a la teoría de Kepler sobre las órbitas, describen de igual modo todos los puntos de una órbita resolviendo de manera alternativa el sistema de ecuaciones diferenciales.

Leyes de Kepler

Antes de comenzar con los elementos orbitales se hace necesario enumerar rápidamente las leyes de Kepler sobre el movimiento de cuerpos celestes.

1. Primera ley. Todos los planetas se desplazan alrededor del sol describiendo orbitas elípticas. El sol se encuentra en uno de los focos de la elipse.
2. Segunda ley. El radio vector que une un planeta y el sol barre áreas iguales en tiempos iguales.
3. Tercera ley. Para cualquier planeta, el cuadrado de su periodo orbital es directamente proporcional al cubo de la longitud del semieje mayor de su órbita elíptica.

El proceso para la construcción de la órbita consistirá en definir primeramente la elipse que genera la órbita (2 variables). A continuación se orienta espacialmente la órbita (3 variables) y finalmente se establece un punto de inicio (1 variable). Con las cinco primeras variables se define la órbita, mientras que al añadir la sexta variable podemos establecer relación entre el tiempo y la

posición. Generalmente en las órbitas dadas por elementos orbitales se necesitan 6 parámetros

Elementos orbitales

Para definir una órbita necesitamos conocer seis parámetros independientes de ésta. Los comúnmente utilizados son el semieje mayor, la excentricidad, la inclinación, longitud de nodo ascendente, argumento de perigeo y anomalía verdadera.

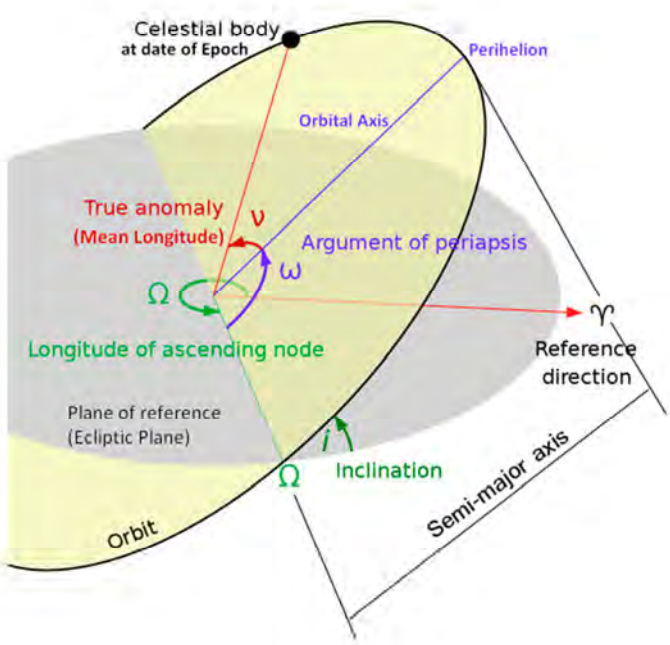


Figura III.1. Esquema de posicionamiento mediante elementos orbitales

No obstante existen otros tantos que serían totalmente válidos. A continuación se enumera una lista de parámetros que podemos encontrar dentro de los cálculos de elementos orbitales.

$a[km]$ **Semieje mayor:** distancia del centro de la elipse al punto más alejado de la misma.

$e[\,]$ **Excentricidad:** grado de desviación de una sección cónica con respecto a la circunferencia.

$v[rad]$ **Anomalía verdadera:** ángulo que se forma entre el semieje mayor y el punto de la órbita elíptica en el que se encuentra el cuerpo medido desde el foco.

$E[rad]$ **Anomalía excéntrica**: el ángulo que forma el punto de intersección

de la línea perpendicular al semieje mayor que pasa por el planeta, y la circunferencia exterior de la elipse, con respecto al eje del perihelio de su órbita, medido desde el centro de la elipse.

$M[rad]$ **Anomalía media**: el ángulo recorrido por el planeta (E), en un intervalo de días dado, medido desde el semieje mayor.

$i[rad]$ **Inclinación:** ángulo que forma el plano de la órbita respecto al plano de referencia.

$\Omega[rad]$ **Longitud de nodo ascendente:** es el ángulo con vértice el foco de la elipse más cercano del perihelio que va desde el punto Aries hasta el nodo ascendente.

Nodo ascendente: es el punto perteneciente a la órbita que corta con el plano de referencia. Se denomina ascendente al punto de la órbita en el cual el vehículo o cuerpo pasa del hemisferio inferior al superior. Al contrario se le denomina descendente.

$\omega[rad]$ **Argumento de periastro:** ángulo que va desde el nodo ascendente hasta el periastro, medido en el plano orbital y el sentido de giro del astro. Cuando la órbita es alrededor del sol se le denomina **argumento de perihelio**. Cuando la órbita es alrededor de la tierra se le denomina **argumento de perigeo**.

La relación que existe entre la anomalía verdadera y la anomalía excéntrica es la siguiente:

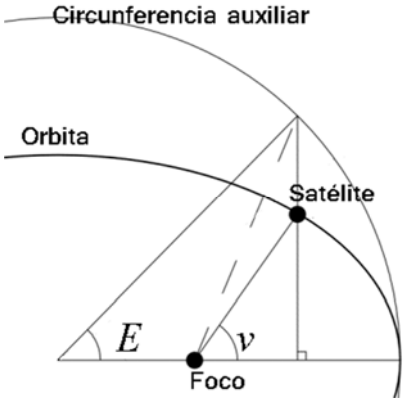


Figura III.2. Relación entre anomalía verdadera y anomalía excéntrica.

En el presente proyecto se ofrecerá la posibilidad de introducir las condiciones iniciales de la nave con coordenadas cartesianas y también con elementos orbitales keplerianos.

2. Potencial gravitatorio

El cálculo del simulador se fundamenta en obtener para cada punto e instante el potencial gravitatorio que ejerce el cuerpo de masa mayor m_c sobre el cuerpo de masa menor m. Se trata entonces del proceso fundamental del núcleo del simulador.

La definición clásica de potencial gravitatorio dada por la ley de gravitación universal de Newton nos dice:

$$U = G \int_{\Omega} \frac{\rho(\vec{s}) d^3 \vec{s}}{|\vec{r} - \vec{s}|} \qquad (III.3)$$

Donde G es la constante de gravitación universal, ρ la densidad, $\vec{s}$ es el vector que recorre el cuerpo de volumen Ω y $\vec{r}$ es la posición desde la que estamos calculando el potencial.

Para conseguir un cálculo analítico exacto en cuerpos masivos no perfectamente esféricos se recurre al desarrollo en armónicos esféricos con las funciones asociadas de Legendre. Para poder aplicar esta teoría y obtener un cálculo analítico exacto es necesario trabajar bajo la hipótesis de densidad constante.

Definimos nuestro punto espacial con coordenadas esféricas del siguiente modo:

$$x = r \cos\phi \cos\lambda$$
$$y = r \cos\phi \sin\phi$$
$$z = r \sin\phi$$

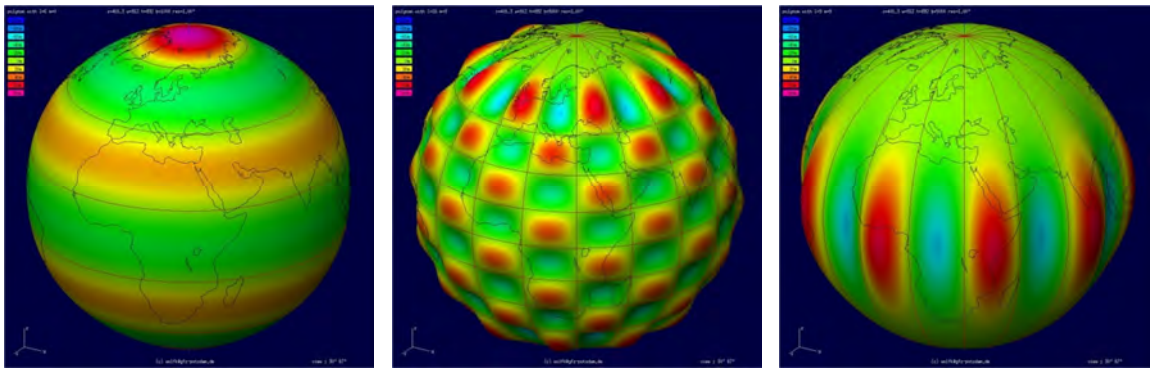
Con ϕ la latitud medida desde el plano $x - y$, r el radio y λ la longitud medida desde el eje x en dirección positiva Este.

El modelo generalmente propuesto para la resolución mediante esféricos armónicos que cumplen las condiciones de contorno de la esfera sería:

$$U(r, \phi, \lambda) = \frac{\mu}{r}\left[\sum_{n=1}^{n_{max}} \sum_{m=0}^{n} \left(\frac{R}{r}\right)^n \bar{P}_{n,m}[sin\phi]\left(\bar{C}_{n,m} \cos(m\lambda) + \bar{S}_{n,m} \sin(m\lambda)\right)\right] \qquad (III.4)$$

Donde μ es el parámetro de gravitación estándar, R es el radio del cuerpo primario, $P_{n,m}$ es el polinomio de Legendre asociado de orden n y grado m y los coeficientes $C_{n,m}$ y $S_{n,m}$ son los denominados Coeficientes armónicos de distribución de masa normalizados o coeficientes de Stoke que nos da una idea de cómo seccionamos y distribuimos la masa de la esfera para obtener mejores aproximaciones en el potencial gravitatorio. Estos coeficientes se obtienen experimentalmente y, bajo ciertas condiciones, en el límite $\lim_{n_{max}\to+\infty} U(r,\phi,\lambda)$ obtendríamos el resultado exacto de potencial gravitatorio generado por ese cuerpo. Para cuerpos celestes, la convergencia de los polinomios de Legendre se garantiza bajo ciertas premisas. Mientras que para un planeta la serie convergería –debido a su forma esférica-, para un asteroide con composición de formas cóncavas y convexas no ocurriría lo mismo.

El problema reside en el coste computacional, ya que en general utilizar órdenes y grados 10 veces mayores conlleva un coste computacional 100 veces mayor.



Zonal con n=6 y m=0 | Teseral con n=16 y m=9 | Sectorial con n=9 y m=9

Figura III.3. Ejemplos para esféricos armónicos $\bar{P}_{n,m}[sin\,\phi]\,cos\,m\lambda$ *de* -1 *(azul) a* $+1$ *(violeta)*

La relación que existe entre el potencial y la aceleración gravitatoria en cada punto viene establecido por el hecho de que éste deriva de una fuerza conservativa, de modo que:

$$\nabla U = \vec{g} \qquad (III.5)$$

El método converge al aumentar el grado y el orden. Se puede comprobar realizando la prueba en un punto arbitrario dado y aumentando el grado en cada prueba y midiendo la diferencia:

$$\Delta U = |U^k - U^{k-1}| \qquad (III.6)$$

Donde U^k representa el potencial calculado en el punto arbitrario de grado k. Realizando el cálculo podemos comprobar que $\Delta U \to 0$ si $k \to \infty$.

3. Perturbaciones

Como ya demostramos en el apartado de elementos orbitales, necesitamos resolver la ecuación diferencial $III.2$ para obtener el vector de estado de posición y velocidad:

$$\frac{d^2\vec{r}}{dt^2} + \frac{\mu_p}{|r|^2} \cdot \frac{\vec{r}}{|r|} = 0$$

No obstante los resultados arrojados por la resolución de esta ecuación son una primera aproximación y esto es debido a que tenemos que considerar otras fuerzas que existen sobre la nave, fuerzas que tendremos que incluir en el cálculo del siguiente modo:

$$\frac{d^2\vec{r}}{dt^2} + \frac{\mu_p}{|r|^2} \cdot \frac{\vec{r}}{|r|} = \vec{f}_{Pert} \qquad (III.7)$$

Siendo el término $\vec{f}_{Pert}$ la suma de todas las fuerzas de perturbación que explicaremos a continuación.

Existen más fuerzas actuando sobre un vehículo espacial además de la gravedad de la tierra atrayendo a la misma. El suponer que tenemos un sistema binario con dos cuerpos –planeta y nave- implica grandes simplificaciones del problema y obtención de planteamientos resolubles mediante los métodos clásicos que conocemos de ecuaciones diferenciales. Esta suposición nos facilita la tarea de conocer el comportamiento de los cuerpos sometidos a fuerzas gravitatorias no obstante hay que considerar que si no incluimos el resto de fuerzas que realmente existen nuestra solución contendrá un error mayor. Con el fin de aumentar la precisión en los resultados obteniendo valores mucho más aproximados debemos de tener en cuenta las perturbaciones que sufren los cuerpos. Estas perturbaciones tienen su origen en dos elementos:

- Cuerpos planetarios o astros
- Fuerzas superficiales

Dentro del primer grupo encontramos las perturbaciones debidas a la no uniformidad en la distribución de masa del planeta sobre el que estamos orbitando así como las debidas a otros cuerpos celestes.

En el segundo grupo encontramos las fuerzas que aparecen en el vehículo espacial debidas a su sección como pueden ser la resistencia aerodinámica o la presión por radiación solar.

En la siguiente figura mostramos todas las fuerzas apreciables que existen sobre un vehículo espacial en función de la altitud que tenga este.

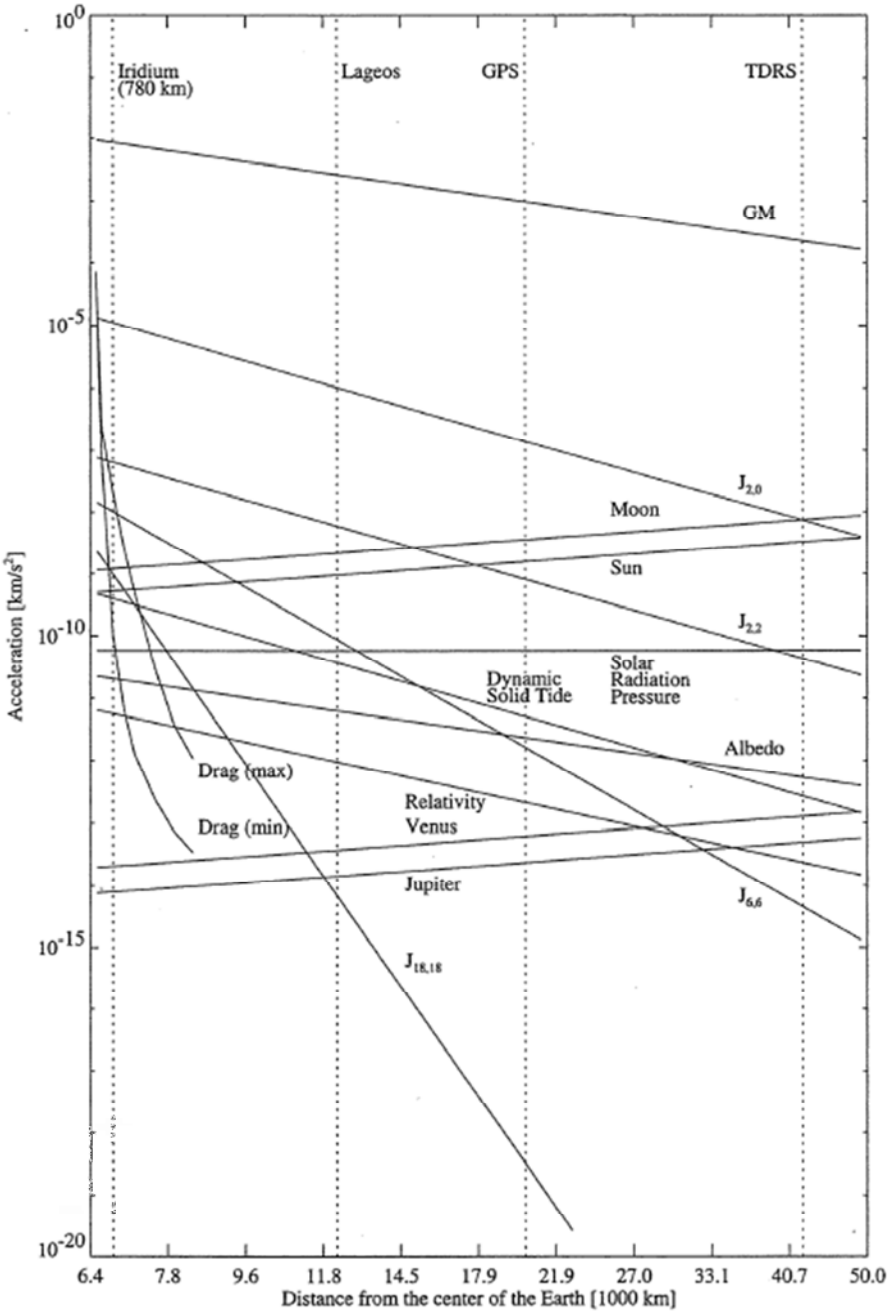


Figura III.4. Fuerzas que actúan sobre un vehículo espacial en función de la altura

Las líneas discontinuas en vertical nos muestran a qué altitud se encuentran los satélites nombrados. Así pues podríamos argumentar que, por ejemplo, para el satélite de GPS no sería computacionalmente eficiente tomar valores de J18 puesto que incluirían términos de 10^{-18}. Del mismo modo no le afectaría la fuerza por drag debido a la altura a la que orbita.

De modo que desde el punto de vista del cálculo para cada situación podremos simplificar el problema quitando términos que no vayan a afectar considerablemente al resultado. Con esto último puesto en mente, y con el fin de simplificar tanto el desarrollo como la utilización final que se le dé al software, en nuestro proyecto se han tomado únicamente las siguientes fuerzas de perturbación que a continuación detallamos.

Perturbaciones debidas a otros cuerpos celestes

El motivo por el que utilizamos la simplificación de $m_{SC} \ll m_c$ es debido a que para un problema de $n-cuerpos$ necesitamos $6n$ ecuaciones las cuales normalmente no tenemos. Realizando la simplificación conseguimos reducir el número de ecuaciones y en el caso de que incluyamos perturbaciones por la presencia de otros cuerpos debemos mantener esta simplificación.

En el caso de un vehículo espacial orbitando alrededor de la tierra y teniendo en cuenta a la Luna es sencillo comprobar que $m_{SC} \ll m_{Tierra} \sim 6 \times 10^{24} Kg$ y también que $m_{SC} \ll m_{Luna} \sim 7 \times 10^{22} Kg$. Por tanto, en el caso de incluir perturbaciones de grandes órdenes de magnitud provocará un aumento en la precisión de la solución además de no suponer un coste computacional mucho mayor. El cálculo de la perturbación se realizará mediante la expresión:

$$\vec{f}_{cuerpo-i} = \mu_i \left(\frac{\vec{d}_i}{\left|\vec{d}_i\right|^3} - \frac{\vec{r}_i}{|\vec{r}_i|^3} \right) \qquad (III.8)$$

Siendo μ_i el parámetro de gravitación del $cuerpo-i$, $\vec{d}_i$ el vector posición del $cuerpo-i$ respecto de nuestra vehículo y $\vec{r}_i$ el vector posición del $cuerpo-i$ respecto del cuerpo central.

La perturbación total generada por $n-cuerpos$ será:

$$\vec{f}_{n-cuerpos} = \sum_{i=1}^{n} \vec{f}_{cuerpo-i} \qquad (III.9)$$

Siendo n el número total de cuerpos a considerar.

Mediante la interfaz de usuario se podrá elegir qué cuerpos incluir en el cálculo.

Perturbaciones debido a la distribución no homogénea de masa de la Tierra

En la teoría el potencial gravitatorio generado por una esfera de masa –planeta- sería del orden de $U{\sim}1/r$. No obstante ni La Tierra ni ningún otro planeta es una esfera perfecta con densidad homogénea lo que produce desajustes en esa relación. Como vimos más arriba en la ecuación $III.4$, el potencial gravitatorio lo calculábamos como:

$$U(r,\phi,\lambda) = \frac{\mu}{r}\left[\sum_{n=1}^{n_{max}}\sum_{m=0}^{n}\left(\frac{R}{r}\right)^{n}\bar{P}_{n,m}[sin\phi]\left(\bar{C}_{n,m}\cos(m\lambda)+\bar{S}_{n,m}\sin(m\lambda)\right)\right]$$

El factor J_n que depende directamente de los coeficientes $\bar{C}_{n,m}$ y $\bar{S}_{n,m}$ nos da una idea del grado de distribución de masa del planeta. Aplicando grados y órdenes mayores obtendremos un resultado mucho más ajustado del potencial gravitatorio. Para nuestro simulador incluiremos grados y ordenes hasta el valor 5 $n,m \leq 5$.

Perturbaciones debidas a la fuerza de resistencia aerodinámica

La fuerza de resistencia atmosférica aparece debido a que a pesar de haber salido de la atmósfera existen bajas concentraciones de partículas relacionadas con la misma. La concentración de estas partículas desciende a medida que nos alejamos de La Tierra. Para satélites que se mueven en orbitas bajas este efecto es mucho más elevado que para satélites que orbitan a mucha distancia.

Cuanto más denso sea el medio más fuerza necesitará el vehículo para moverse. Además dentro de la ecuación se incluye la superficie efectiva que nos da una idea de la sección máxima que tiene el vehículo en la dirección axial en la que se produce el movimiento. Así, cuanto más superficie tenga, más resistencia aerodinámica –también denominada Drag- aparecerá. El valor de la fuerza se calcula como:

$$\vec{f}_D = \frac{1}{2}\rho\frac{A}{M}C_D V_r^2\left(\frac{-\vec{V}_r}{|V_r|}\right) \qquad (III.10)$$

La fórmula que da resultado negativo vectorialmente indica que se produce en dirección opuesta a la velocidad relativa entre el vehículo y atmósfera. La velocidad relativa del vehículo espacial V_r es una incógnita que se resuelve con la propia simulación. Esta velocidad relativa se construye como la diferencia entre la velocidad del vehículo –que es una variable de estado- y la velocidad de la atmósfera –que se suele suponer que "corrota" con La Tierra- Por otro lado la densidad ρ del fluido es conocida mediante una expresión que explicamos a continuación y la sección transversal A del vehículo también. M es la masa de la nave. El coeficiente de Drag C_D es estimado. Conocidos todos los datos podemos estimar la fuerza de Drag que deberemos incluir en el sistema de cálculo.

Existen muchos modelos de cálculo de la densidad. Algunos analíticos, como los que se presentan, y otros basados en datos experimentales. Para nuestro proyecto se han tomado dos. El primero, el más sencillo, que dada una altura h calcula la densidad con la expresión:

$$\rho = \rho_0 e^{\frac{h-h_0}{H}} \qquad (III.11)$$

Siendo ρ_0 la densidad conocida a $h_0 = 400Km$ $\rho_0 = 3.725 \times 10^{-12} kg/m^3$ y $H = 58.515Km$ altura de referencia del exterior de la atmósfera.

El segundo modelo utiliza la misma expresión $III.11$ que el anterior pero en este caso, dada una altura h, los valores correspondientes de ρ_0 y H se obtienen de interpolar en la siguiente lista, entrando con el valor de la altura:

$h[Km]$	$\rho_0[Kg/m^3]$	$H[Km]$
150	2.07e-9	22.525
180	5.464e-10	29.740
200	2.789e-10	37.105
250	7.248e-11	45.546
300	2.418e-11	53.628
350	9.518e-12	53.298
400	3.725e-12	58.515
450	1.585e-12	60.828
500	6.967e-13	63.822

Tabla III.1. Valores de densidad del aire en función de la altura.

Mediante la Interfaz de usuario que desarrollamos para este simulador damos la opción de elegir entre ambos modelos: simplificado o complejo. Los motivos por los que no fijamos el modelo complejo como modelo por defecto son para dejar de mano del usuario final el considerar el tiempo de computación como una variable más.

La distinción entre modelo simple y complejo de Drag incluye también una consideración respecto a la velocidad de la atmósfera medida desde un marco de referencia inercial. Si optamos por el modelo simplificado la velocidad de la atmosfera en el punto que se encuentra el vehículo será $V_{atm} = 0$ mientras que si tomamos el modelo completo, la velocidad de la atmósfera se calculara como:

$$\vec{V}_{atm} = \vec{\omega} \times \vec{r} \qquad (III.12)$$

Siendo $\vec{r}$ la posición del vehículo espacial y $\vec{\omega}$ la velocidad angular de giro de la atmósfera que se calcula como $\vec{\omega} = M\vec{\omega}_r$ con M la matriz de rotación en el marco ME2000 para el cuerpo central que tratemos –que se obtiene mediante funciones incluidas en CSPICE del que hablaremos más adelante- y ω_r la velocidad de rotación del planeta definida como $\omega_r = 2\pi/T$ en rad/s y T el tiempo sideral de una órbita en segundos.

De modo que si tomamos el modelo complejo la velocidad que el simulador tendrá en cuenta será la relativa entre la nave y la atmósfera $\vec{V_r}' = \vec{V}_r - \vec{V}_{atm}$.

Volviendo a la fuerza de arrastre, la fuerza de Drag será mayor en el perigeo de la órbita puesto que en ese punto se alcanzará el máximo en velocidad y en densidad.

Una vez calculados los datos se ha podido comprobar que incluyendo esta fuerza aerodinámica, para órbitas elípticas se obtiene como resultado una disminución del semieje mayor y para orbitas circulares una disminución del radio que conllevará un aumento de velocidad del vehículo y la consiguiente disminución del periodo.

En la Tabla III.2 mostramos el orden de perturbación que representan cada uno de estos parámetros para órbitas de 500Km y para órbitas geoestacionarias siendo A la sección efectiva de la nave en m^2 y M la masa de la nave en Kg.

Como podemos observar, a 500Km de altura el orden de la fuerza es $f_D \sim 10^{-5}$ y si nos separamos lo suficiente como para una órbita geoestacionaria a unos

35.780Km el orden decae hasta $f_D \sim 10^{-13}$ lo cual tiene muy poco impacto sobre nuestro simulador.

Perturbaciones debidas a la presión por radiación solar

Otra fuerza que aparece sobre los vehículos espaciales es la Presión por radiación solar o SRP por sus siglas en ingles. La luz que incide del sol aporta una cantidad de momento lineal sobre los objetos que ilumina. Si una superficie refleja luz solar existe un intercambio de momentos sobre esa superficie que aunque sea pequeño, es medible y produce sobre esa superficie el 'efecto de presión'. Lo mismo ocurre si una superficie absorbe luz aunque en este caso el efecto es mucho menor.

Esa presión se calcula como:

$$P = F_*/C \qquad (III.13)$$

Siendo F_* el flujo de energía solar que produce el sol en el punto de la órbita con unidades $[W/m^2]$ y C la velocidad de la luz.

Para órbitas terrestres $W = 1400 W/m^2$ por lo que la presión solar se puede calcular como $P = 4.7 \times 10^{-6} N/m^2$. Este dato lo podemos considerar constante en toda la trayectoria.

La fuerza que ejerce esta presión se calcularía como:

$$f_{SRP} = (1 + k_r)\frac{A}{M} P \left(\frac{R_*}{r_*}\right) \qquad (III.14)$$

Siendo $1 + k_r$ el coeficiente de reflexión de la nave, A la sección transversal, P la presión por radiación solar calculada más arriba, R_* la distancia media entre el sol y la Tierra –equivalente a una unidad astronómica $UA = 149.597.870.700m$ – y r_* la distancia del sol a la nave. Para órbitas de satélites de un planeta siempre se dará que $R_*/r_* \sim 1$ por lo que la fuerza F_{SRP} será siempre del orden $F_{SRP} \sim 4.7 \times 10^{-6} \frac{A}{M}$ independientemente de la altura a la que nos encontremos.

Órbita	LEO	GEO
Dra	$4 \times 10^{-5} A/M$	$1.8 \times 10^{-13} A/M$
SRP	$4.7 \times 10^{-6} A/M$	$4.7 \times 10^{-6} A/M$

Tabla III.2. Ordenes de magnitud de perturbaciones por drag y por SRP

4. Ejes de referencia

En la astronomía existen infinidad de ejes de referencia distintos que responden a distintas necesidades. En lo que se refiere a este proyecto no buscamos cubrir todo el rango de posibilidades respecto a este punto. No obstante vamos a desarrollar dos marcos de referencia que cubran la posibilidad de un observador fijo en La Tierra, que gira con ella, y un observador fijo en La Tierra, que no gira con ella. Se trata por tanto de dos ejes de referencia, uno inercial y otro fijo a la tierra.

En cuanto al eje inercial hemos escogido el marco denominado ECI –Earth Centered Intertial- que únicamente tiene aceleraciones de translación del origen.

Dentro de esta categoría existen dos tipos de ejes, los que el plano XY se contiene en el plano ecuatorial, y los que el plano XY se contiene en el plano de la elipse de giro de la tierra alrededor del sol.

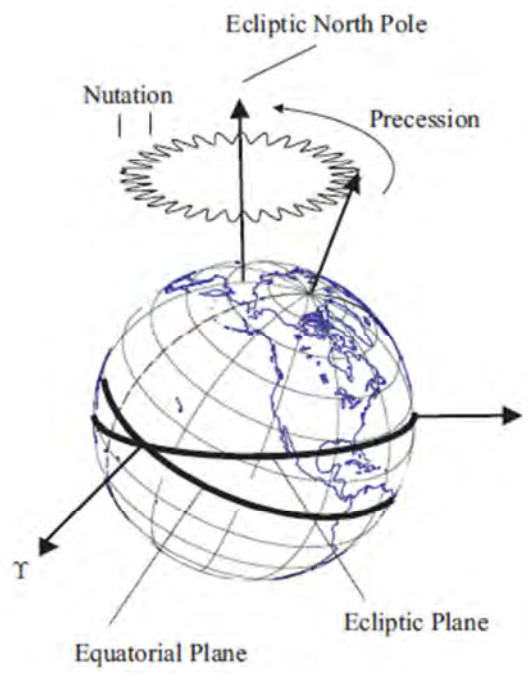


Figura III.5. Planos de referencia comúnmente utilizados para La Tierra

Nosotros tomaremos por defecto el mencionado del plano ecuatorial. Los ejes se construyen del siguiente modo: el eje X está contenido en el plano ecuatorial y apunta siempre hacia la constelación Aries ♈, punto que se puede considerar constante a lo largo del año de modo que este eje es fijo y siempre mira hacia el mismo punto, con independencia del giro de la tierra. El eje Y se construye de

modo que sea perpendicular al X y que además esté contenido en el plano ecuatorial. El otro marco de referencia cambia en este punto haciendo que el eje Y sea perpendicular a X pero esté contenido en el plano elíptico –plano que contendría al sol y la tierra- y ambos serían totalmente válidos. Finalmente el eje Z para nuestro marco de referencia se construirá de modo que sea perpendicular a X e Y. Puesto que el plano XY incluye el ecuador de la tierra, el eje Z cruzará por el centro del polo norte y no tendrá la inclinación característica de la tierra debido a la nutación –y que sí que tendría el que contiene el plano de la elipse-. Para más ajuste, se tomará como sistema de referencia estos tres ejes en la época J2000 debido a que la posición del polo, y por tanto del ecuador terrestre varía con el tiempo.

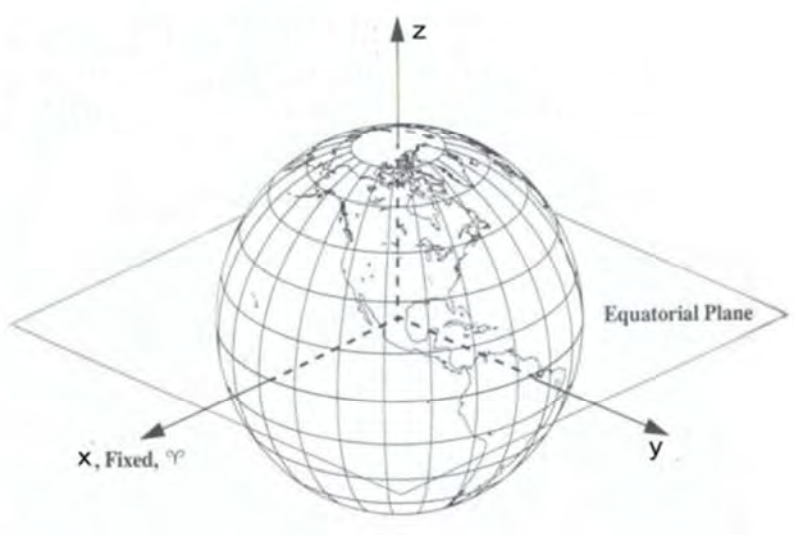


Figura III.6. Ejes de referencia con plano XY Contenido en el plano ecuatorial

El segundo marco de referencia que utilizaremos es uno no-inercial que se mueve solidario con la tierra denominado ECEF –Earth Centered Earth Fixed-. Además, el plano XY está también contenido en el plano ecuatorial lo que significa que el eje Z coincide en ambos sistemas de referencia. La conversión entre uno y otro se realizará únicamente en el plano XY como se puede observar en la siguiente figura:

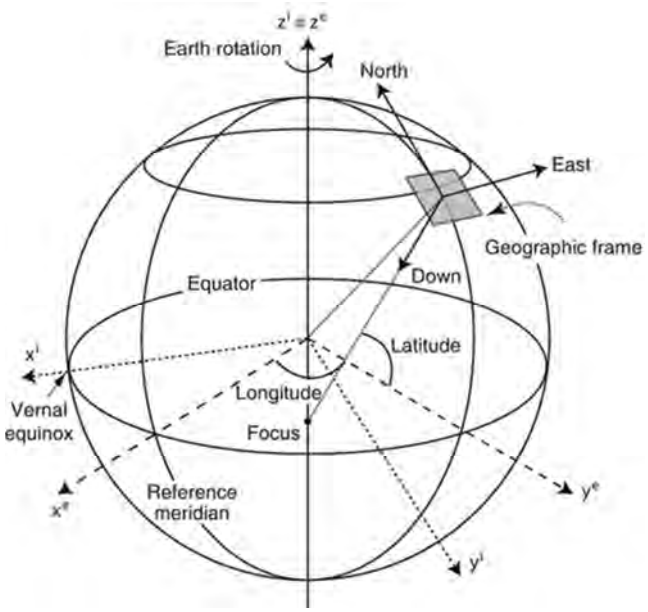


Figura III.7. Conversión entre marco inercial y marco no-inercial

Mientras que el sistema ECI lo componen los ejes fijos XYZ(i), el sistema de referencia ECEF lo componen los ejes XYZ(e). Conociendo el ángulo de giro se podrán calcular las 6 posiciones de interés.

En nuestro simulador, el sistema que viene por defecto es el ECI de modo que si no se especifica nada, todas las variables referidas a la posición y velocidad se calcularán con éste. Asimismo damos la posibilidad de transformar estas variables al sistema ECEF y únicamente en este sentido, de modo que para nuestro simulador sólo necesitaremos las matrices de rotación del ECI al ECEF.

Las matrices de rotación para posición y velocidad son las siguientes:

$$M_{pos} = \begin{pmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \qquad M_{vel} = \begin{pmatrix} -\omega\sin\theta & \omega\cos\theta & 0 \\ -\omega\cos\theta & -\omega\sin\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Siendo ω la velocidad de rotación del planeta – que en la tierra es $\omega = 7.292115 \times 10^{-5}\ rad/s$ - y θ el ángulo de giro que se calcula como $\theta = \omega\Delta t$.

En nuestro caso no nos es necesaria la matriz de rotación de aceleraciones puesto que únicamente trabajamos con posiciones y velocidades.

En conclusión, mediante estas matrices de rotación podremos ofrecer los resultados ajustados a cada uno de los marcos que el usuario de la GUI precise.

IV. Herramientas de implementación

Como parte importante del proyecto, la investigación sobre cuál era el objetivo que esperábamos obtener, nos ha llevado a la búsqueda de multitud de herramientas que nos ayudasen a conseguirlo. Se ha buscado maximizar los resultados no gastando tiempo ni recursos en construir aplicaciones o herramientas que ya estaban hechas por otros compañeros de la universidad, o como herramientas libres en internet.

Esto tiene su parte buena y su parte mala. Por un lado, el centrarnos en el objetivo que nos fijamos e intentar utilizar herramientas que ya existían nos ha hecho ahorrar mucho tiempo además de la seguridad de que esas utilidades que hemos adoptado funcionan bien en los entornos que vamos a necesitar. Nos ha facilitado el trabajo. Por otro lado, cada vez que usamos una de estas utilidades para nuestro proyecto creamos una dependencia con la misma, no pudiendo gestionar nosotros mismos la relación que se creará entre nuestra GUI y las herramientas de apoyo que utilicemos.

Como veremos a continuación algunas de estas herramientas son desde el punto de vista práctico imprescindibles.

Algunas de las herramientas que exponemos a continuación han servido de base para construir posteriormente lo que más se ajustaba a nuestras necesidades, siendo modificadas en gran parte con respecto al código inicial.

1. Software propio

Cuando hablamos de software propio nos referimos a programas creados desde la universidad con objetivos educativos y de acceso gratuito. Antiguos programas que diseñaron investigadores y otros creados por compañeros de carrera en sus respectivos proyectos.

Propagador desarrollado por la Carlos III

Esta función creada en Matlab por el investigador Filippo Cichocki del departamento de Ingeniería aeroespacial de la Universidad Carlos III nos sirvió como base y ejemplo de acoplamiento entre los datos que queremos incluir en la GUI y el propagador de Cowell.

Se trata de un Script de 200 líneas donde se crean estructuras con todos los datos de interés que posteriormente se enviarán al propagador. Además nos muestra cómo utilizar los resultados que arroja el propagador para su posterior uso en gráficas ya sean en 2D o en 3D. Se trata de una herramienta muy útil cuya estructura hemos tomado para nuestro simulador.

Propagador de Cowell

Se trata de una función creada en Matlab cuya finalidad es construir la ecuación diferencial del movimiento deseado para enviárselo al integrador. En nuestro proceso, éste se encontraría entre el propagador de la Carlos III y el integrador de la función diferencial.

En este caso se trata de una función que incluye 16 archivos que se encargan de construir la expresión analítica de la ecuación diferencial que cumple las condiciones preseleccionadas en el script desarrollado por Filippo Cichocki –que a su vez vienen de la GUI-. Vamos a ir viendo poco a poco todas las posibilidades que cubre este propagador.

El objetivo de este propagador es obtener cuatro expresiones:

- Expresión analítica de la ecuación diferencial
- Tiempo de integración
- Condiciones iniciales
- Opciones de integración

Para construir la expresión analítica de la ecuación diferencial recurrimos primeramente a la expresión $III.7$ que ya hemos desarrollado con anterioridad en:

$$\frac{d^2\vec{r}}{dt^2} + \frac{\mu_p}{|r|^2} \cdot \frac{\vec{r}}{|r|} = \vec{f}_{Pert}$$

Que también podemos escribir como:

$$\frac{d\vec{v}}{dt} = -\frac{\mu_p}{|r|^3}\vec{r} + \vec{f}_{Pert} \qquad (IV.1)$$

Considerando nuestro vector de estado $\vec{s}$ inicial como:

$$\vec{s} = (r_x, r_y, r_z, v_x, v_y, v_z)$$

Deberemos construir el vector $\dot{\vec{s}}$ para integrarlo. La estructura de este vector sería la siguiente:

$$\dot{\vec{s}} = \left(v_x, v_y, v_z, \dot{v}_x, \dot{v}_y, \dot{v}_z\right)$$

Siendo las tres primeras componentes conocidas y las tres últimas el vector resultante de resolver $(IV.1)$.

Para la resolución por parte del integrador –del que hablaremos más abajo- es necesario escribir una expresión analítica del tipo $\dot{\vec{s}} = f(t, \vec{s})$ por lo que el trabajo del propagador se reducirá únicamente a escribir los términos $\frac{\mu_p}{|r|^3}\vec{r}$ y $\vec{f}_{Pert}$ en función del vector $\vec{s}$.

El primer término únicamente posee el vector $\vec{r}$ además de la constante, por lo que es sencillo de plantear teniendo en cuenta que podemos escribir $\vec{r}$ como:

$$\vec{r} = s(1)\vec{u}_x + s(2)\vec{u}_y + s(3)\vec{u}_z = \vec{s}(1{:}\,3)$$

El segundo término lo dividimos en $\vec{f}_{Pert} = \vec{f}_{dist.masa} + \vec{f}_{n-cuerpos} + \vec{f}_{SRP} + \vec{f}_{Drag}$, todas ellas fuerzas ya explicadas.

Para la primera fuerza $\vec{f}_{dist.masa}$ debemos de calcular el potencial $U(r, \phi, \lambda)$ expresado más arriba siendo los coeficientes armónicos $\bar{C}_{n,m}$ y $\bar{S}_{n,m}$ conocidos para el caso de La Tierra. Para expresar este potencial en función de $\vec{s}$ tan solo debemos transformar las coordenadas esféricas r, ϕ, λ en cartesianas x, y, z las cuales podemos relacionar directamente con $s(1), s(2)\, y\, s(3)$. La conversión sería la siguiente:

$$r = \sqrt{x^2 + y^2 + z^2}$$
$$\phi = \arcsin\frac{z}{r}$$
$$\lambda = \operatorname{atan2}(y, x)$$

Para la perturbación creada al añadir más astros al cálculo $\vec{f}_{n-cuerpos}$ se utilizará la expresión desarrollada con anterioridad:

$$\vec{f}_{n-cuerpos} = \sum_{i=1}^{n} \vec{f}_{cuerpo-i} = \sum_{i=1}^{n} \mu_i \left(\frac{\vec{d}_i}{\left|\vec{d}_i\right|^3} - \frac{\vec{r}_i}{|\vec{r}_i|^3} \right)$$

El valor de $\vec{r}_i$ nos lo da la función SPICE de la que hablaremos más adelante de modo que es un dato conocido y el vector $\vec{d}_i$ se puede construir como $\vec{d}_i = \vec{r}_i - \vec{s}(1{:}\,3)$.

En la perturbación de Presión por radiación solar $\vec{f}_{SRP}$ la expresión que teníamos que utilizar era:

$$f_{SRP} = (1 + k_r)\frac{A}{M}P\left(\frac{R_*}{r_*}\right)$$

Como ha quedado dicho anteriormente, todos los datos son conocidos o definidos por nuestro vehículo y en el único que interviene el vector de posición es en el elemento r_* que definía la distancia del Sol a la nave. Para calcularlo en función del vector $\vec{s}$ utilizamos que $\vec{r}_* = \vec{d}_{Sun} - \vec{s}(1:3)$ con $\vec{d}_{Sun}$, el vector de posición del Sol respecto a nuestro cuerpo central que se calcula con la función SPICE.

Para terminar, la fuerza por Drag atmosférico $\vec{f}_{Drag}$ se calculaba como:

$$\vec{f}_D = \frac{1}{2}\rho\frac{A}{M}C_D V_r^2\left(\frac{-\vec{V}_r}{|V_r|}\right)$$

Los parámetros A , M y C_D vienen definidos por la nave. El vector velocidad $\vec{V}_r$ lo podemos construir como:

$$\vec{V}_r = s(4)\vec{u}_x + s(5)\vec{u}_y + s(6)\vec{u}_z = \vec{s}(4:6)$$

Y la densidad ρ se calculaba en función de la altura h que podemos expresarla como sigue:

$$h = \sqrt{s(1)^2 + s(2)^2 + s(3)^2} - R = \|\vec{s}(1:3)\| - R$$

Siendo R el radio de la tierra en la posición $\vec{s}(1:3)$ que nos lo proporciona otra función de SPICE.

$$\rho = \rho_0 e^{\frac{h-h_0}{H}}$$

Una vez realizado todo este proceso ya tenemos una expresión del tipo $\dot{\vec{s}} = f(t, \vec{s})$ que podemos enviar al integrador. Otros parámetros necesarios para la integración son el estado inicial $\vec{s}_0$ que se trata de un parámetro enviado desde la GUI, las opciones de integración que incluyen umbrales de valores para detenerla en cualquier momento o el número máximo de steps a realizar. Finalmente necesitamos los parámetros temporales de la integración que incluyen el instante inicial de integración t_0, el tiempo total que queremos integrar t y un valor Δt que le indicará al integrador cada cuantos segundos queremos que se actualicen las variables de salida.

2. Software de terceros

En el desarrollo de este proyecto se ha hecho uso también de software de terceras partes que se antojaban imprescindibles como herramientas para conseguir el mejor resultad. Algunas de ellas son de licencia libre y otras son de uso bajo pago de licencia.

CSPICE

Esta utilidad incluida en el paquete *Navigation and Ancillary Information Facility* comúnmente denominado NAIF creada por investigadores de la NASA que ponen a disposición de todos gratuitamente un completo sistema de información dinámica recolectada entre todas efemérides registradas por la organización.

Ofrecen a todos los investigadores un completo paquete de funciones con las que poder simplificar en gran medida los cálculos geométricos que conciernen al sistema solar.

Las funciones CSPICE están indicadas para uso tanto de investigadores en astronomía como para los ingenieros que requieran datos precisos en distancias cercanas a La Tierra. Se trata de una herramienta muy útil de la que hemos ya hablado en varias ocasiones para calcular distancias entre planteas, radios terrestres o matrices de giro.

La cantidad de funciones que incluye este paquete es inmensa y el uso que le damos es muy limitado para nuestra aplicación. Algunas de las funciones que usamos en nuestro código se muestran en la Tabla IV.1:

Función	Resultado
`cspice_str2et( Epoca )`	Tiempo de efmérides
`cspice_bodvrd( Cuerpo, 'GM', 1 )`	Parámetro de gravitación $GM = \mu[(km^3)/s^2\ ]$
`cspice_conics(ElmOrbit, et)`	Conversión entre vector de elementos orbitales y cartesianos
`cspice_bodvrd(Cuerpo, 'RADII', 3 )`	Ejes del elipsoide generado por un cuerpo
`cspice_pxform('J2000', Cuerpo, et)`	Matriz de rotación de un vector para el marco de referencia J2000

Tabla IV.1. Ejemplos de funciones incluidas en CSPICE

Existen versiones de estas funciones tanto para Matlab como para C, Fortran u otros lenguajes. Para el proyecto hemos utilizado la versión para Matlab de 64 bits.

Matlab

Como núcleo de cálculo hemos utilizado Matlab R2013a. No hace falta describir a este potente motor de cálculo.

Tanto el propagador como el integrador de trayectoria lo realizamos con este software. Se trataría de la función nativa de Matlab ODE45 definida como:

```
[T,Y] = ode45(odefun,tspan,y0,options)
```

Siendo `odefun` la ecuación diferencial analítica que hemos calculado con anterioridad en el propagador del tipo $\dot{\vec{s}} = f(t, \vec{s})$. El parámetro `tspan` tiene en cuenta los tiempos que definimos anteriormente y el `y0` serían las condiciones iniciales del problema. El último parámetro `options` es opcional, como su nombre indica, e incluiría todas esas leyes que queremos que evalúe para detener el cálculo a mitad de proceso.

Como salida nos devuelve la matriz Y que tendrá las mismas filas que el vector $\vec{s}$ pero con una columna por cada Δt de cálculo. Esta es la matriz que muestra la evolución de nuestro vehículo espacial cuando se le aplican todas las fuerzas anteriormente descritas. El vector T sería un vector de tiempos.

Dedicaremos un capítulo especial más adelante a la interacción entre nuestro simulador y el motor de Matlab –mediante la herramienta de Matlab Engine- para poder crear programas abiertos con comandos propios.

Eclipse java JMonkey

Para recrear la simulación 3D que se ofrece como objeto de salida del cálculo, hemos utilizado el paquete gratuito de librerías de JMonkey.

Se trata de una capa de personalización particular para recrear entornos en tres dimensiones sobre un proyecto de código abierto como es el IDE de Eclipse y para ser programado en lenguaje Java.

Una vez descargado su fork, tenemos preinstaladas todas las librerías y funciones para crear fácilmente entornos en 3D. Nuestro proyecto busca conseguir que el entorno generado en tres dimensiones se adecúe al resultado

de la simulación e intente mostrar la evolución de nuestra nave a lo largo del tiempo con el mayor realismo posible.

Uno de los objetivos del proyecto es conseguir un entorno tridimensional consistente con cualquier posible resultado que arroje el simulador.

Microsoft Visual Studio 2013

Se trata del IDE creado por Microsoft para desarrollar aplicaciones en entornos Windows. Se trata de un producto de pago pero que posee la opción de obtener una licencia gratuita para estudiantes e instituciones educativas.

Hemos utilizado esta herramienta en dos ocasiones. La primera para desarrollar la interfaz de usuario GUI de la que trata el proyecto. El lenguaje elegido ha sido .NET por su simplicidad a la hora de crear interfaces llamativas, simples e interactivas.

El otro motivo por el que hemos utilizado esta IDE es porque una de las interfaces que tenemos que utilizar, la que hace referencia a la capacidad de utilizar funciones nativas de Matlab desde nuestro simulador, programado en C/C++, denominada Matlab Engine, está específicamente recomendada para esta IDE y para este lenguaje por lo que aquí no hemos tenido muchas opciones más donde elegir.

GMAT

Este software creado por la NASA del que hablamos con anterioridad, además de sernos útil en la investigación previa a la realización de nuestro proyecto, también nos brinda la posibilidad de comparar nuestros resultados con los generados por él siéndonos de gran ayuda para la depuración de errores y para comprobar la calidad de nuestro código.

TinyXML

Se trata de unas librerías Open Source realizadas para C/C++ con las que gestionar archivos XML.

En nuestro proyecto y como explicaremos a continuación, las interfaces entre cada uno de los bloques se elaborarán mediante un archivo XML con toda la información. En el caso del bloque de la GUI, escrito en .NET, hemos utilizado librerías nativas del propio lenguaje. Por el contrario, para el bloque del simulador, escrito en C/C++, hemos preferido utilizar este paquete de funciones

que simplifican en gran medida la lectura y escritura en este tipo de archivos que hoy en día ya se han convertido en estándares.

Toolbox Google Earth

Se trata de un script realizado en el 'Institute for Biodiversity and Ecosystem Dyniamics' de la universidad de Amsterdam. Este script, creado en Matlab, nos permite crear ficheros con formato KML que se usa por defecto en Google Earth. Estos archivos contienen puntos y líneas con el objetivo de crear geometrías tridimensionales que posteriormente se cargarán en el visionado tridimensional del programa de Google. El uso que se le da en este proyecto es el de crear una línea tridimensional que una todos los puntos resultantes de nuestra simulación y que de modo automatizado genere el archivo .kml para observar la trayectoria.

V. Interfaces

Para la realización del proyecto hemos dado más peso a la facilidad de creación de cada uno de los bloques que lo componen, en detrimento de complicar las interfaces de los mismos.

Una interfaz es la conexión funcional entre dos sistemas o dispositivos. En nuestro proyecto las conexiones que se producen son entre bloques y la información que se transmite son datos. Dependiendo de en qué punto nos encontremos los datos serán de una naturaleza u otra. A continuación mostramos un esquema básico del recorrido del a información mostrando cada uno de los bloques y las interfaces que los conectan.

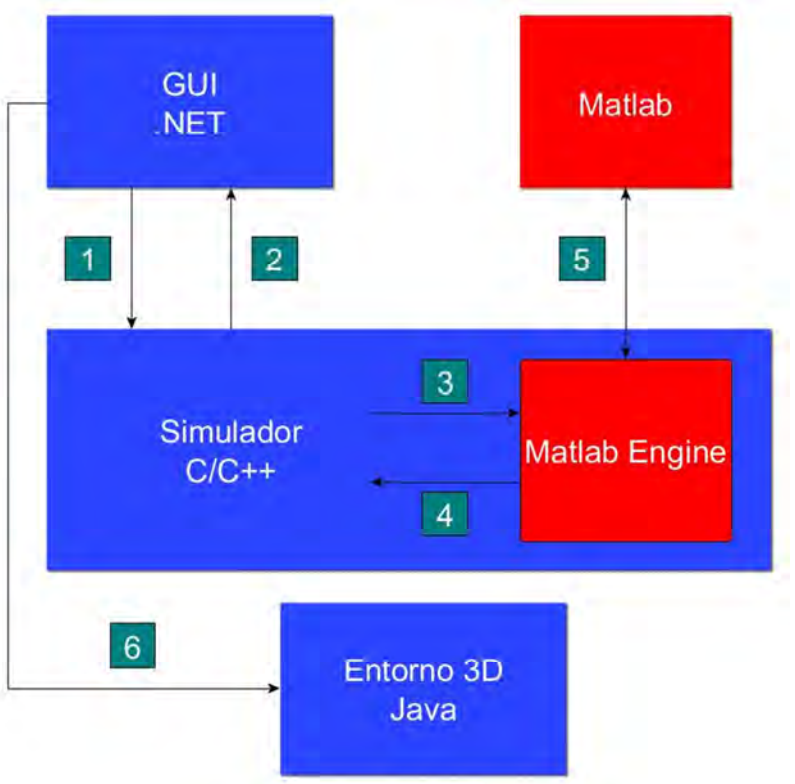


Figura V.1. Diagrama de flujo de información entre bloques.

Como podemos observar, entre todos los bloques existen interfaces para la transferencia de datos. En el plano de desarrollo, existirán tres tipos de interfaces:

- Creadas y gestionadas por nosotros
- Gestionadas por nosotros
- Gestionadas por terceros

Vamos a desarrollar cada una de estos canales de información detallando en cada uno de los 6 casos qué método hemos escogido y por qué motivo para que se produzca la transferencia de datos.

1. Interfaz GUI → Simulador

El método de intercambio de datos se realizará mediante un archivo temporal con formato XML. Se tratará por lo tanto de una interface creada y gestionada por nosotros. Nosotros especificamos que el intercambio entre programas se haga por archivo y además somos capaces de personalizar ese archivo.

Una vez realizados todos los ajustes de la simulación en la interfaz de usuario, procederemos a iniciar la simulación. Cuando ejecutemos esta acción el proceso que se realizará por detrás será la escritura de un archivo XML y la posterior ejecución de un comando que iniciará el simulador. En el simulador se leerá ese archivo XML y se cargará toda la información necesaria para iniciar la simulación deseada.

La estructura del XML que conecta ambos software es la siguiente:

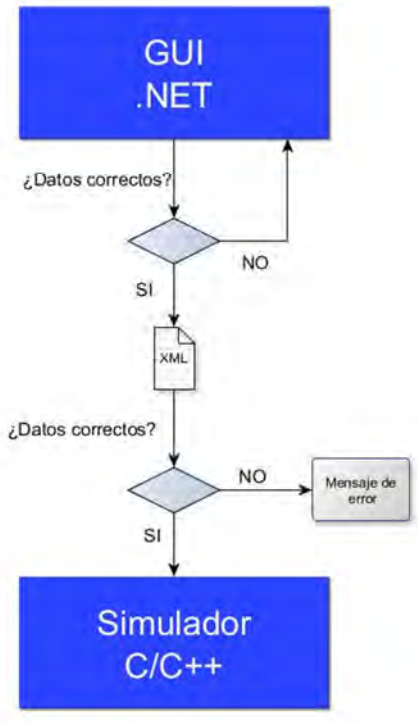
```
<?xml version="1.0" encoding="utf-8"?>
<mision>
    <dir>
      ...
    </dir>
    <steps>
      ...
    </steps >
    <naves>
      ...
    </naves>
    <propagadores>
      ...
    </propagadores>
    <motores>
      ...
    </motores>
    <outputs>
      ...
    </outputs>
    <variables>
      ...
    </variables>
</mision>
```

El archivo XML contendrá un nodo principal con la etiqueta `<mision>` que a su vez contendrá distintos nodos. El primer nodo `<dir>` hace referencia a la localización del programa en el disco duro. Esto es necesario para más tarde

poder realizar operaciones con el Matlab Engine así como guardar archivos con datos de salida. A continuación en el nodo `<steps>` introduciremos todas las etapas de la integración siendo posible encadenar varias consecutivas. Cada step contendrá un tiempo, una condición de parada y un propagador asociado. El tercer nodo `<naves>` incluye todas las naves que hayamos creados en la interfaz. Para cada nave tendremos parámetros como condiciones iniciales, características físicas de la misma o modelos en 3D para su visualización.

El nodo `<propagador>` contendrá toda la información en referencia al método de cálculo que se emplea. A su vez contendrá otros nodos con información sobre el tiempo de simulación, el modelo de fuerzas a incluir o el step de cálculo entre otras. El nodo `<motores>` guarda información relacionada con los propulsores del vehículo para realizar maniobras.

Mediante la etiqueta `<outputs>` gestionaremos las salidas de nuestro simulador. Dentro de esta etiqueta se podrán encontrar nodos de tipo archivo, gráfica, groundtrack o simulación 3D. Para cada uno de estos nodos se detallará todas las características que lo definen y que posteriormente se usarán para generar los archivos de salida. Dentro del nodo `<variables>` incluiremos todas las variables que, una vez ejecutado el código, se evaluarán para poder obtener los parámetros de salida que buscamos en los outputs. Mediante la GUI se pueden gestionar estas variables y añadir nuevas.



Como ya hemos explicado en otros puntos, queremos realizar un software consistente que sea capaz de prever si el usuario ha introducido mal alguno de los datos, o simplemente no lo ha introducido. Un ejemplo sería intentar iniciar la simulación con el parámetro de 'Tiempo total de simulación' con valor `NULL`. Esto generaría un error en todo el proceso que viene a continuación.

Para evitar esto, dotaremos a nuestro programa de una doble verificación. Tanto al salir de la GUI como al entrar en el simulador se verificará que el archivo contiene todos los datos necesarios para realizar la simulación.

El motivo por el que se hace una doble comprobación cuando en principio una sola bastaría es que la GUI también trabajará gestionando misiones con este mismo formato XML. Se podrán guardar y volver a cargar en el futuro misiones configuradas con este mismo esquema de etiquetas. Y si se guarda una misión no se realizará la comprobación de que todos los datos están bien, debido a que se presupone que el usuario no la ha configurado entera. Es por este motivo por el que se comprueban los datos a la entrada del simulador dado que algún usuario se podría ver tentado de colocar el archivo guardado previamente en la interfaz y lanzar directamente el simulador, proceso que sería del todo correcto siempre y cuando la estructura del XML sea la correcta.

Así mismo, se ha dotado a la GUI de la capacidad de adelantarse y comprobar si un dato introducido es o no es correcto. Así si introducimos una letra en un campo que requiere un valor decimal, el programa nos mostrara un mensaje de error y cargará el valor que había previamente.

2. Interfaz Simulador → GUI

Una vez terminada la personalización de la GUI iniciaremos el simulador. El proceso que viene a continuación es el siguiente. Al presionar el botón de *Play* la GUI guardará toda la información en un archivo de extensión XML en la carpeta denominada `/temp` y ejecutará el simulador. Una vez lanzada la petición de ejecución del simulador la GUI se cerrará. El simulador cuando se inicia, carga el archivo generado en la carpeta `/temp` y lee todos los datos que contiene. Una vez clasificados los datos y guardados en variables, el simulador lanza el código mediante la ejecución de un archivo Matlab que utiliza como interfaz el Matlab Engine, explicado a continuación. Una vez terminado el cálculo, el simulador guarda todos los datos Outputs requeridos y lanza una petición de ejecución de la GUI. Para que la GUI diferencie entre esta petición que viene del simulador, y una petición inicial del usuario, hemos incluido el parámetro "1" en los argumentos del comando de petición de ejecución. Así pues, la GUI cuando reciba una petición de ejecución sin ningún parámetro sabrá que debe iniciarse por defecto, mientras que si existe un parámetro "1" en la línea de comandos, se iniciará en modo Outpus.

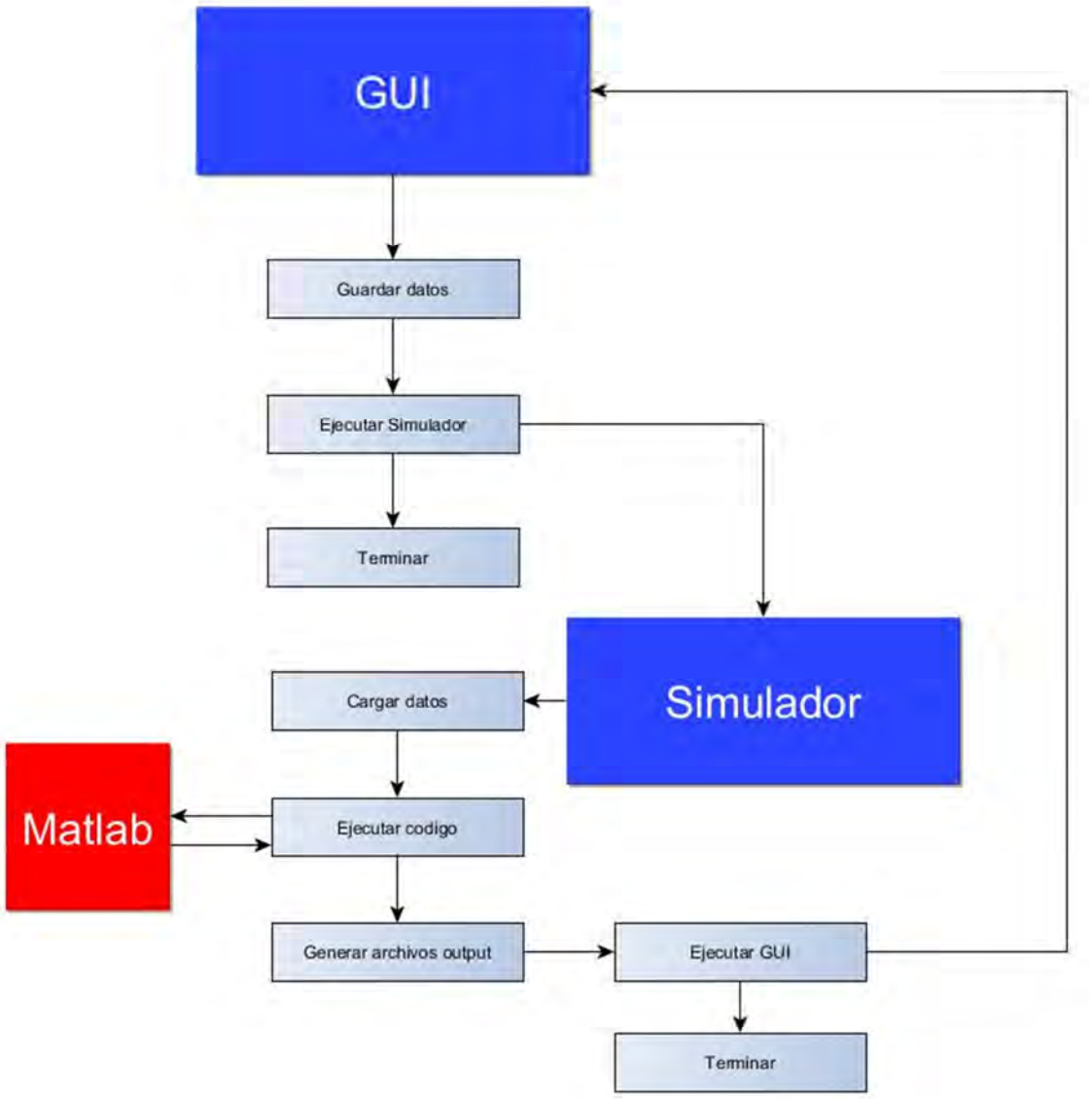


Figura V.2. Diagrama de flujo de información GUI/Simulador

Una vez iniciado en modo Output, la GUI sabrá qué archivos debe cargar y qué información debe mostrar como respuesta a la llamada del simulador.

De modo que para concluir esta interfaz podremos decir que únicamente consta de un parámetro en línea de comandos, con el consiguiente modulo programado en la GUI para hacerlo entendible.

3. Interfaz Simulador → Matlab Engine

Para explicar esta interfaz primeramente debemos explicar qué es el Matlab Engine.

El Matlab Engine es un conjunto de librerías que ponen a nuestra disposición los creadores de Matlab con el fin de poder utilizar comandos nativos de Matlab dentro de nuestros propios programas. En nuestro caso, estos comandos se utilizarán dentro del simulador, escrito en C/C++. Para poder utilizar Matlab Engine es necesario realizar un proceso previo de instalación, descrito en este mismo proyecto en el anexo 2. Una vez realizado este proceso podemos utilizar un conjunto de funciones dentro de nuestro código para enviar comandos a Matlab, incluso si éste no está en ese momento iniciado.

Es importante resaltar que las librerías de Matlab Engine no ejecutan comandos de Matlab en nuestro código, sino que hace de interface entre nuestro código y el propio motor nativo de Matlab que se ejecuta en segundo plano como un servicio. De modo que se trataría de una interfaz más desde la que podemos enviar información a Matlab desde nuestro simulador y del mismo modo recibirla.

Una vez realizada la instalación del Matlab Engine, y habiendo incluido la librería `#include "engine.h"`, el comando que nos permite enviar información hacia Matlab es:

```
engEvalString(ep, comando);
```

Siendo `ep` una variable de tipo `Engine*` previamente inicializada hacia el Matlab Engine y `comando` una variable de tipo `String*` que contiene la instrucción nativa en lenguaje Matlab. Las posibilidades de `comando` son infinitas. Podemos introducir todas las funciones que introduciríamos normalmente en la consola de Matlab.

Se trataría pues de una interfaz creada por Matlab pero gestionada por nosotros mismos teniendo la capacidad de personalizar el comando a nuestro placer y tantas veces como queramos. Esta es una diferencia muy grande con respecto a las interfaces por archivo XML. En este caso podremos enviar tantos comandos como deseemos en una misma ejecución mientras que en el caso de archivo XML únicamente haremos una conexión con toda la información.

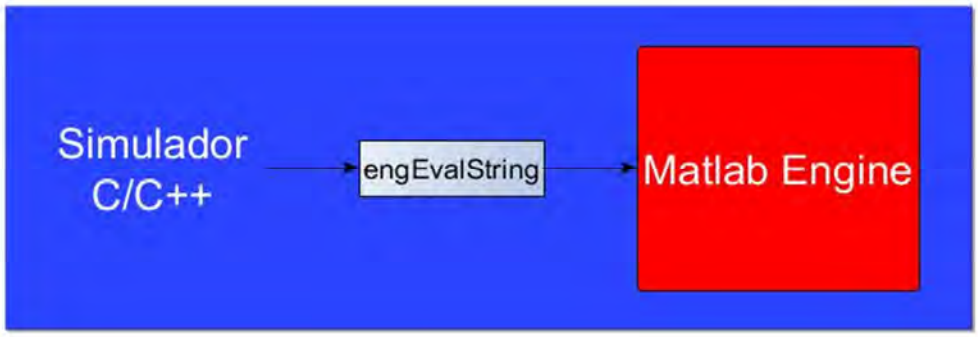


Figura V.3. Diagrama de interface entre Simulador y Matlab Engine

4. Interfaz Matlab Engine → Simulador

Se trataría de la interfaz opuesta a la anterior. El Matlab Engine enviará información a nuestro simulador escrito en C/C++. Para utilizar esta interface necesitaremos exactamente lo mismo que para el de sentido contrario. Una vez instalado el Matlab Engine y habiendo incluido en el proyecto el comando `#include "engine.h"`, podremos utilizar la función:

```
engOutputBuffer(ep, buffer, BUFSIZE);
```

Donde `ep` es la misma variable de tipo `Engine*` que habíamos inicializado con anterioridad, `buffer` una variable de tipo `Char*` donde se guardará la respuesta y `BUFSIZE` el tamaño máximo –o longitud- que tendrá la variable `buffer`.

Ejecutando este comando podremos entonces guardar en la variable `buffer` la respuesta al comando Matlab. Cabe destacar que en este caso la transferencia se hace mediante buffer puesto que la respuesta de un comando Matlab puede ser de un tamaño enorme –imaginemos que le pedimos que muestre una matriz con cientos de columnas y cientos de filas- de modo que habrá que gestionar el tamaño del buffer.

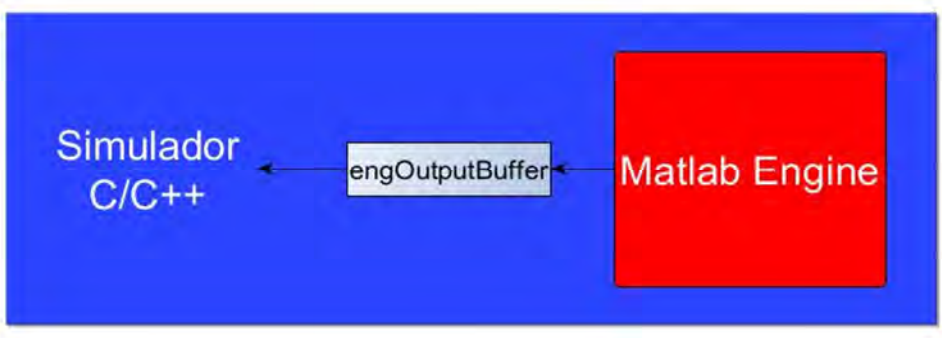


Figura V.4. Diagrama de interface entre Matlab Engine y Simulador

5. Interfaz Matlab Engine ↔ Matlab

Es la única interfaz que ni hemos creado ni hemos gestionado. Cuando realizamos el set-up del Matlab Engine –explicado en el anexo 3- activamos un proceso que se ejecuta en segundo plano y que hace de interface entre la librería `"engine.h"` que hemos incluido en nuestro proyecto y el propio Matlab. Como hemos explicado con anterioridad, cuando incluimos la librería `"engine.h"` en nuestro proyecto de C/C++ no estamos incluyendo las funciones nativas de Matlab, sino la funciones de interface entre Matlab y nuestro proyecto. De modo que básicamente esta librería que incluimos sería la interfaz entre nuestro proyecto y Matlab. Ni la hemos creado nosotros ni tampoco la gestionamos nosotros, pero sabemos que existe y que está trabajando cada vez que lanzamos un comando hacia el Matlab Engine.

6. Interfaz GUI → Entorno 3D

Una vez realizada la simulación con todos los cálculos necesarios y habiéndose guardado todos los archivos outputs requeridos, volvemos a la interfaz GUI. Estando en la interfaz GUI podremos ver y gestionar todos los outputs generados en el simulador y en particular los concernientes a la simulación en 3D.

La simulación en 3D se realiza en un programa aparte creado en JAVA y que contiene todos los objetos y texturas que la GUI puede requerir. De modo que la interfaz entre la GUI y el simulador 3D se realizará mediante argumentos en una línea de comando.

La GUI lanzará un comando de ejecución del simulador con determinados parámetros. El orden de cada parámetro está fijado de modo que de cambiarse alguno de los parámetros de posición no sería posible realizar la simulación. Este orden lo gestiona la propia GUI por lo que el usuario no debe por qué conocerlo ni preocuparse. Los parámetros que se pasan y el orden son los siguientes:

#	Tipo	Uso
1	String	Nombre del archivo para guardar el archivo de video .avi
2	Integer	Tiempo total de simulación
3	Integer	Cuerpo central
4	Integer	Factor de tiempo

5	Integer	Resolución X
6	Integer	Resolución Y
7	Boolean	Grabar video
8	Float	Escala del modelo
9	Integer	Modelo de nave
10	Float	Parte R del color de la órbita
11	Float	Parte G del color de la órbita
12	Float	Parte B del color de la órbita
13	Float	Ajuste de horas respecto meridiano cero
14	Float	Ajuste de minutos respecto meridiano cero
15	Float	Ajuste de segundos respecto meridiano cero

Tabla V.1. Orden de los parámetros y tipo de variables para el simulador 3D

El simulador en 3D realizará una conversión de todos los parámetros introducidos por línea de comando –que por defecto son de tipo `string`- a los diferentes tipos que se especifican en la tabla para después comenzar la simulación.

Es importante decir que la simulación que se puede visualizar se está generando y renderizando en tiempo real de modo que los niveles de memoria que se necesitan para esta aplicación deben ser altos.

7. Interfaces de datos

Todos los datos que sean calculados en el simulador son a continuación guardados en archivos de texto plano con una línea de puntos para cada variable seleccionada. El método que se utiliza para guardarlos consiste en enviar un comando `save` a Matlab con las variables necesarias para cada caso y añadir al final el parámetro `-ASCII` para que se guarde en texto plano. El nombre del archivo será el nombre del objeto Output del árbol de elementos en la GUI, por lo que éstos deberán ser únicos.

Por tanto cuando salimos del simulador deberemos de tener guardados en archivos de texto todas los puntos de todas las variables que se han requerido desde la GUI. Las carpetas donde se guardarán serán según el tipo de datos `/temp` u `/outs` –todo el sistema de carpetas se explica más adelante en el Capítulo VI.5- y del mismo modo, al reiniciar la GUI en modo outputs las carpetas donde se buscarán los archivos para cargarlos serán éstas.

8. Interfaces de leyes de control

Una vez creadas todas las herramientas para realizar propagaciones de órbitas y verificados sus resultados en el Capítulo VII se ha decidido implementar otra utilidad mas que es complementaria a lo que ya había. Se trata de un módulo de leyes de control que gobiernen el vehículo espacial bajo las condiciones que el usuario prefiera. Se desarrollaría como un apartado experimental del programa que nos permite ejecutar en cada ciclo un diagrama de bloques con varios parámetros de entrada y varios de salida. En este diagrama de bloques se deberá implementar la ley que queremos que controle el vehículo que dependerá de los parámetros de entrada y producirá un efecto en la nave mediante la adición de un empuje.

Los diagramas de bloques se deberán realizar con la herramienta de Matlab Simulink. Mediante esta herramienta es posible crear fácilmente leyes lógicas que consigan el efecto deseado para ser aplicado a la nave.

Esta implementación permitirá al usuario realizar cualquier tipo de maniobra, controlar cualquier parámetro y realizar los ajustes necesarios a la misión para conseguir el objetivo. Las opciones son infinitas y en este informe únicamente detallaremos cómo se implementa y estudiaremos un caso en particular en el Capítulo VIII. Se trata pues de una herramienta muy potente y requiere de conocimientos avanzados de programación en Matlab.

La ley de control consta de dos archivos. El primero, el código de interface entre el integrador y el modelo Simulink se encuentra en el archivo `/x64/matlab/GeneralPurposeOrbitPropagator/leyes_control.m`, se trata de una función definida del siguiente modo:

```
function [ empuje ] = leyes_control(et, state, mu)
```

Dispondremos por tanto a cada instante de las variables tiempo `et`, estado `state` y la constante planetaria `mu`. Mediante un correcto uso de éstas podremos enviar información al módulo de Simulink que constará de varios parámetros de entrada y varios de salida, como se muestra en la imagen:

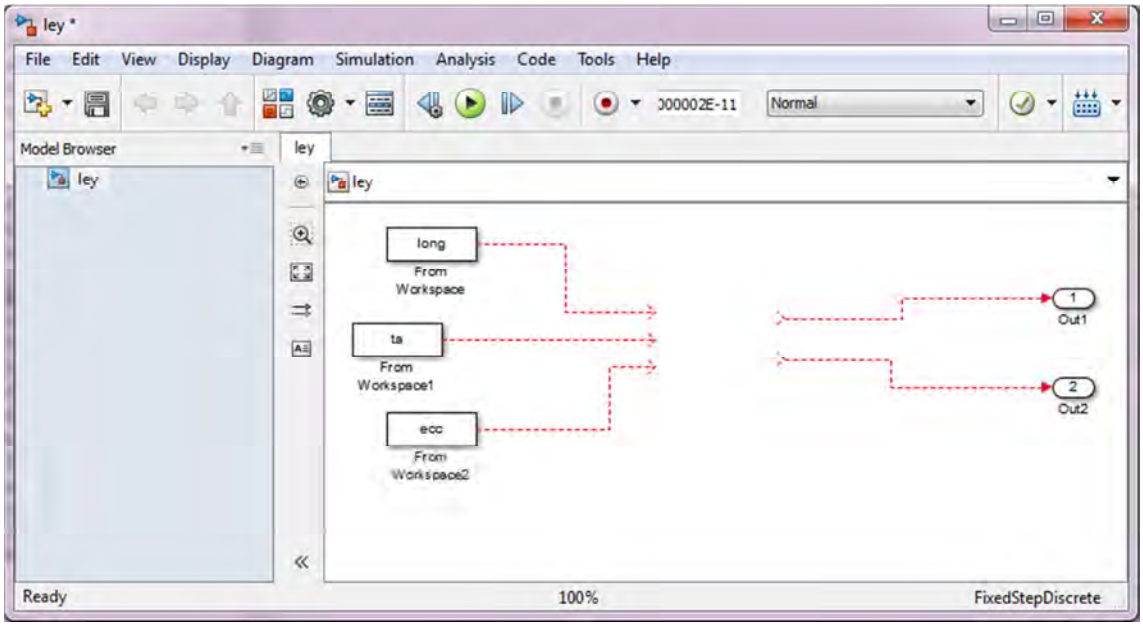


Figura V.5.Ejemplo de in/outs en diagrama de bloques

Este diagrama de bloques –contenido en el archivo `ley.mdl`- podremos modificarlo como más convenga para la ley que deseamos desarrollar. En la imagen se muestran tres variables de entrada y dos de salida. Las variables de entrada son bloques 'From Worksapce' mientras que las variables de salida son bloques 'Out'. Entre las variables de entrada y las de salida se deberá implementar la ley. Para ello podremos utilizar todas las herramientas que nos pone a nuestra disposición Simulink y en función de la habilidad del usuario conseguiremos leyes más eficaces y más precisas.

Para conectar el código del archivo `leyes_control.m` con el diagrama de bloques del archivo `ley.mdl` se han implementado dos funciones de interface. La primera se utiliza para asignar un valor a una variable de entrada. El procedimiento es el siguiente: primero es necesario crear en el diagrama un bloque del tipo 'From Workspace' y darle doble click para asignarle un nombre al parámetro de entrada en el campo de texto 'Data':

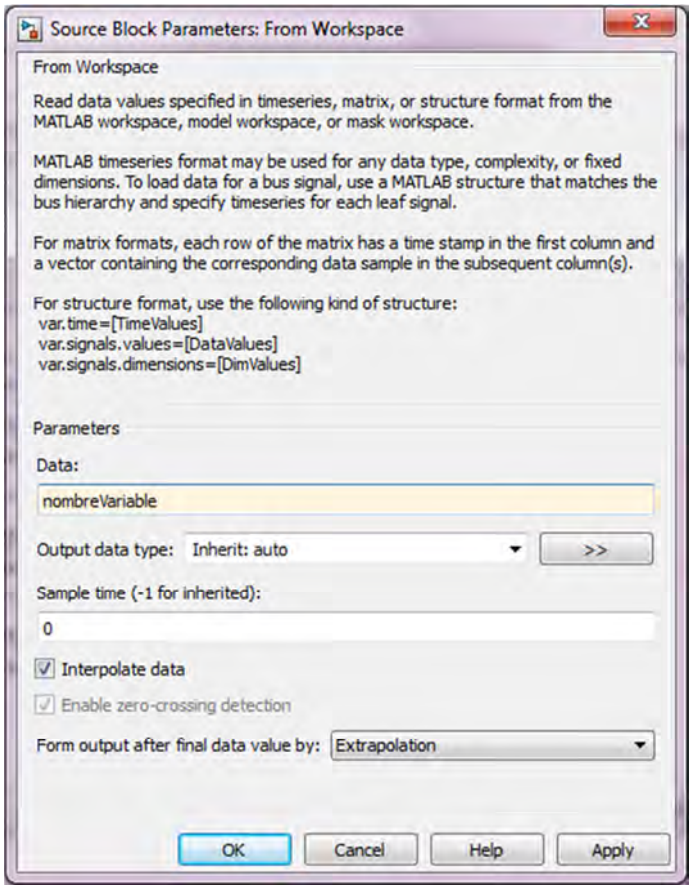


Figura V.6. Propiedades de bloque From Workspace

Una vez que tenemos este bloque, para asignare un valor desde el código deberemos utilizar la función:

```
set_variable_in_ley_control( var, nombreBloqueFromWorkSpace )
```

En nuestro caso, si quisiésemos asignar el valor de la variable 'x' a este bloque deberíamos utilizar la función del siguiente modo:

```
set_variable_in_ley_control(state(1), 'nombreVariable');
```

Este proceso debería de repetirse para cada una de las variables de entrada que deseemos. Es importante hacer notar que estas variables de entrada pueden ser cualquiera que derive del estado y el tiempo como por ejemplo la longitud, latitud, semieje mayor, excentricidad etc, tan solo es necesario realizar al conversión previa en el archivo `leyes_control.m`.

Una vez que hemos iniciado todas las variables de entrada del modelo Simulink podremos realizar la simulación del modelo que producirá unas salidas. En

general trabajaremos con dos salidas. La primera denominada `flag` que será cero cuando el empuje sea nulo, y uno cuando queramos añadir empuje al movimiento. La segunda variable es el módulo de la fuerza que deseamos añadir al movimiento. Estas dos variables han sido seleccionadas arbitrariamente en función del modelo que se ha implementado. El usuario final es libre de cambiarlas o añadir otras nuevas como más le convenga.

Para añadir salidas al diagrama utilizaremos bloques 'Out'. Una vez añadidos estos bloques al diagrama la función implementada para realizar la simulación es la siguiente:

```
[ s ] = get_variable_out_ley_control( nombreDiagrama )
```

Siendo `nombreDiagrama` el nombre del archivo Simulink que estamos implementando. En nuestro caso el archivo se llama `ley.mdl` de modo que la función se llamaría del siguiente modo:

```
y = get_variable_out_ley_control('ley');
```

La llamada a esta función implicará que el modelo Simulink realizará su ciclo de simulación con los parámetros de entrada previamente establecidos y al terminar guardará las salidas en la variable `y`.

La variable `y` será un vector con tantas entradas como bloques 'Out' se incluyan en el modelo de modo que para nuestro caso accederíamos a los valores de salida del siguiente modo:

```
 flag = y(1);
fuerza = y(2);
```

Así pues ya estamos en disposición de utilizar las salidas del modo que mejor nos convenga para nuestra propagación.

Antes de pasar a un ejemplo concreto es interesante hacer notar algunas consideraciones al respecto.

La primera de todas es la concerniente al tiempo de simulación. Según está estructurado el código, a cada ciclo de cálculo se realizará una simulación del modelo de Simulink. Esto puede provocar que el tiempo de cálculo de una simulación simple se multiplique exponencialmente puesto que para cada paso

se deberá de ejecutar la simulación lo que implica la compilación y ejecución del modelo. Para evitar que el tiempo se alargue se ha propuesto una ejecución de la simulación cada determinado tiempo mediante el uso de variables `persistent` o también conocidas como estáticas. En el ejemplo más adelante se puede observar cómo realizar esta implementación con el objeto de reducir el tiempo de simulación.

Con respecto a estas variables otra consideración es que únicamente poden ser utilizadas en el archivo `leyes_control.m` y no en el diagrama de bloques. Esto es debido a que el modo de simulación elegido implica una compilación del diagrama para cada instante lo que imposibilita el uso de este tipo de variables estáticas.

VI. Desarrollo e implementación

Una vez descritas todas las características que debe de cumplir cada bloque así como las interfaces que deben existir entre bloques vamos a pasar a desarrollar las aplicaciones de cada uno de los elementos que componen el software.

Con objeto de no llenar el informe de líneas de código –siendo posible consultarlo en los archivos adjuntos del proyecto- vamos a pasar a resumir para cada uno de los bloques la metodología utilizada y las características del programa para cada lenguaje de programación.

1. GUI

Este programa se ha realizado en .NET como hemos comentado con anterioridad. La organización de este módulo se ha realizado mediante clases. Las clases coinciden con cada uno de los puntos mostrados en el archivo XML de interfaz entre la GUI y el simulador. En la *Tabla VI.2* podemos observar todas las clases. La clase primaria es `recMision` que, imitando la estructura del XML, abarca todas las demás clases.

Nombre	Descripción
`recMision`	Clase principal que tiene a su vez varias `recNave`, varios `recPropagador`, un `recSS`, un `recOutput`, y un `recVariablesSalida`. Alguna de las funciones más útiles de esta clase es `Public Function getXMLMision()`, que nos devuelve una cadena de texto con el archivo XML que utilizaremos como interfaz o para guardar nuestra misión.
`recNave`	Contiene variables de las características de la nave como condiciones iniciales, masa, sección, modelo de nave 3D, etc.
`recPropagador`	Contiene variables de las características del propagador como el tipo de integrador, el tiempo de integración, el step o el modelo de fuerzas que se utilizará.
`recMotor`	Contiene el módulo de las fuerzas así como las direcciones en las que se aplicarán.
`recSS`	Formado por varios `recPlaneta.`
`recOutput`	Compuesto por varios `recOutputArchivo`, `recOutputGrafica`, `recOutputGroundTrack` y `recOutput3D`.

`recVariablesSalida`	Contiene listas de nombres de variables junto con comandos y con las características de las variables como dimensiones o unidades.
`recOutputArchivo`	Lo conforman variables que definen cada uno de los outputs de tipo archivo como el número de puntos, precisión, etc.
`recOutputGrafica`	Abarca variables que definen cada uno de los outputs de tipo grafica como el número de puntos, variables en el eje de ordenadas o el de abcisas.
`recOutputGroundTrack`	Incluye variables que definen cada uno de los outputs de tipo GroundTrack como la nave que muestrear.
`recOutput3D`	Engloba variables que definen cada uno de los outputs de tipo 3D como la resolución de video o el modelo de nave 3D entre otras.
`recPlaneta`	Contiene varias `recLuna` además de otras características que definen al planeta como la constante, el achatamiento u otros.
`recLuna`	Contiene características que definen a la luna como la constante, el achatamiento u otros.
`misionStep`	Clase que indica las maniobras que se realizarán en distintos steps. Si únicamente se requiere una propagación, solo existirá un objeto de esta clase.

Tabla VI.2. Clases principales creadas para la GUI

En definitiva, la GUI básicamente gestiona el manejo de todas estas clases y de las variables que las constituyen. Mediante campos de textos, listas de texto, imágenes, checks o desplegables se pueden modificar todas las variables que requiere el problema, consiguiendo una misión personalizada. Para facilitar el cambio entre todas las clases, se ha introducido un árbol de nodos en el menú de la izquierda del programa con el que poder gestionar cada una. Pinchando en cada nodo, se cargarán a la derecha todas las características y variables de esa clase para hacer las modificaciones que se deseen. También mediante el click derecho del ratón hemos habilitado menús de gestión de clases pudiendo añadir o eliminar nuevos objetos de cada tipo de clase al conjunto de la misión.

Mediante una barra superior introducimos la opción de gestionar misiones –nueva misión, guardar misión, abrir misión- así como el iniciar el simulador.

La distribución de cada uno de los elementos en la interfaz sería la siguiente:

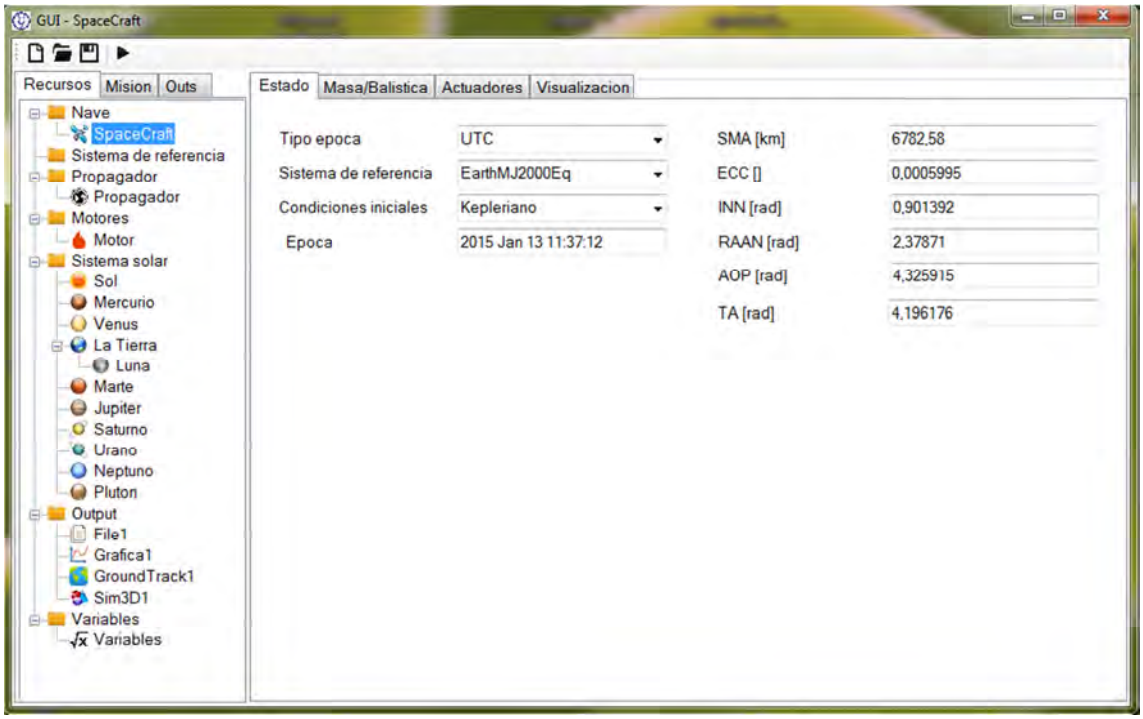


Figura VI.1. Vista previa de la interfaz de usuario

Hemos incluido además el redimensionamiento de todos los elementos que se muestran por pantalla cuando se agranda o disminuye la ventana quedando siempre todos los campos alineados los unos con los otros y con los marcos de la ventana.

Existen dos modos de ejecutar la GUI. El primero, el comúnmente utilizado, al abrir el ejecutable se iniciará el programa en modo Default lo que significa que preestablecerá unos datos por defecto que están introducidos dentro del código. No obstante y como hemos explicado en el apartado de interfaces, se ha habilitado la capacidad de iniciar el programa en un modo Outputs con distinta secuencia de iniciación. Cuando iniciamos en este modo no se cargan los datos preestablecidos dentro del código, sino que se cargan los datos escritos dentro del archivo data_in.xml que hemos usado anteriormente de interface entre el la GUI y el simulador. Una vez que están cargados estos datos se procederá a analizar todas los outputs que se habían pedido y cargar todos los archivos de texto que se espera que existan en las carpetas `/temp` y `/outs` generados por el simulador. Este modo de entrada es el que utiliza el simulador para llamar a la GUI cuando éste ya ha terminado y así poder mostrar las salidas.

Este sería el resumen de la GUI que contiene más de 4000 líneas de código que se pueden consultar en los documentos adjuntos en la carpeta `src/gui/`.

2. Simulador

La elaboración de este programa se ha realizado en C/C++ y se ejecuta en una ventana de comando. Puesto que se trata de un programa de cálculo, el procedimiento es secuencial. En este caso no se ha entrado en complicaciones de clases debido a que las operaciones con los datos no están sujetas a peticiones del usuario, sino a la información que está disponible en el archivo XML de interfaz proveniente de la GUI. Así pues, básicamente la secuencia que ejecuta el programa es la siguiente:

#	Función	Descripción
1	`datosEntrada()`	Abre el archivo XML de interfaz y lee cada una de los nodos y etiquetas del archivo para guardarlas posteriormente en variables. Como se ha explicado con anterioridad si alguno de los datos no es correcto, no se procederá a las siguientes funciones y el programa se cerrará con un mensaje de error.
2	`iniciarMatlabEngine()`	Se inicia la interfaz con el Matlab Engine para enviarle comandos.
3	`ejecutarCodigo()`	Se ejecuta el script de Matlab. Mediante comandos vamos enviando todas las variables a estructuras dentro del 'workspace' virtual de Matlab y posteriormente se ejecuta el Script. Este script se detallará más adelante.
4	`guardarArchivos()`	Por cada uno de los outputs de tipo archivo seleccionados en la GUI se generará un archivo de tipo texto plano con las variables deseadas. Además se ofrece al usuario la opción –mediante un check en la GUI- de guardar los datos también en un archivo con formato Matlab para su posterior uso.
5	`guardarGraficas()`	Por cada uno de los outputs de tipo gráfica seleccionados en la GUI se generará un archivo de tipo texto plano con las variables X e Y deseadas. Se presupondrá una precisión doble –16 dígitos- por defecto.
6	`guardarGroundTrack()`	Por cada uno de los outputs de tipo ground track seleccionados en la GUI se generará un archivo de tipo texto plano con las variables latitud y

		longitud referentes al movimiento calculado.
7	`guardarSim3D()`	Por cada uno de los outputs de tipo 3D seleccionados en la GUI se generará un archivo de tipo texto plano puntos X Y Z describiendo el movimiento del satélite alrededor del cuerpo.
8	`cerrarMatlabEngine()`	Se cierra la interfaz con el Matlab Engine.
9	`system("gui.exe 1");`	Se ejecuta de nuevo la GUI con el valor 1 en los parámetros para que ésta se inicie en modo Output.
10	`Return`	Se finaliza la ejecución.

Tabla VI.3. Secuencia de ejecución del simulador

Además de todas estas funciones descritas se han dispuesto otras tantas de ayuda para realizar de manera más sencilla la interfaz entre el programa en C/C++ y el Matlab Engine y funciones de conversión entre tipos –como por ejemplo el tipo `std::string` y el tipo `char*-` logrando con ellas una mayor eficiencia a la hora de realizar las acciones para las que realmente hemos diseñado el programa.

La función más importante del código es `ejecutarCodigo()` que busca organizar la información cargada del archivo XML para enviarla al Matlab. Este proceso se lleva a cabo mediante la creación de estructuras. En este caso son cuatro las estructuras que se crean. La primera estructura denominada `nave` incluye todos los datos referentes a la nave como condiciones iniciales, características físicas etc. La segunda estructura es la del `propagador` que contiene toda la información referente al cálculo que el Matlab va a realizar a continuación. La estructura `motores` contiene todas las fuerzas extras que podemos aplicar en un punto de la órbita. Finalmente en la estructura `step` incluimos para cada tramo de la misión la condición de finalización que podrá ser completar el tiempo, llegar al perihelio, etc. La creación de estas cuatro estructuras es imprescindible puesto que el script que se ejecutará a continuación requerirá de todas ellas además de todos los datos introducidos en las mismas. Un ejemplo de las cuatro estructuras sería el siguiente:

nave =	nombre:	'SpaceCraft'
	tipoEpoca:	0
	tipoSistemaCoordenadas:	0
	epoca:	'2015 Jan 13 11:37:12'
	ci:	[4.7603e+03 877.3023 -4.7553e+03 -3.8451 6.0368 -2.7369]
	CD:	2.2000
	seccionDrag:	15
	seccionSRP:	1
	coeficienteReflexion:	1.8000
	masa:	850
propagador =	tiempo:	3000
	maxStep:	50000
	ordenTolerancia:	-9
	dt:	0.1000
	cuerpoCentral:	'EARTH'
	modeloPotencial:	1
	modeloOtrosPlanetas:	1
	orden:	4
	grado:	4
	modeloDrag:	0
	incluirDrag:	1
	numeroPerturbacionesCuerpos:	4
	idPerturbaciones:	{'SUN' 'MERCURY' 'VENUS' 'MARS'}
	incluirSRP:	1
	stopFunFlag:	1
motores =	fx:	1.5
	fy:	0
	fz:	0
	dir1:	1
	dir2:	0
	dir3:	0
step =	tipoCondicion:	1

Una vez configuradas las cuatro estructuras se ejecutaría el script de Matlab del propagador que explicamos a continuación.

El código consta de unas 1000 líneas y está disponible en los archivos adjuntos del proyecto en la carpeta `/src/simulador/`.

3. Propagador

Función creada en Matlab y que recoge toda la información recibida por la interfaz de Matlab Engine para generar estructuras que se enviarán al integrador. Se trata de una función que tiene como variables de entrada las estructuras previamente definidas `nave`, `propagador`, `motores` y `step` y como variables de salida las variables `estado` y `t`. El script consta de unas 90 líneas de código por lo que lo incluiremos a continuación.

```
function [estado,  t] = propagador2(nave, ...
    propagador , motores, step)


path2mice = '..\MicePackage';

ME = load_spice_kernels ( path2mice );

if (nave.tipoEpoca == 0)
    et0 = cspice_str2et( nave.epoca );
else
    et0 = nave.epoca;
end

format long;

if ( strcmp(propagador.cuerpoCentral,'USER_DEFINED')== 0)
   mu = cspice_bodvrd( propagador.cuerpoCentral , 'GM', 1 );
else
    mu = propagador.muUD;
    for i = 1:6
        Env.cbody_elems{i} = propagador.ciUD(i);
    end
end

if (nave.tipoSistemaCoordenadas==0)
    S0      = zeros(7,1);
    S0(1:6) =  nave.ci;
    S0(7)   = nave.masa;

elseif (nave.tipoSistemaCoordenadas==1)  %rad
    E0 = zeros(8,1);
    E0(1:6)= nave.ci;
    E0(7)= et0;
    E0(8)= mu;
    S0      = zeros(7,1);
    S0(1:6) = cspice_conics(E0, et0);
```

```
    S0(7)   = nave.masa;

elseif (nave.tipoSistemaCoordenadas==2)  %deg
    E0 = zeros(8,1);
    E0(1) = nave.ci(1);
    E0(2) = nave.ci(2);
    E0(3) = nave.ci(3)*pi/180;
    E0(4) = nave.ci(4)*pi/180;
    E0(5) = nave.ci(5)*pi/180;
    E0(6) = nave.ci(6)*pi/180;


    E0(7)= et0;
    E0(8)= mu;
    S0      = zeros(7,1);
    S0(1:6) = cspice_conics(E0, et0);
    S0(7)   = nave.masa;


end

SC_params.cs_drag = nave.seccionDrag;
SC_params.cd      = nave.CD;
SC_params.cs_srp  = nave.seccionSRP;
SC_params.rc_srp  = nave.coeficienteReflexion;

Env.cbody         = propagador.cuerpoCentral;
Env.insphe        = propagador.modeloPotencial;
Env.nsphe_deg     = propagador.grado;
Env.nsphe_ord     = propagador.orden;

Env.cbody_mu = propagador.muUD;


Env.idrag         = propagador.modeloDrag;
Env.isrp          = propagador.incluirSRP;

if propagador.modeloOtrosPlanetas==0
    Env.itdb          = 0;
    Env.num_tdb       = 0;
else
    Env.itdb          = 1;
    Env.num_tdb       = propagador.numeroPerturbacionesCuerpos;
    for i = 1:Env.num_tdb
        Env.tdb_names{i} = propagador.idPerturbaciones(i);
    end
end

deltat = propagador.tiempo;

if step.tipoCondicion == 0
    stop_fun = 'none';
elseif step.tipoCondicion == 1
    stop_fun = @(et, state)pericentre_pass_finder (et, state, mu);
elseif step.tipoCondicion == 2
    stop_fun = @(et, state)apocentre_pass_finder (et, state, mu);
end

tiempos.tolerancia = 10^propagador.ordenTolerancia;
```

```
tiempos.maxStep = propagador.maxStep;
etf = et0 + deltat;
tiempos.vectorTiempo = et0:propagador.dt:etf;
leyesControl = propagador.leyesControl;
ref_frame = nave.sistemaReferencia;

[~, ~, estado, t]  = cowell_propagator (ref_frame, S0, et0, deltat,
...
                                        SC_params, Env, ...
                                        stop_fun, tiempos, leyesControl);
```

Como se puede observar se trata de un proceso secuencial de cálculo de parámetros, transformaciones de coordenadas y creación de estructuras. Todo este código se trata de una versión del propagador de Filippo Cichocki adaptada para nuestro propósito. En varios puntos hacemos uso de funciones `cspice` detalladas con anterioridad para calcular parámetros relacionados con marcos de referencia de elementos orbitales así como datos de efemérides. El script desgrana las estructuras previamente definidas procedentes del simulador para construir las siguientes variables o estructuras:

Variable	Descripción
S0	Vector de 7 coordenadas que incluye las 6 condiciones iniciales del movimiento y la masa de la nave.
et0	Tiempo inicial en formato 'Efemerid time'.
deltat	Tiempo de cálculo.
SC_params	Estructura con parámetros que definen físicamente a la nave.
Env	Estructura con parámetros que definen el entorno.
prnt_out_dt	Step de cálculo para extraer datos a las variables de salida.
stop_fun	Funciones especiales para detener el cálculo.
tiempos	Estructura con tiempos referentes a parámetros de la función ODE45.
motores	Fuerzas extras que se aplicarán al movimiento así como las direcciones en que lo harán.

Tabla VI.4. Variables que se enviarán al propagador

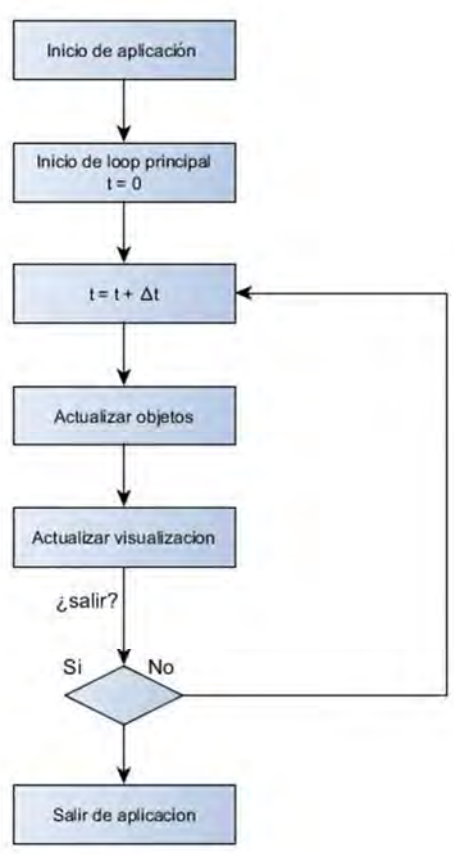
Mención especial merece el parámetro `stop_fun` que como hemos descrito más anteriormente lo conforman funciones especiales para detener el cálculo. En la GUI se habilitará la opción de elegir qué tipo de función se realizará en cada step. Por defecto, la condición que se buscará para terminar un escalón de cálculo será la de cumplir el tiempo preestablecido. No obstante podremos cambiar, mediante el menú habilitado para ello, a otras como por ejemplo alcanzar el periaxis o alcanzar el apoaxis. De cualquier modo el step de cálculo siempre terminará cuando se cumpla la condición escogida, o bien cuando se

rebase el tiempo establecido. Siempre que se configure un step del simulador e independientemente de la función especial de detención que se escoja, se deberá asignar un tiempo de simulación.

4. Simulador 3D

Es el último de los módulos que realizaremos y está escrito en lenguaje Java. Los motivos para utilizar un lenguaje distinto al de los otros módulos han sido ya comentados más arriba y su elección posibilita el uso de las librerías Open Source de desarrollo de entornos en 3D llamadas JMonkey. Se trata de una completa colección de herramientas para diseñar entornos en 3D y realizar movimientos de cuerpos, crear texturas, giros de cámara, etc. Estas librerías fueron diseñadas con el propósito de crear videojuegos a partir de ellas, no obstante pueden ser de gran utilidad para lo que en este proyecto se pretendía en cuanto a la simulación en 3D.

Las opciones que bridan estas librerías son infinitas, y desde luego cualquier desarrollador un poco especializado en entornos 3D habría sido capaz de obtener grandes resultados. Por nuestra parte, hemos utilizado tan solo unas pocas funciones de todas las disponibles, que serán las que describamos aquí.



Para empezar debemos explicar cómo funcionan estas librerías. Lo primero de todo es explicar que el desarrollo del simulador en 3D se ha realizado en la IDE propia de JMonkey que trae todas las librerías necesarias ya incluidas y que facilita mucho el trabajo a la hora de utilizar funciones de esta colección. Una vez inicializado el proyecto tienes pre-cargadas una serie de funciones que se ejecutarán en forma de bucle, como la figura que se muestra a continuación.

De manera muy simplificada, el programa realiza lo que se muestra en el diagrama. Básicamente hasta que el usuario no termina

la ejecución, se va repitiendo el bucle con un Δt de desfase. Este Δt no tiene por qué ser constante durante toda la ejecución sino que tenderá al menor intervalo de tiempo que nuestro ordenador con sus especificaciones técnicas pueda generar para refrescar la pantalla a una tasa de fps aceptable para la correcta visualización. Así, las propias librerías de JMonkey gestionan todo esto incrementando los fps si el escenario no es muy complicado, o reduciéndolos si existen muchos objetos cargados en el entorno.

Para nosotros esto no es un problema y deberemos únicamente tener en cuenta este Δt a la hora de la velocidad de propagación de la órbita.

La estructura del programa se realiza de nuevo por clases. Cada una de las clases contiene funciones propias de cálculo así como todas las variables que definen a ese objeto. Las clases utilizadas son las siguientes:

Nombre	Descripción
Menú	Contiene toda la información acerca del menú mostrado en pantalla de gestión de tiempo y posición.
MenuPantalla	Se trata de una clase necesaria de interfaz entre las clases Menu y la pantalla.
Orbita	Contiene todos los puntos de la órbita además de funciones para gestión de posiciones en el espacio. Más adelante hablaremos con mayor detalle sobre funciones de esta clase.
Planeta	Contiene la información referente al cuerpo central, como el radio. Además gestiona qué cuerpo central se mostrará dependiendo de la línea de comando que llegue de la GUI y que hemos explicado con anterioridad.
Satélite	Contiene la información referente al satélite o cuerpo menor que órbita alrededor del planeta. Se trataría de una clase con funciones parecidas a la de Planeta y que gestiona qué modelo en 3D se carga dependiendo de las preferencias seleccionadas en la GUI.

Tabla VI.5. Clases principales creadas para el simulador 3D

Vamos a mencionar dos de las funciones que se encuentran dentro de las clases de este programa. La primera es la relativa a orbita y está contenida dentro del a clase Orbita. Se trata del método utilizado para gestionar los puntos de la órbita en un tiempo dado.

Primeramente se cargan todos los puntos mediante la lectura de un archivo de texto plano generado en el simulador. A continuación se guardan en ternas

todos esos puntos. En este punto podemos saber ya toda la trayectoria del satélite alrededor del astro, no obstante no sabemos en qué punto está ni a qué tiempos. Para ello se utiliza el parámetro `tiempoSimulacion` enviado desde la GUI y que no tiene por qué coincidir con el tiempo preestablecido en las opciones del propagador –imaginemos el supuesto de que existe una colisión y la simulación termina en un tiempo mucho menor al establecido-. Una vez conocidos estos datos, habilitamos la función `public Vector3f getPosicion(float t)` que para un tiempo dado nos devuelve una terna con los valores XYZ. A esta función se la llamará en cada instante y puesto que contamos con puntos y no una función continua deberemos interpolar para cada valor de t.

La segunda función que queremos comentar es la relativa al movimiento dentro de un entorno en 3 dimensiones. Hemos habilitado mediante la rueda del ratón el poder acercarse y alejarse del centro del cuerpo mediante el aumento o disminución del vector $\vec{q}$ de la figura. La posición de la cámara podrá variar entonces alejándose y acercándose al planeta y la dirección en la que mira, siempre será la del centro de este cuerpo.

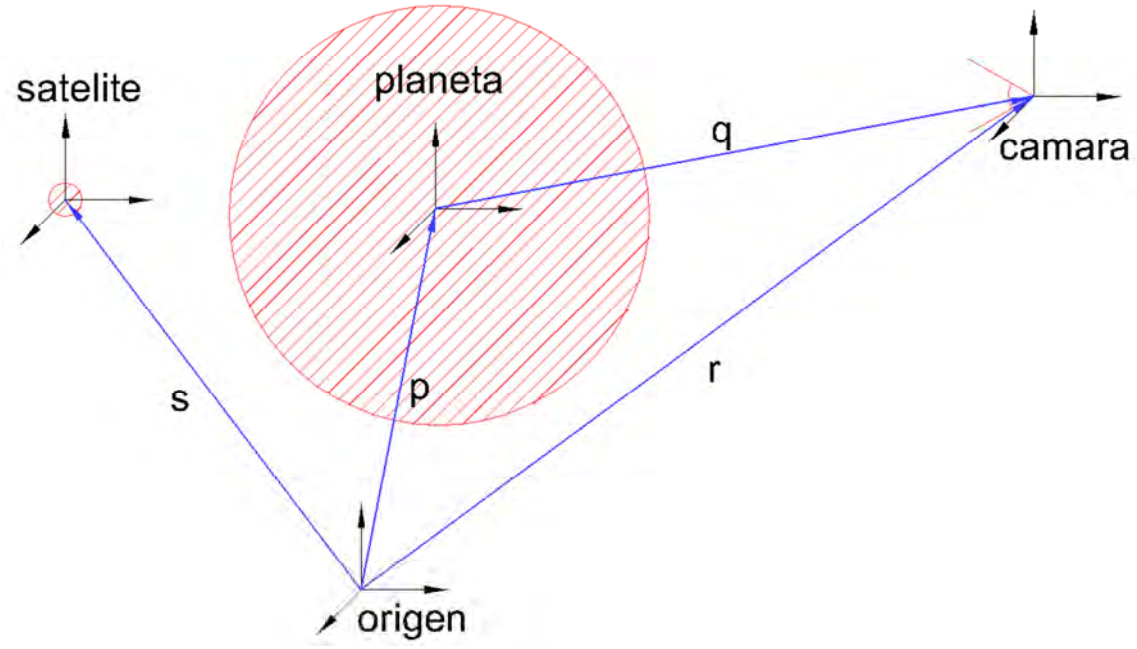


Figura VI.2. Sistema de posicionamiento en módulo de simulación 3D

Por otro lado, para poder observar convenientemente el resultado de la simulación se ha habilitado del mismo modo un método para poder girar la visión mediante la acción pinchar-arrastrar. El método que se ha ideado es el

siguiente. Para cada movimiento o giro de cámara, una vez realizado, se recalcularán los 3 vectores unitarios siguientes:

Nombre	Descripción
Radial	Vector que va desde la cámara hasta el origen del planeta. Siempre podremos calcularlo puesto que siempre conoceremos ambos puntos.
Tangencial	Vector que va desde un punto de la trayectoria hasta el punto siguiente. Una vez terminada la animación, este vector iniciará en el penúltimo punto de la trayectoria y terminará en el último. Es por tanto conocido únicamente hasta que termina la animación y la nave se detiene en el punto final.
Arriba	Vector definido como *arriba* o lo que sería la parte superior de nuestra cámara según queremos ver la secuencia. En el punto final de la trayectoria lo podemos calcular como el producto vectorial entre el *radial* y *tangencial*.

Tabla VI.6. Vectores principales para la orientación en el entorno tridimensional

Disponemos pues inicialmente –cuando la nave se detiene en el último punto de la trayectoria y dejamos al usuario poder modificar la posición de la cámara- de los tres vectores que son perpendiculares entre si y que definen las tres direcciones de interés.

Una vez conocidos esta terna de vectores, mediante la acción de pinchar-arrastrar el usuario podrá girar la cámara como si se encontrase en una esfera de origen el centro del cuerpo central, y como radio el vector $\vec{q}$ dando la impresión de que lo que en realidad gira son los objetos que hay dentro de los límites de la esfera. Conseguimos así el efecto de estar girando el planeta, el satélite y la órbita.

Existen cuatro direcciones para poder girar definidas como *arriba*, *abajo*, *izquierda* y *derecha*. Y se definen así porque no hay que olvidar que se han establecido las tres direcciones principales previamente. De modo que si queremos girar en dirección *derecha*, la cámara se moverá siguiendo el vector *tangente* en dirección positiva. Una vez realizado el giro se deberá volver a recalcular los tres vectores pero en este caso, y como mostramos en la figura, el vector *radial* es siempre conocido, y de los otros dos, el vector *arriba* permanece igual puesto que el giro que hemos realizado ha sido de izquierda a derecha. De modo que el vector *tangencial* podremos recalcularlo como producto vectorial de los otros dos vectores.

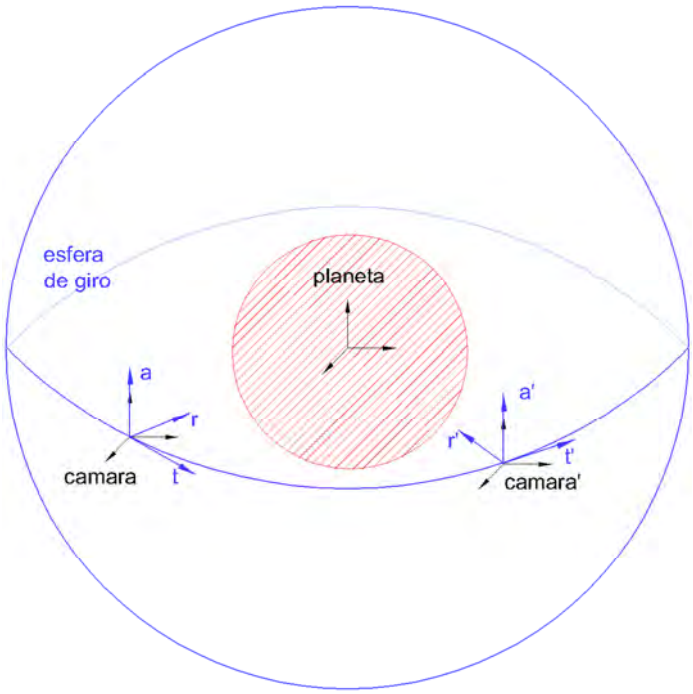


Figura VI.3. Esfera de giro de cámara en módulo 3D.

En la figura observamos como los vectores $\vec{a}$ y $\vec{a}'$ son paralelos por lo que para girar de derecha a izquierda este vector permanece constante y conociendo el vector radial –que también cambia pero que podemos conocer siempre- podemos finalmente calcular el nuevo tangencial $\vec{t}'$.

Es importante resaltar que los términos *derecha* e *izquierda* no hacen referencia a un lugar fijo del espacio sino a un lugar relativo a donde nosotros nos encontramos y que consideramos la derecha o la izquierda.

El giro de arriba-abajo se realiza de igual modo solo que en este caso, el vector que permanece constante seria el vector tangencial y podríamos asegurar que $\vec{t}$ y $\vec{t}'$ son paralelos. A continuación calcularíamos el nuevo vector arriba $\vec{a}'$ mediante el producto vectorial de los otros dos vectores.

Después de cada giro se recalculan las tres direcciones principales quedándose preparadas para el siguiente giro.

Un problema sabido, como se advertirá en la imagen citada previamente, consiste en que si partimos de un punto y nos movemos una unidad en dirección de $\vec{t}$, el punto al que llegaremos no estará contenido dentro de la esfera de giro, y aunque estos tres vectores son unitarios, existe un pequeño error en el posicionamiento. Esta pequeña variación del radio solo se percibe

cuando se realizan giros muy prolongados y continuados aunque es fácilmente corregible mediante la acción de acercar-alejar.

5. Procesos internos

Una vez que están desarrolladas todas las herramientas, en este punto vamos a pasar a describir cómo interaccionan entre ellas, no solo a nivel esquemático –como ya hicimos en el apartado de las interfaces- sino a nivel de estructura de archivos y lectura de los mismos.

Para poder explicar esto primero hablaremos sobre cómo se realiza el cálculo de las variables que posteriormente se transformarán en un archivo de texto para una gráfica o para un grountrack. A continuación detallaremos la distribución de archivos y carpetas que desarrolla nuestro proyecto y finalmente definiremos qué archivos van en qué carpetas y cómo se entiende esto desde la GUI.

Cálculo de variables

Se ha desarrollado un sistema de variables abierto, de modo que cualquier usuario puede añadir o quitar variables en función de sus preferencias o necesidades. Para ello hemos creado un archivo con formato XML que se encuentra en la carpeta de instalación del programa y que incluye todas las variables que ejecutará el simulador después de haber calculado el resultado.

La estructura de este XML es la siguiente:

```
<?xml version="1.0" encoding="utf-8"?>
<variables>
<variable>
  <nombre> ... </nombre>
  <comando> ... </comando>
  <tipo> ... </tipo>
  <uds> ... </uds>
</variable>
 ...
</variables>
```

Para cada etiqueta variable tendremos un nombre, un comando, un tipo y unas unidades.

Como ya se ha dicho, todas estas variables se ejecutarán secuencialmente una vez realizado el cálculo de la posición y velocidad por el simulador. El modo de ejecución es el siguiente. Para cada variable de la lista se ejecutará el comando:

```
>> nombre = comando;
```

Para la ejecución de este comando utilizaremos la interfaz de Matlab Engine. De modo que si el usuario quiere añadir alguna otra variable que desee, el nombre será arbitrario pero el comando deberá ser entendible mediante el motor de Matlab.

En el archivo `variables.xml` creado en la instalación inicial existen variables que en el campo `<comando>` no tienen ningún comando escrito. Esto es debido a que son variables que se ejecutan desde el simulador y nosotros como usuarios podemos disponer de sus resultados para su posterior procesado en otras variables que se precisen. Es importante decir que se recomienda al usuario no borrar ninguna de las variables que vienen en la instalación y añadir todas las que se le antoje. Si borras algunas de las variables que usa intrínsecamente el simulador, se producirán errores y se detendrá la ejecución del programa. Por ejemplo si se borran la latitud o la longitud, cuando el usuario pida un groundtrack el programa lanzará error hasta que no se vuelvan añadir. Por eso se recomienda no eliminar bajo ningún concepto ninguna de las variables que vienen por defecto.

Por otro lado, el campo `<tipo>` hace referencia a la dimensión de la variable. Los tipos que podemos utilizar son los siguientes.

#	Definición
0	Tipo escalar
1	Tipo vectorial
2	Tipo matricial
3	Tipo no definido

Tabla VI.7. Identificadores de dimensión para variables

En el único punto que afecta esta descripción es en la posibilidad de solicitar esa variable para una gráfica. Debe especificarse que es de tipo Vector, para que al pulsar aparezca el menú de selección para la variable X o la Y.

Finalmente la etiqueta `<uds>` no afecta en absoluto a ninguno de los cálculos, pero se utiliza a la hora de mostrar las leyendas en las gráficas que se generan por la GUI. De modo que este campo es totalmente libre y de ninguna manera el programa verificará si estas unidades son o no correctas. En el archivo de variables que viene por defecto se ha utilizado el símbolo asterisco * para

referirnos a matrices que poseen distintos tipos de datos que no hacen referencia a una única unidad de medida –como la matriz estado que posee posiciones [km] y velocidades [km/s]-.

Mediante este sistema de gestión de variables brindamos al usuario final la posibilidad de crear nuevos puntos de vista a los resultados que arroja el simulador.

Distribución de archivos

La organización que se ha elegido para contener todos los bloques del proyecto ha sido la siguiente.

En la carpeta principal, en el nivel superior, se encuentran los tres grandes bloques de la GUI (GUI.exe), el simulador (sim.exe) y el simulador en 3D (sim3d.exe). Además encontramos los dos archivos de librería con extensión .dll que son necesarios para la ejecución del módulo de simulación en 3D. Finalmente tenemos el archivo XML de variables del que hemos hablando con anterioridad.

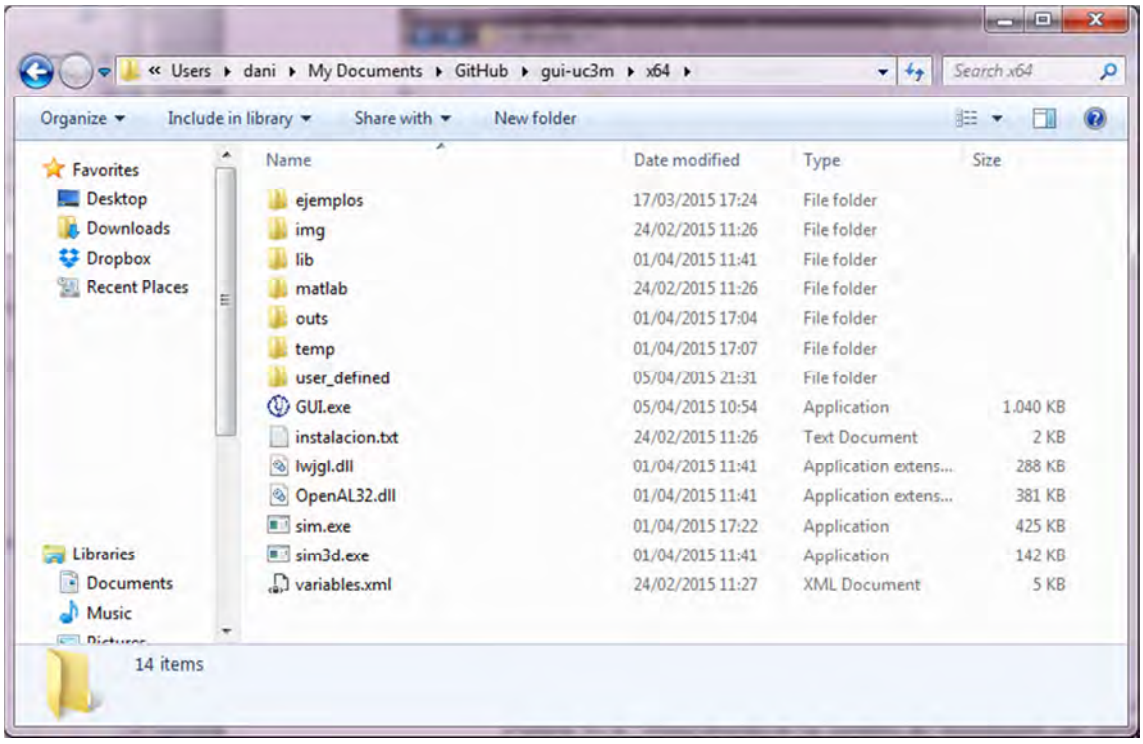


Figura VI.4. Vista previa de la carpeta de instalación del software

En la carpeta `/ejemplos` podemos encontrar distintos archivos XML de misiones que el usuario puede cargar y los cuales están configurados para

realizar una tarea –como por ejemplo una transferencia de Hoffman entre órbitas- que se podrá observar mediante los distintos tipos de salidas.

En la carpeta /`img` se encuentran las imágenes que utiliza la GUI como fondo en los groundtracks de La Luna y de La Tierra y en /`lib` se encuentran todas las librerías necesarias con extensión `.jar` que utiliza el simulador en Java.

En la carpeta /`matlab` se encuentran todos los archivos necesarios para poder realizar la integración de la ecuación diferencial así como distintas herramientas creadas a lo largo del proyecto para hacer más sencillos los procesos. En esta carpeta también podemos encontrar todas las librerías CSPICE de las que ya hemos hablado con anterioridad. Finalmente también se incluyen las herramientas de la Toolbox Google Earth para crear planos tridimensionales con formato KML.

En el directorio /`outs` se irán almacenando todos los archivos de texto con las variables deseadas por el usuario y seleccionadas en la GUI. Los formatos que aparecerán aquí podrán ser de tipo texto plano `.txt` y de tipo Matlab `.mat` si se seleccionó el check correspondiente en la interfaz de configuración.

La carpeta /`temp` alberga todos los ficheros de datos que utilizamos como interfaces así como para guardar variables. Todos ellos son archivos de carácter temporal, es decir, que si se generan dos simulaciones, los archivos de la primera desaparecerán para crear los archivos de la segunda. En esta carpeta es donde se crean la interfaz `data_in.xml` que se utiliza entre la GUI y el simulador, todos los archivos de texto que contienen variables seleccionadas en outputs de tipo groundtrack, gráficas y simulación 3D, además del archivo KML generado para la visualización sobre Google Earth entre otros. Además es en esta carpeta donde podemos consultar todas las variables introducidas en el script mediante el Matlab Engine desde el simulador en el archivo `workspace.mat`.

Finalmente en la carpeta /`user_defined` debe estar el modelo del asteroide que se cargará en el módulo 3D si se selecciona esta opción de simulación.

Esta configuración de archivos y carpetas no se puede modificar puesto que en muchos puntos de cada uno de los bloques del proyecto se ha establecido estos valores como constantes a la hora de cargar o guardar archivos.

Petición de datos

Una vez que se llama al simulador desde la interfaz de usuario, éste realiza el cálculo y una vez terminado procesa todas las peticiones de tipo Output que se han preseleccionado en la GUI. Para cada petición, el simulador sabe qué tipo de archivo debe de crear y en qué carpeta lo debe de guardar.

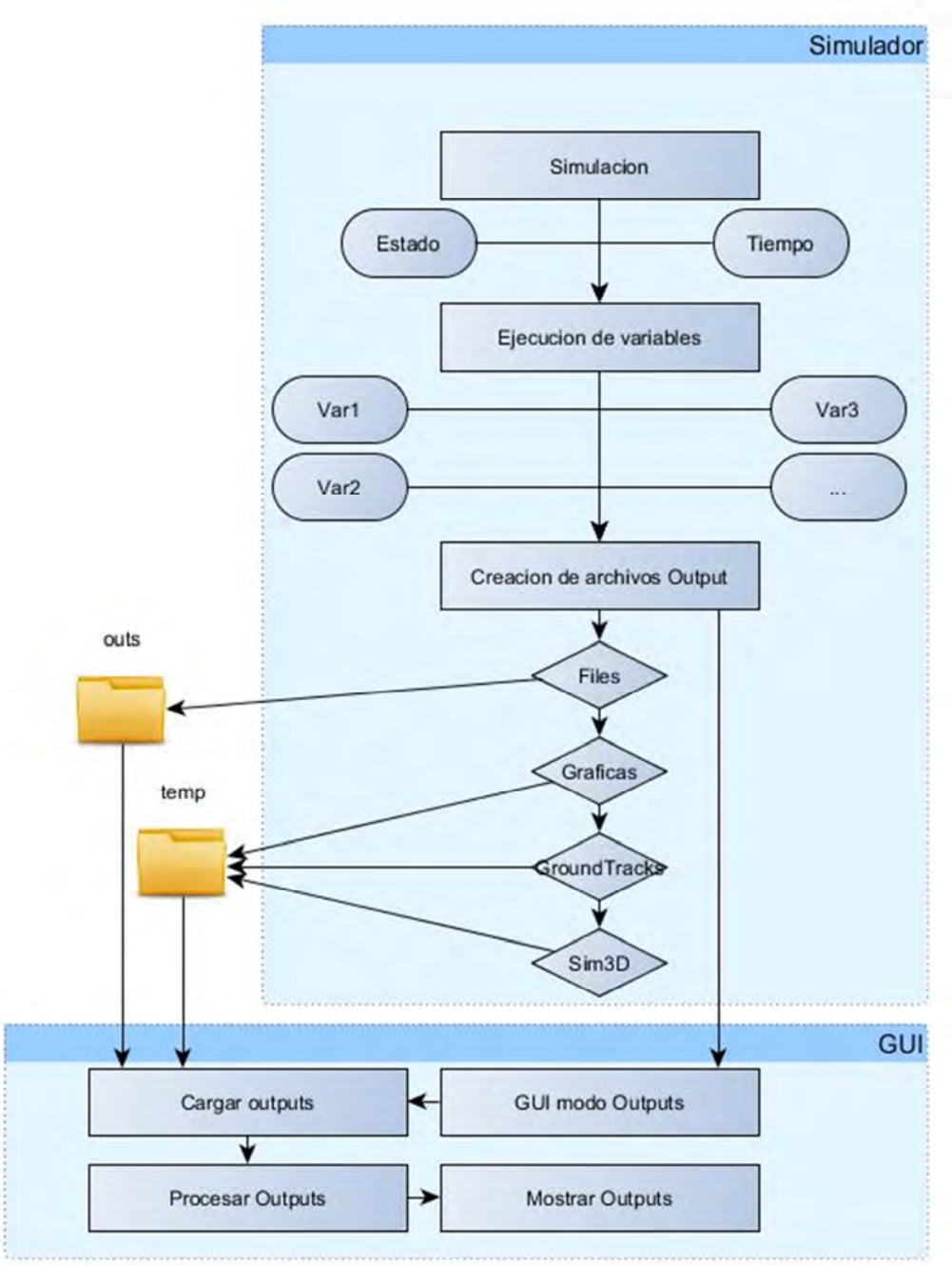


Figura VI.5 Diagrama de secuencia entre el simulador y la GUI

Los archivos creados por el simulador para files, gráficas, groundtracks y simulaciones 3D son siempre archivos de texto plano con una secuencia de

puntos que dependerán de la naturaleza del output. Si en la gráfica se ha solicitado la curva de la componente X de la velocidad frente al tiempo, entonces se creará un archivo de texto con dos líneas de puntos, la primera el tiempo y la segunda la velocidad. Todo esto se gestiona desde el simulador y por supuesto que las variables solicitadas para mostrar deben haber sido introducidas en el esquema XML de variables que hemos explicado con anterioridad.

Una vez se han creado todos los archivos de texto plano, se sale del simulador y se ejecuta la GUI en modo Outputs. El proceso que se lleva a cabo en este modo es el siguiente. El primer paso es cargar el archivo XML de interface que se ha guardado previamente en la carpeta `/temp`. Con esto ya tendremos todas las variables configuradas como cuando lanzamos la simulación. En este punto conocemos ya qué outputs ha solicitado el usuario y para cada uno de ellos podremos buscar en la carpeta correspondiente. El archivo que se buscará en la carpeta será el que se llame como el nombre del output que por consiguiente, deberá de ser único.

Para cada tipo de output se procesa la información de modo distinto. Para los outputs de tipo file, se lee el archivo de texto y se guardan todos los puntos en matrices que se redimensionarán en tiempo de ejecución adaptándose a cada uno de los archivos pedidos –puesto que tanto el número de variables como el número de puntos podrán variar de uno a otro- y cuando el usuario solicite ver estos archivos se cargarán en un campo de texto multilínea. Este es el más simple de los procesados que vamos a encontrar.

Para el caso de las gráficas, así como para los groundtracks –que no son otra cosa que gráficas de latitud/longitud- el procesado será distinto. En este caso es necesario además de cargar la información en variables, hacer una conversión entre valores de tipo `string` y valores de tipo `double` –números decimales- para que posteriormente el objeto 'Chart' pueda entenderlo y pintarlos. Del mismo modo que en los outputs de tipo file, en las gráficas se redimensiona en tiempo de ejecución la matriz que contendrá todos los valores `double` para cada punto y para cada gráfica seleccionada. Finalmente cuando el usuario solicita visionar una gráfica se cargan todos los datos referentes a esa solicitud en el objeto Chart mediante puntos que posteriormente se unirán con líneas y se añadirán los títulos de los ejes para el nombre de la variable y las unidades –datos que se sacarán del archivo de variables.xml y que ya se habló en sun

momento-. El programa da la opción de poder guardar las gráficas en formato JPG mediante un botón que se sitúa sobre ellas.

Finalmente, para los outputs de tipo simulación 3D, el procesado se lleva a cabo en el bloque del simulador tridimensional. En este bloque se realiza un procedimiento muy parecido al realizado en la GUI. Se cargan todos los datos de tipo `string` del archivo `.txt` creado en la carpeta `/temp` y se guardan con formato `double`. Una vez estructurada la matriz de puntos XYZ tendremos una trayectoria espacial que podremos dibujar uniendo cada uno de los puntos. Para el movimiento del satélite alrededor del planeta tomaremos la posición para cada instante como la interpolación entre los dos puntos conocidos a dos tiempos conocidos consiguiendo el efecto de movimiento suave y continuado siguiendo la trayectoria.

VII. Validación de resultados

En este apartado se va a tratar de validar los resultados que arroja el cálculo realizado por el simulador.

Antes de comenzar necesitamos definir la herramienta que vamos a usar para comparar nuestros resultados para determinar el grado de exactitud y para ello se va a utilizar el software creado por la NASA GMAT (General Mission Analysis Tool) del que ya se han comentado sus características y su propósito. Se trata de la herramienta en la que más se puede confiar debido a la solvencia y reputación de la institución que la realizó. Por tanto basaremos nuestro análisis en la premisa de que los datos arrojados por este software son reales y su grado de exactitud es muy elevado. De modo que nos servirán de guía para poder analizar los resultados calculados por nuestro simulador.

El proceso de validación de resultados consistirá en la realización de diversas pruebas en ambos simuladores con las mismas condiciones iniciales además de las fuerzas de perturbación y la configuración del integrador. Las variables que analizaremos para medir el error serán las posiciones y velocidades en cada uno de los puntos puesto que son las que definen todo el movimiento. Un bajo error en posición y velocidad garantiza que todos los datos calculados a partir de ellos serán también precisos.

Antes de comenzar con las pruebas se va a resumir rápidamente el método de medición de errores. Por un lado debemos de considerar que por lo general el número de puntos que arrojen los resultados del GMAT no tiene por qué coincidir con el número de puntos que devuelva nuestro programa. Esto es un problema puesto que la comprobación del error se realizará punto a punto. Además está el inconveniente añadido de que incluso siendo el número de puntos iguales en ambas muestras, el vector de tiempos en ambos casos no tiene por qué coincidir tampoco.

Para solucionar esto se ha realizado el siguiente procedimiento. Una vez decididas las características de cada una de las pruebas se configura el GMAT y se realiza la simulación. De los gráficos obtenidos sacamos 6 archivos de texto para cada una de las variables –posiciones y velocidades- y para cada uno de estos archivos obtenemos dos columnas, la base de tiempos que ha utilizado GMAT y el valor de la variable para ese tiempo.

Una vez tenemos estos archivos, se ha creado un pequeño script en Matlab que realiza los siguientes pasos:

Primero carga uno de los archivos y extrae únicamente el vector de tiempos que ha utilizado el GMAT –y que coincide en todos los casos para las 6 variables- y con ese vector, forzamos a nuestro programa a que nos devuelva las variables de estado para esos mismos tiempos. Esto es sencillo de realizar puesto que la función ODE45 permite establecer en qué valores de tiempo quieres que devuelva el estado. Ejecutamos nuestro programa y comparamos los resultados en ambos casos.

Realizando este procedimiento conseguimos solventar los dos problemas mencionados puesto que el número de puntos siempre coincidirá y también la base de tiempos.

Haciendo esto conseguimos dos arrays de valores para cada variable, evaluados justo en el mismo instante y nos ahorramos tener que interpolar con el consiguiente aumento de error.

Una vez tenemos estos dos arrays para cada una de las variables comparamos los puntos uno a uno y mostramos la evolución de los errores relativos para la posición y velocidad mediante las siguientes expresiones adimensionales:

$$\frac{\|\vec{r}_{GMAT} - \vec{r}_{GUI}\|}{\|\vec{r}_{GMAT}\|} \qquad \frac{\|\vec{v}_{GMAT} - \vec{v}_{GUI}\|}{\|\vec{v}_{GMAT}\|}$$

Para las pruebas se han escogido muestras que toquen todas las posibles situaciones en las que un satélite se puede encontrar.

Para las cuatro misiones que vamos a estudiar hemos utilizado la misma época $ep = 2000\,Jan\,01\;11{:}59{:}28$, la misma masa del vehículo $m = 850kg$ y el sistema de referencia $J2000$ que equivale al sistema inercial ecuatorial en la época $J2000$. Para el propagador se ha tomado orden de precisión $ord = -9$, paso mínimo $\Delta t_{min} = 0.1$ y paso máximo $\Delta t_{max} = 50000$.

Lo que diferencia cada una de las pruebas serán las condiciones iniciales, el modelo de fuerzas y el tiempo de integración.

1. Orbita LEO sin excentricidad y sin perturbaciones

En la primera prueba vamos a realizar una misión simple sin muchas variables. Para empezar se trata de una órbita de baja altura –LEO por sus siglas en inglés- con baja excentricidad para que sea lo más circular posible. Las condiciones iniciales han sido:

$$CI_{cartesianas} = \begin{pmatrix} X = & 7100 & km \\ Y = & 0 & km \\ Z = & 1300 & km \\ VX = & 0 & km/s \\ VY = & 7.35 & km/s \\ VZ = & 1 & km/s \end{pmatrix} \quad CI_{EO} = \begin{pmatrix} SMA = & 7191.9 & km \\ ECC = & 0.0245 & \\ INN = & 12.85 & deg \\ RAAN = & 306.6148 & deg \\ AOP = & 314.1905 & deg \\ TA = & 99.8877 & deg \end{pmatrix}$$

Por otro lado, el modelo de fuerzas no incluye ningún tipo de perturbación. Únicamente actúa la fuerza de atracción de La Tierra. El tiempo de integración ha sido $t = 1\ semana$.

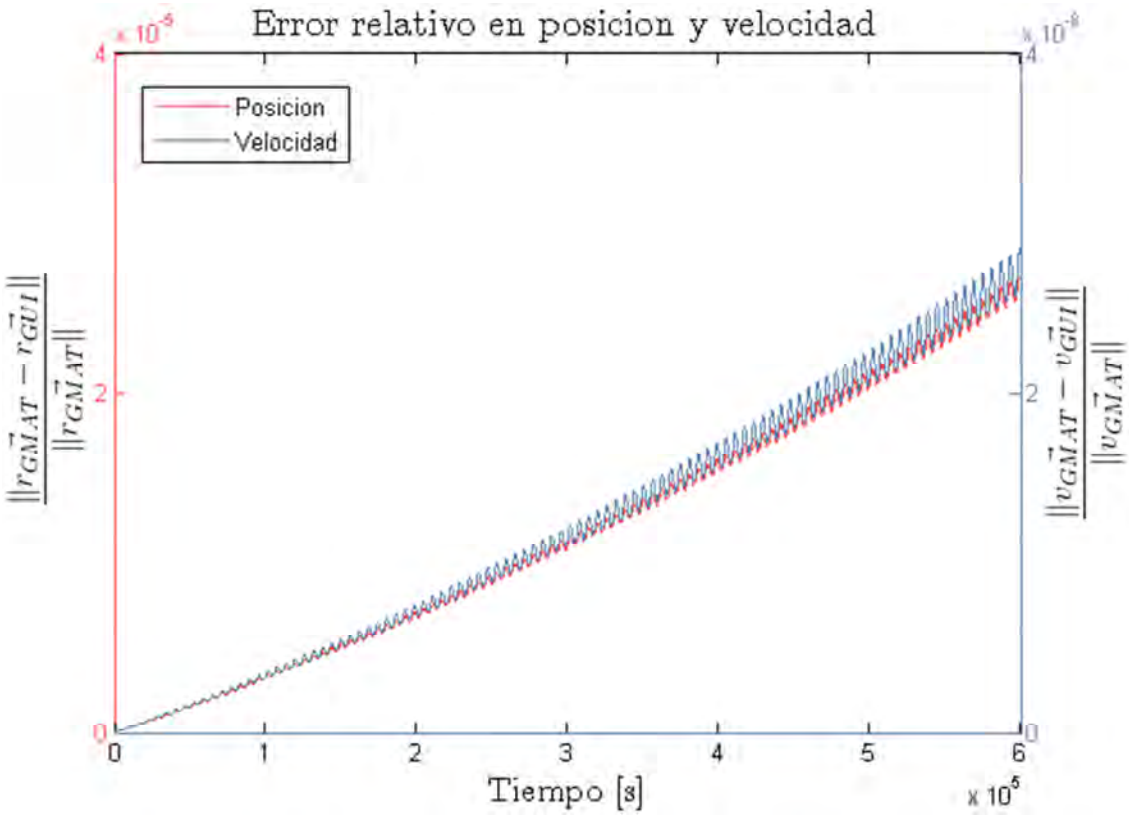


Figura VII.1. Errores de posición y velocidad de la prueba 1

2. Orbita LEO con excentricidad alta y sin perturbaciones

La segunda prueba es prácticamente igual a la anterior con la salvedad de que hemos aumentado la velocidad VY con el objetivo de incrementar la excentricidad de la órbita. Las condiciones iniciales han sido:

$$CI_{cartesianas} = \begin{pmatrix} X = & 7100 & km \\ Y = & 0 & km \\ Z = & 1300 & km \\ VX = & 0 & km/s \\ VY = & 9.35 & km/s \\ VZ = & 1 & km/s \end{pmatrix} \qquad CI_{EO} = \begin{pmatrix} SMA = & 18099.09 & km \\ ECC = & 0.6013 & \\ INN = & 11.9720 & deg \\ RAAN = & 300.2901 & deg \\ AOP = & 57.3330 & deg \\ TA = & 2.9225 & deg \end{pmatrix}$$

Del mismo modo el modelo de fuerzas no incluye ningún tipo de perturbación. Únicamente actúa la fuerza de atracción de La Tierra. El tiempo de integración ha sido $t = 1\ semana$.

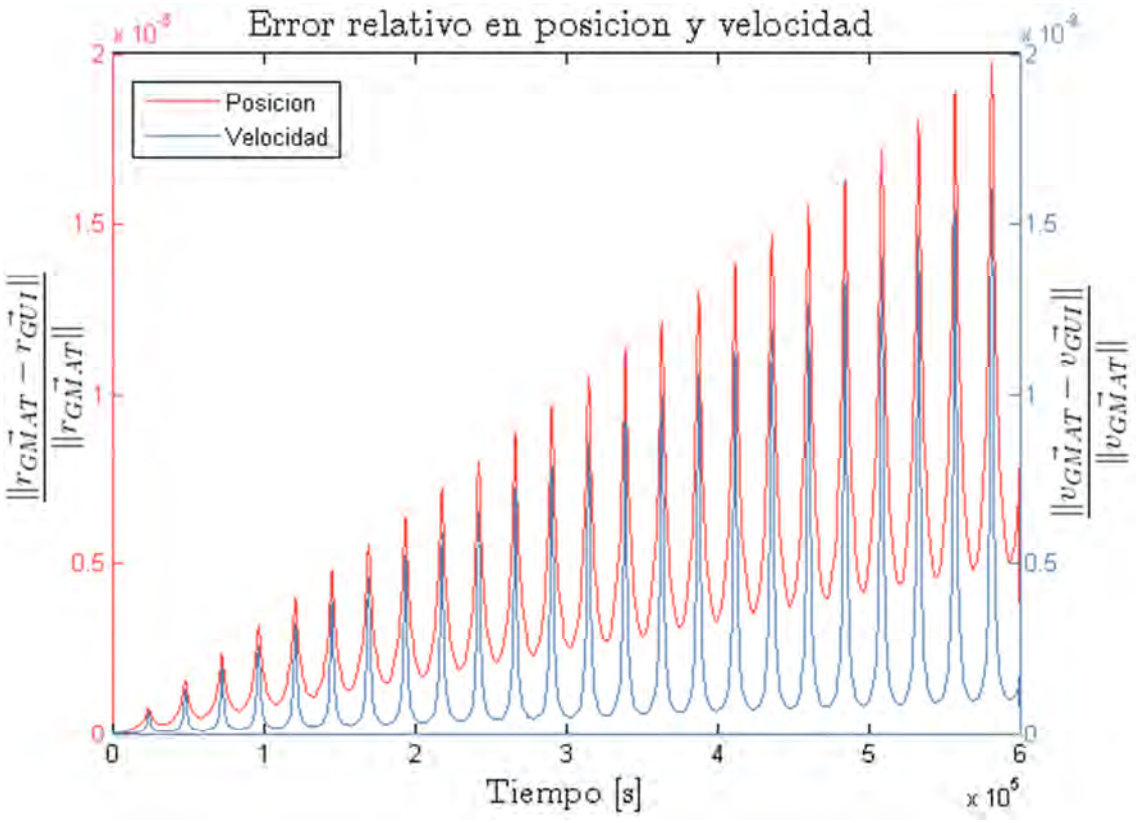


Figura VII.2. Errores de posición y velocidad de la prueba 2

3. Orbita LEO circular con perturbaciones J2, Sol y Luna

En esta tercera prueba entramos en las perturbaciones. La posición inicial del vehículo ha sido la misma que en la primera prueba. Las condiciones iniciales han sido:

$$CI_{cartesianas} = \begin{pmatrix} X = & 7100 & km \\ Y = & 0 & km \\ Z = & 1300 & km \\ VX = & 0 & km/s \\ VY = & 7.35 & km/s \\ VZ = & 1 & km/s \end{pmatrix} \qquad CI_{EO} = \begin{pmatrix} SMA = & 7191.9 & km \\ ECC = & 0.0245 & \\ INN = & 12.85 & deg \\ RAAN = & 306.6148 & deg \\ AOP = & 314.1905 & deg \\ TA = & 99.8877 & deg \end{pmatrix}$$

En el modelo de fuerzas hemos introducido como perturbaciones el Sol y la Luna y además se ha introducido perturbación por el efecto J2 que equivale a $grado = 2, orden = 0$. El tiempo de integración ha sido $t = 1\ semana$.

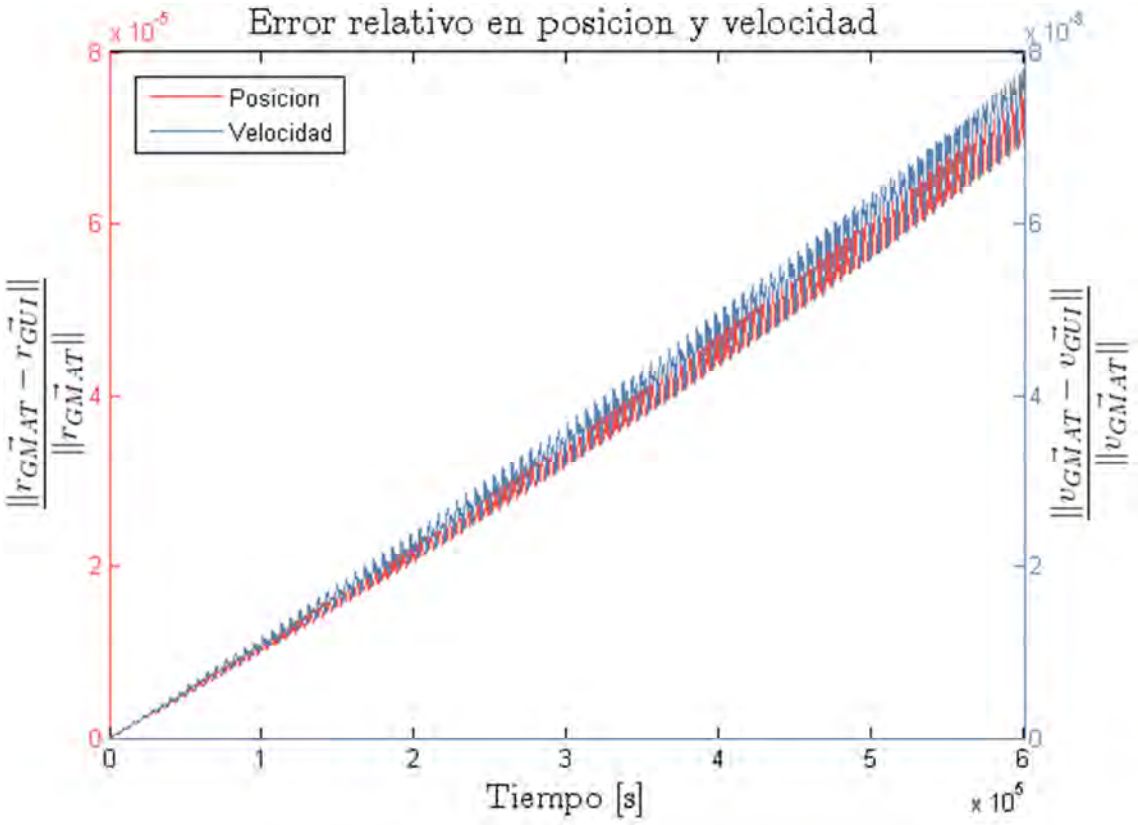


Figura VII.3. Errores de posición y velocidad de la prueba 3

4. Orbita GEO circular con perturbaciones C22, S22, Sol y Luna

Para el último caso se ha tomado una órbita GEO –también conocida como órbita geoestacionaria- . Las condiciones iniciales han sido:

$$CI_{cartesianas} = \begin{pmatrix} X = & 42164 & km \\ Y = & 0 & km \\ Z = & 0 & km \\ VX = & 0 & km/s \\ VY = & 3.0746 & km/s \\ VZ = & 0 & km/s \end{pmatrix} \qquad CI_{EO} = \begin{pmatrix} SMA = & 42164 & km \\ ECC = & 0 & \\ INN = & 0 & deg \\ RAAN = & 0 & deg \\ AOP = & 0 & deg \\ TA = & 0 & deg \end{pmatrix}$$

En el modelo de fuerzas hemos considerado como perturbaciones el Sol y la Luna y además se ha introducido perturbación por el efecto C22 y S22 que equivale a $grado = 2, orden = 2$. El tiempo de integración ha sido $t = 4\ semanas$.

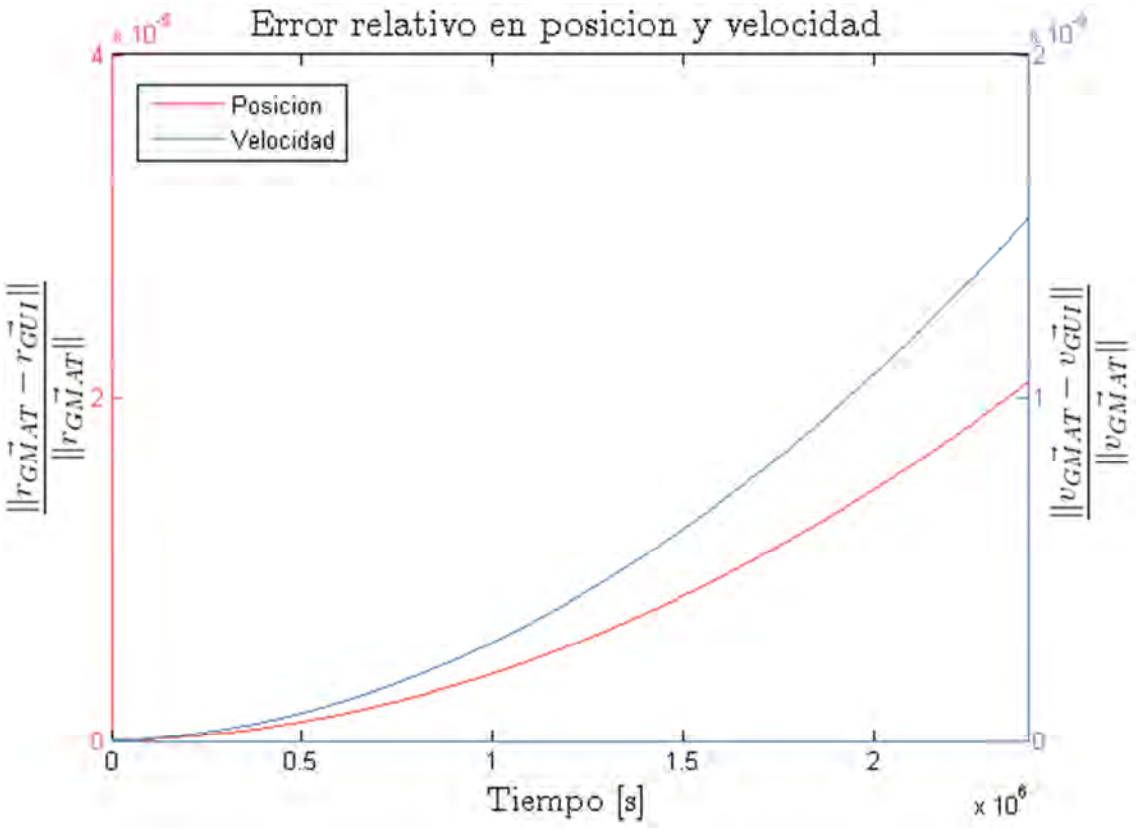


Figura VII.4. Errores de posición y velocidad de la prueba 4

5. Análisis de resultados

A continuación se realizará un análisis de los resultados obtenidos para cada una de las pruebas y se intentará argumentar por qué existe error y qué podemos esperar de futuras simulaciones.

En general los resultados obtenidos en las cuatro muestras han sido muy satisfactorios. Como era de esperar, para los cuatro casos el error ha ido en aumento según transcurría el tiempo. También era de esperar que los errores en la posición y en la velocidad fuesen a la par –aunque en distintos órdenes de magnitud- debido a que ambas variables están interrelacionadas. La tendencia de las curvas sería exponencial para todos los casos aunque en los tres primeros no se observa del todo hay que tener en cuenta que el tiempo de simulación en estos ha sido únicamente una semana mientras que el cuarto caso han sido cuatro semanas.

En el primer caso obtenemos que transcurrida una semana los errores relativos en la posición son del orden de $e = 2.5 \times 10^{-5}$. Dado que en este primer caso trabajamos con una órbita LEO esto implica errores absolutos del orden de $\|\vec{r}_{GMAT} - \vec{r}_{GUI}\| \sim 7000 \times 2.5 \times 10^{-5} = 0.175km$ o lo que es lo mismo, tras una semana de integración nuestro vehículo estará unos 175 metros más alejado de donde debería. En la velocidad el error relativo ha sido del orden de $e = 3 \times 10^{-8}$. Considerando el módulo de la velocidad del orden de $\|\vec{v}_{GMAT}\| \sim 8km/s$ obtenemos un error absoluto del orden de $\|\vec{v}_{GMAT} - \vec{v}_{GUI}\| \sim 2.4 \times 10^{-7} km/s$.

Para el segundo caso los errores poseen variaciones mayores dentro de los cada uno de los periodos alrededor de La Tierra. En este ejemplo es sencillo contar el número de vueltas que ha dado a la tierra en una semana que equivaldría al número de picos que hay en las gráficas. En este caso se realizarían 24 vueltas completas alrededor de La Tierra. Y en este tiempo el error que se ha producido ha sido del orden de $e = 2 \times 10^{-5}$ en el perigeo y del orden de $e = 0.5 \times 10^{-5}$ en el apogeo. Tomamos el mayor que se produce precisamente cuando el vehículo alcanza el perigeo y su velocidad es máxima –recordemos que en este caso la excentricidad ha sido muy elevada-. El error absoluto será del orden de $\|\vec{r}_{GMAT} - \vec{r}_{GUI}\| \sim 18000 \times 2 \times 10^{-5} = 0.36km$. Para las velocidades el razonamiento sería el mismo siendo el error relativo máximo $e = 1.5 \times 10^{-8}$. Otro punto importante es que al haber aumentado la excentricidad de la órbita, las variaciones en velocidad y posición son mayores permitiéndonos observar un hecho interesante que es que ambas magnitudes están en fase.

En el tercer caso hemos tenido en cuenta las perturbaciones debidas tanto a los cuerpos del Sol y de la Luna como los efectos producidos por el J2 debido a la no homogeneidad de la densidad de La Tierra.

Los errores relativos en la posición para esta prueba han sido del orden den $e = 8 \times 10^{-5}$. Recordemos que todas las condiciones iniciales de la órbita eran las mismas que para el primer caso, no obstante considerando las perturbaciones el error se ha multiplicado por 3. Los errores absolutos serían del orden de $\|\vec{r}_{GMAT} - \vec{r}_{GUI}\| \sim 7000 \times 8 \times 10^{-5} = 0.56km$. Es decir, al considerar otras perturbaciones nuestro cálculo obtiene más error en comparación al del GMAT. Más adelante se intentará argumentar este hecho.

En las velocidades para el tercer caso obtenemos un error relativo del orden de $e = 8 \times 10^{-8}$ que también es casi tres veces mayor que en el primer caso.

Finalmente para el último caso, el único en el que hemos considerado una órbita geoestacionaria y con las mismas perturbaciones que en el anterior, los errores relativos en la posición transcurridas 4 semanas han sido de $e = 2 \times 10^{-5}$. Esto implica unos errores absolutos del orden de $\|\vec{r}_{GMAT} - \vec{r}_{GUI}\| \sim 42000 \times 2 \times 10^{-5} = 0.84km$. Es interesante hacer notar que para este caso, habiendo añadido perturbaciones y aumentado en 4 el tiempo de integración, el error en la posición no se ha visto incrementado en la misma medida. Para la velocidad el error es incluso un orden menor que en el resto de los casos siendo éste $e = 1.5 \times 10^{-9}$.

A continuación mostramos en una tabla el resumen de los errores para cada una de las pruebas.

	Posición		**Velocidad**	
	e_{rel}	$e_{abs}[km]$	e_{rel}	$e_{abs}[km/s]$
Prueba 1	2.5×10^{-5}	0.175	3×10^{-8}	2.4×10^{-7}
Prueba 2	2×10^{-5}	0.36	1.5×10^{-8}	1.4×10^{-7}
Prueba 3	8×10^{-5}	0.56	8×10^{-8}	5.8×10^{-7}
Prueba 4	2×10^{-5}	0.84	1.5×10^{-9}	4.5×10^{-9}

Tabla VII.1. Resumen de errores en posición y velocidad para los cuatro casos estudiados

Las conclusiones que sacamos de los resultados son las siguientes. Para empezar podemos afirmar que los resultados arrojados por nuestro software son muy buenos. En ninguno de los cuatro casos el error ha sido mayor a 600

metros para una semana de integración y en el cuarto caso el error ha sido menor a 1km para cuatro semanas. También podemos afirmar que para las orbitas con mayores radios –como son el segundo y el cuarto caso- los errores se reducen. Esto es debido a que al cubrir una distancia mayor pasan menos veces por el perigeo no existiendo grandes variaciones en su velocidad provocando menos error.

Otra afirmación que podemos sacar a la vista de los resultados es que cuantas más perturbaciones incluyamos, mayor será el error. Esto podría parecer incongruente a primera vista puesto que si incluimos perturbaciones es para hacer nuestro cálculo más preciso. Pero tiene una explicación muy sencilla. Nuestros datos los comparamos con los de otro software, no con efemérides de elementos reales que se mueven por el espacio. La ecuación que resolvemos en cada caso es la siguiente:

$$\frac{d^2\vec{r}}{dt^2} + \frac{\mu_p}{|r|^2} \cdot \frac{\vec{r}}{|r|} = \vec{f}_{Pert}$$

El término de perturbaciones para los dos primeros casos es cero. De modo que comparando los datos de nuestro programa con los del GMAT existirán menos posibles términos donde cometer error. Esto significa que cuantas más fuerzas de perturbación incluyamos, mas términos existirán donde cometer error lo que implicará un error mayor en los resultados.

Este hecho puede ser debido a distintos factores. Cada vez que realizamos un cálculo en nuestro simulador tenemos una precisión de variable prefijada. En Matlab, las variables `doubles` contienen 16 dígitos por defecto de modo que esta sería nuestra precisión máxima. Es cierto nuestros errores son del orden de -5 no obstante hay que tener en cuenta que el error de -16 que cometemos se repite a cada ciclo pudiendo aumentar al final de la simulación el error relativo en gran medida.

Otro punto a tener en cuenta son las constantes que usamos en nuestro programa y las que usa GMAT. Es posible que algunas de ellas no sean exactamente iguales. Por poner un ejemplo, las constantes referidas a la no homogeneidad de la densidad de La Tierra para nuestro simulador tienen una precisión de 11. Un ejemplo sería el J2

```
C(2,0) = -1.08262668355*1e-3;
```

Si GMAT para este coeficiente utiliza otro valor con más o menos decimales existirá un error en la constante que provocará errores en los resultados.

Finalmente mencionar también el hecho de que para nuestro simulador, la tarea de integrar la ecuación diferencial se externaliza mediante la función ODE45 de Matlab. Si exigimos a esta función que nos devuelva valores a tiempos muy concretos –como se ha hecho para realizar la medición de los errores- es posible que el mecanismo que utiliza sea el de interpolar y aunque lo haga de modo que garantice la precisión preestablecida, una interpolación siempre producirá error en los resultados.

VIII. Aplicaciones

Una vez desarrollada la GUI y validados los resultados vamos a pasar a explicar qué aplicaciones prácticas se le puede dar al conjunto de bloques que conforman el programa.

1. Propagación de órbita

El cálculo de la posición y velocidad de una órbita a lo largo de su trayectoria representa el fin mismo de la creación de este programa. Todas las operaciones que realiza se fundamentan en lo mismo, la integración de la ecuación diferencial del movimiento de la nave alrededor de un cuerpo a la que es posible añadir más términos de perturbación con el propósito de aumentar el grado de exactitud.

Por medio de la GUI se puede resolver el problema de calcular la posición y velocidad de una órbita a lo largo del tiempo. Una vez seleccionadas las condiciones iniciales –del tipo cartesiano o kepleriano- además de la época y los parámetros de la nave podemos fácilmente obtener los resultados deseados que podrán ser de tipo File, Gráfica, Groundtrack o simulación 3D.

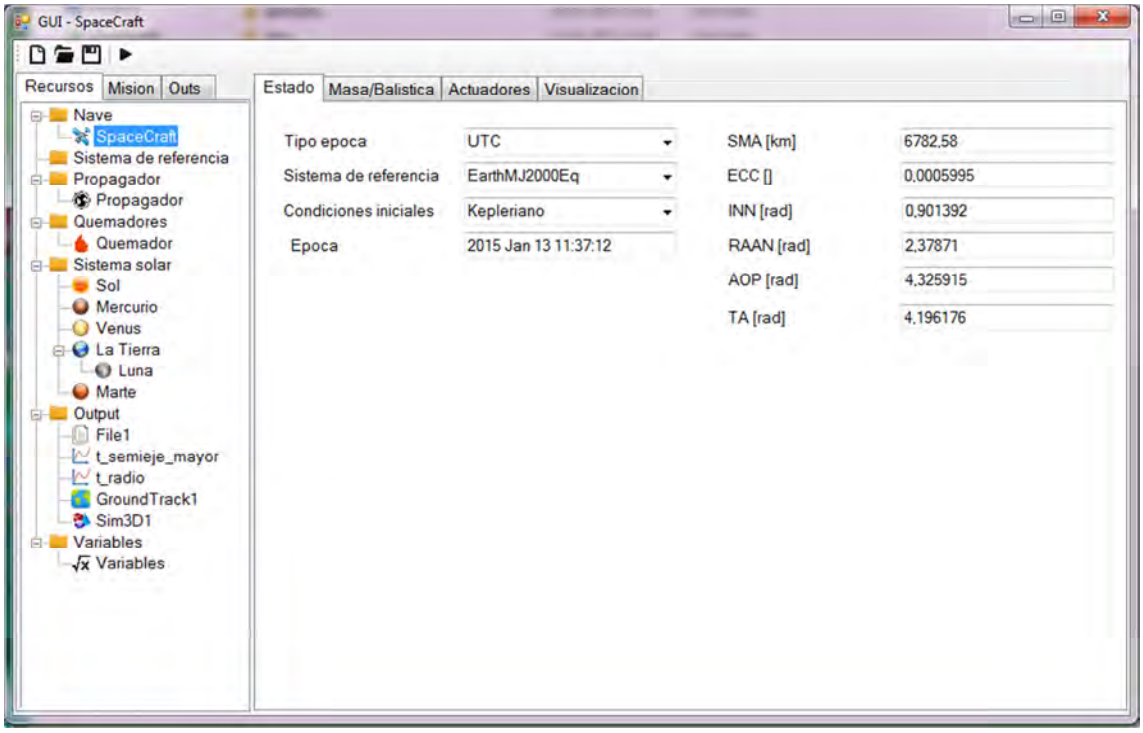


Figura VIII.1. Vista previa de configuración inicial de SpaceCraft en la interfaz de usuario

Mediante la GUI configuramos las condiciones iniciales así como la época y ajustamos el tiempo de simulación a 8000 segundos. A continuación ejecutamos el simulador.

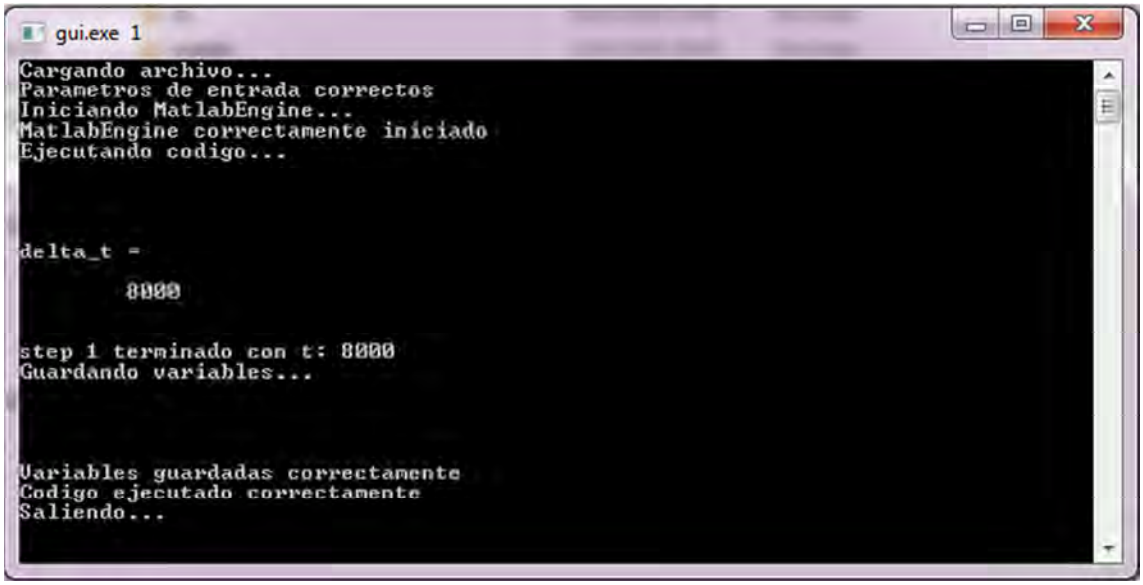


Figura VIII.2. Vista previa de secuencia de ejecución del simulador

El simulador comienza a ejecutar la secuencia y cuando termina nos devuelve a la GUI en modo Outputs. Dentro del os Outputs podemos observar las salidas de tipo File como columnas de datos. Habrá tantas columnas como variables se hayan seleccionado en la GUI previamente.

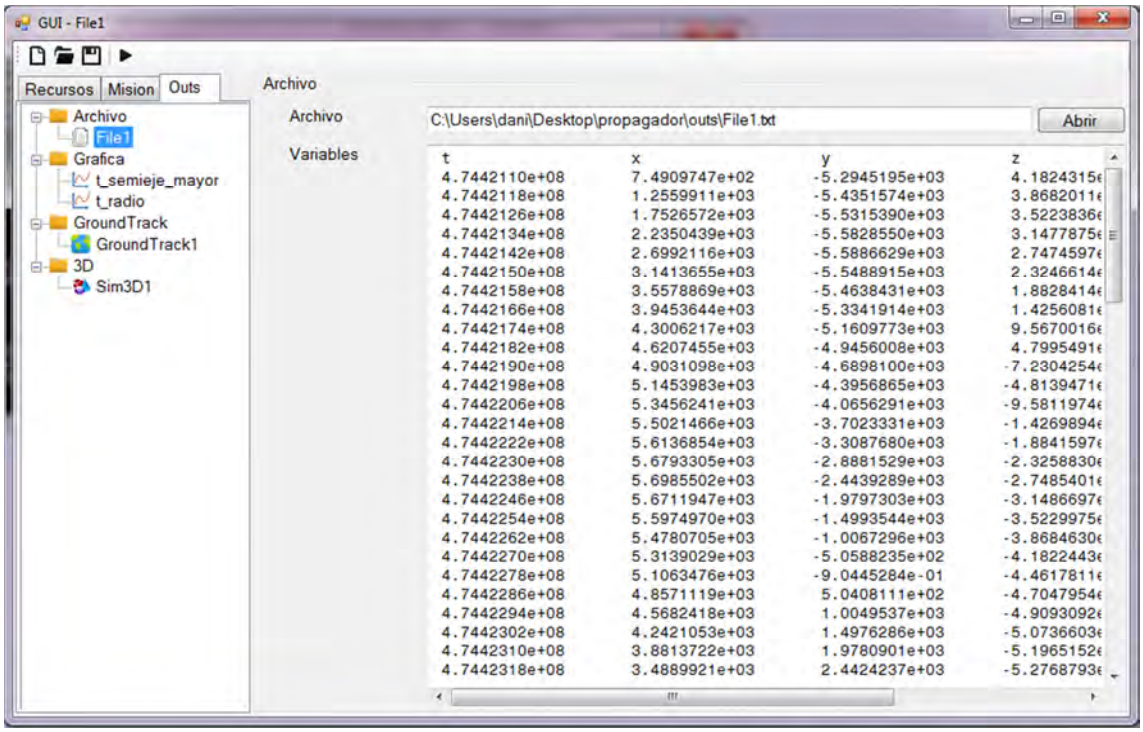


Figura VIII.3. Vista previa de resultados de tipo File en interfaz de usuario

Así mismo se podrá observar las gráficas de las variables XY que se hayan configurado previamente en la pestaña de Recursos.

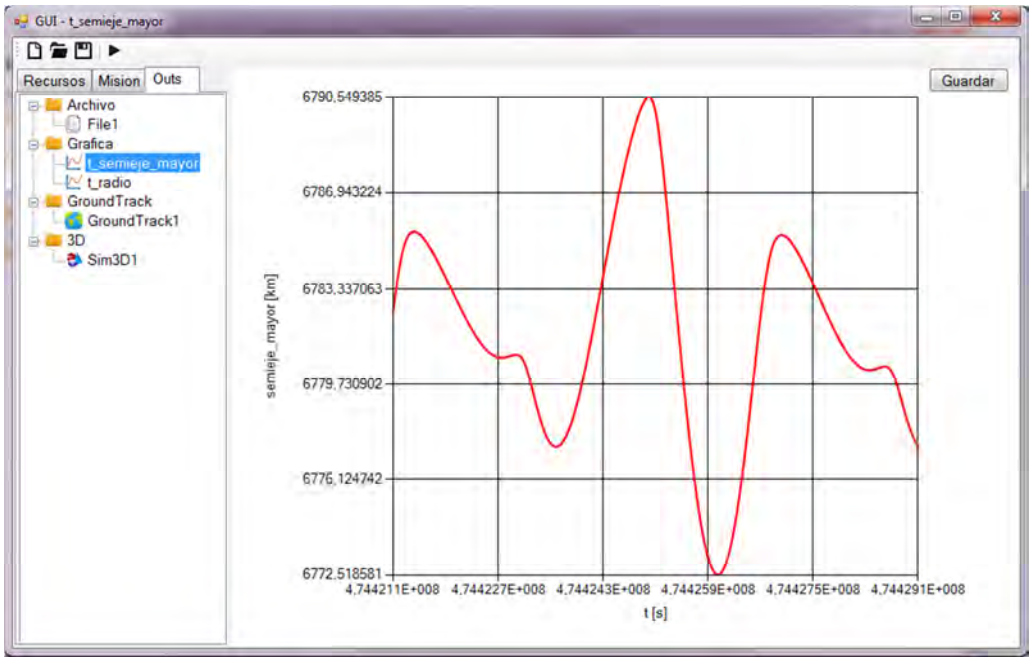


Figura VIII.4. Vista previa de resultados de tipo Grafica en interfaz de usuario

Podremos poner tantas gráficas como deseemos.

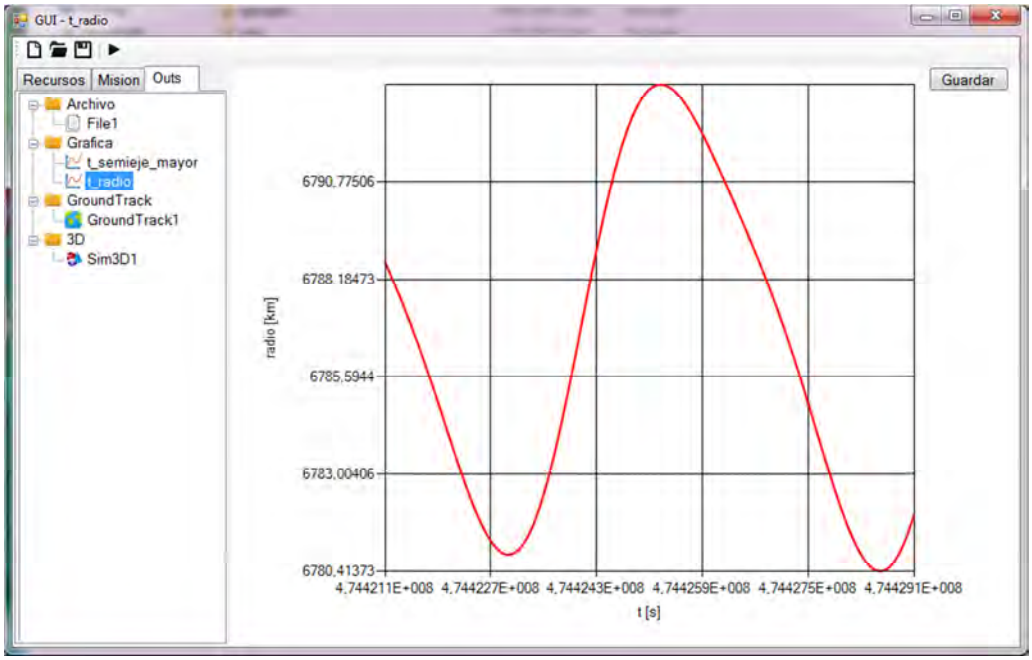


Figura VIII.5. Vista previa de resultados de tipo Grafica en interfaz de usuario

Además se dispone de la opción de visualizar la salida como un gráfico de latitud longitud en marco de referencia ECEF y mediante la superposición sobre

un mapa terrestre –o lunar- podemos crear los conocidos Groundtracks de movimiento.

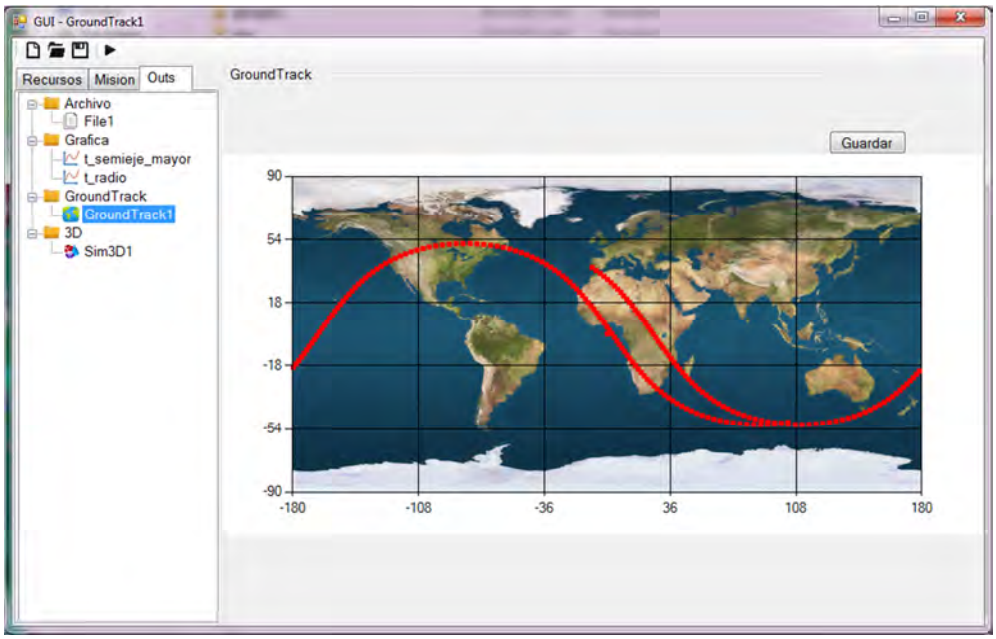


Figura VIII.6. Vista previa de resultados de tipo Groundtrack en interfaz de usuario

Finalmente y si lo deseamos, podemos ejecutar la simulación en 3D mediante el simulador que se ha desarrollado específico para esto, que crea los objetos en tres dimensiones y les asigna una textura esférica del planeta –o luna- que estemos tratando.

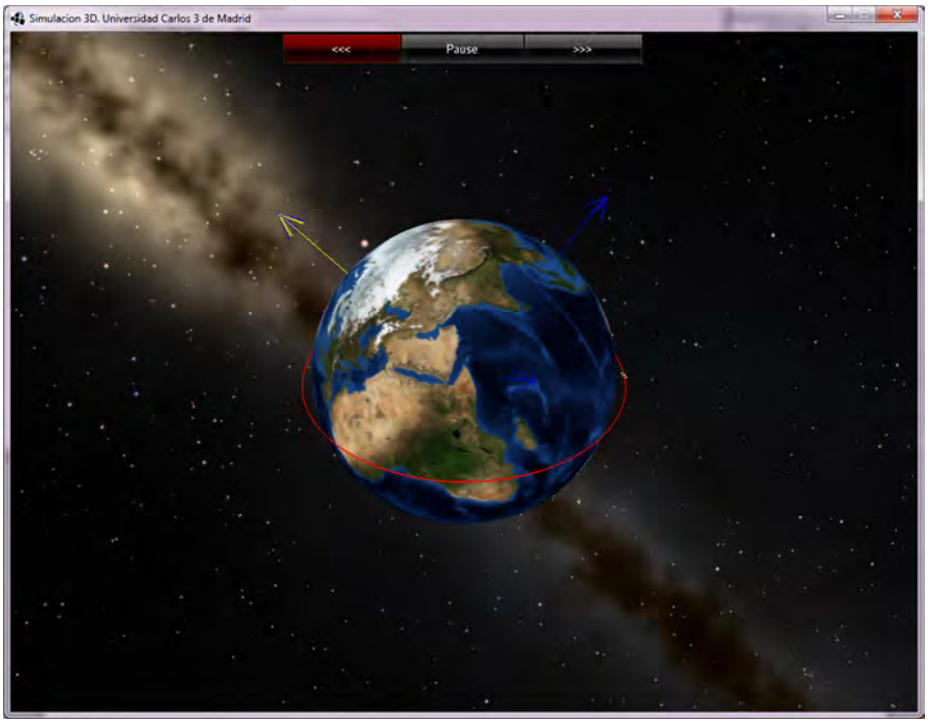


Figura VIII.7. Vista previa de resultados en módulo de simulación 3D

Si lo preferimos, podemos observar la simulación 3D mediante la herramienta que hemos habilitado para exportar los resultados a Google Earth y que nos garantiza mayor calidad de visionado.

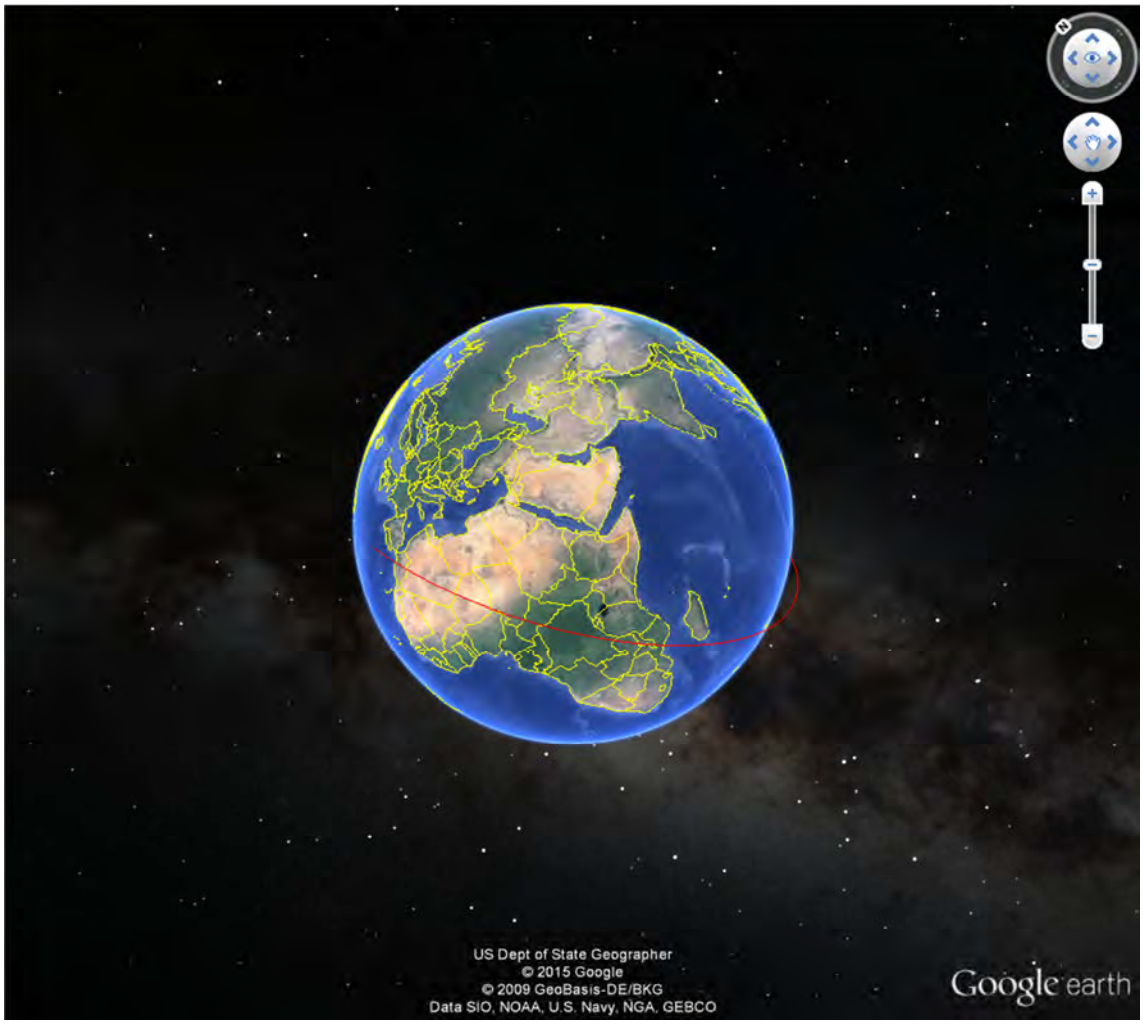


Figura VIII.8. Vista previa de resultados en Google Earth

Hemos habilitado del mismo modo la posibilidad de realizar todos los cálculos en un entorno lunar. No obstante, para el entorno lunar no se tienen los datos de perturbaciones como para el entorno terrestre por lo que los resultados serán menos ajustados.

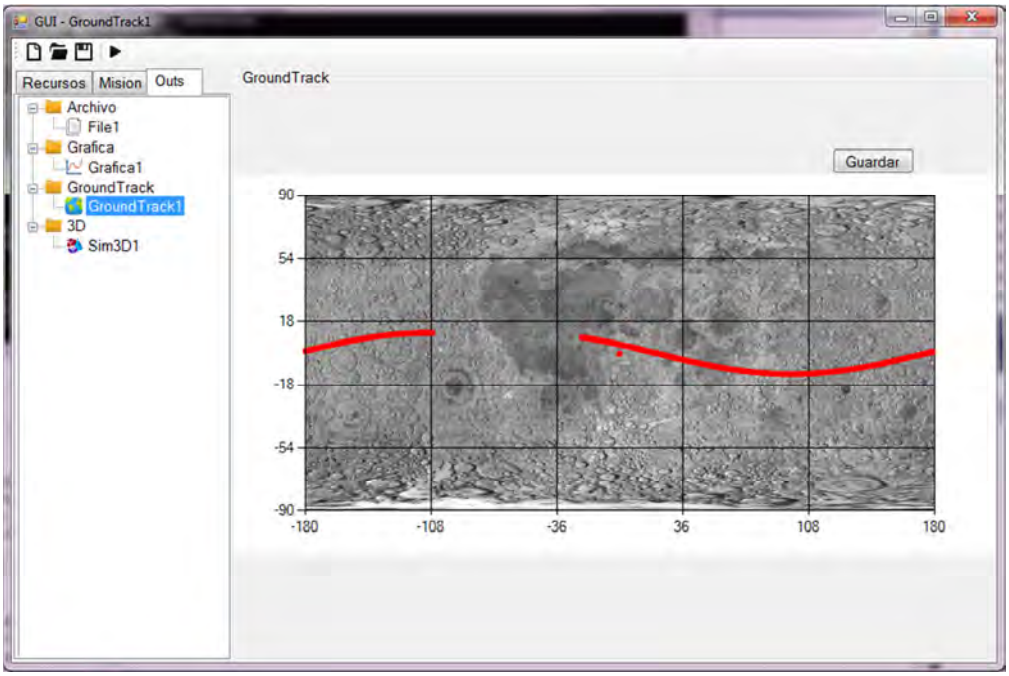


Figura VIII.9. Vista previa de resultados de tipo Groundtrack en interfaz de usuario

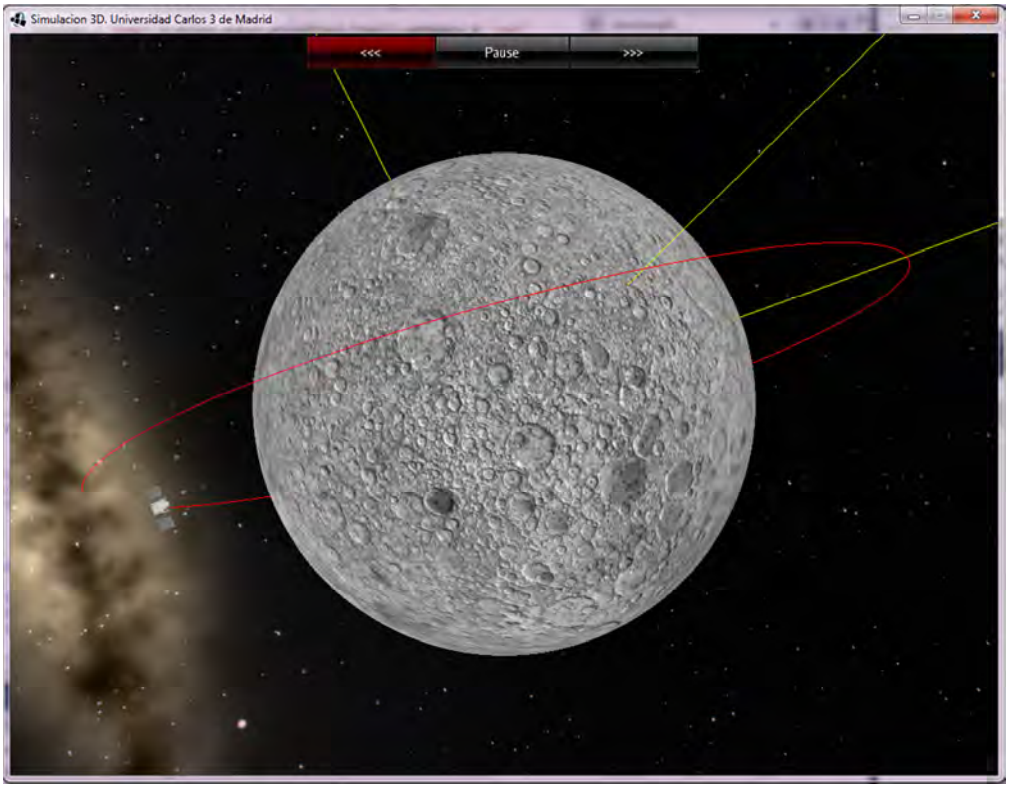


Figura VIII.10. Vista previa de resultados en módulo de simulación 3D

2. Misiones

La propagación de órbitas es la operación básica que resuelve nuestro simulador. No obstante es posible encadenar *steps* de cálculos en los cuales realizamos varias propagaciones de manera continuada donde las condiciones finales del primer paso son las iniciales del segundo, con el objetivo de poder realizar maniobras.

Para ello se cuenta con los elementos llamados *motores* que introducen dentro de la ecuación diferencial fuerzas constantes que incrementan tanto posición como velocidad, logrando realizar maniobras en los satélites.

Estos elementos añaden una complejidad extra al cálculo debido a que nuestro simulador sólo actúa con las variables de posición y velocidad, y no calcula en ningún caso los ángulos de la nave así como sus variaciones con el tiempo. De modo que nuestro campo de aplicación queda muy reducido en este sentido, puesto que no podemos *dirigir* la fuerza hacia un punto que deseemos relativo a la nave. No obstante la solución que hemos adoptado para paliar este problema consiste en aplicar la fuerza de los propulsores respecto a unas direcciones que vienen predefinidas y el simulador se encargará de calcular convenientemente en cada caso como corresponda. Las direcciones que hemos habilitado son las que se muestran en la siguiente figura.

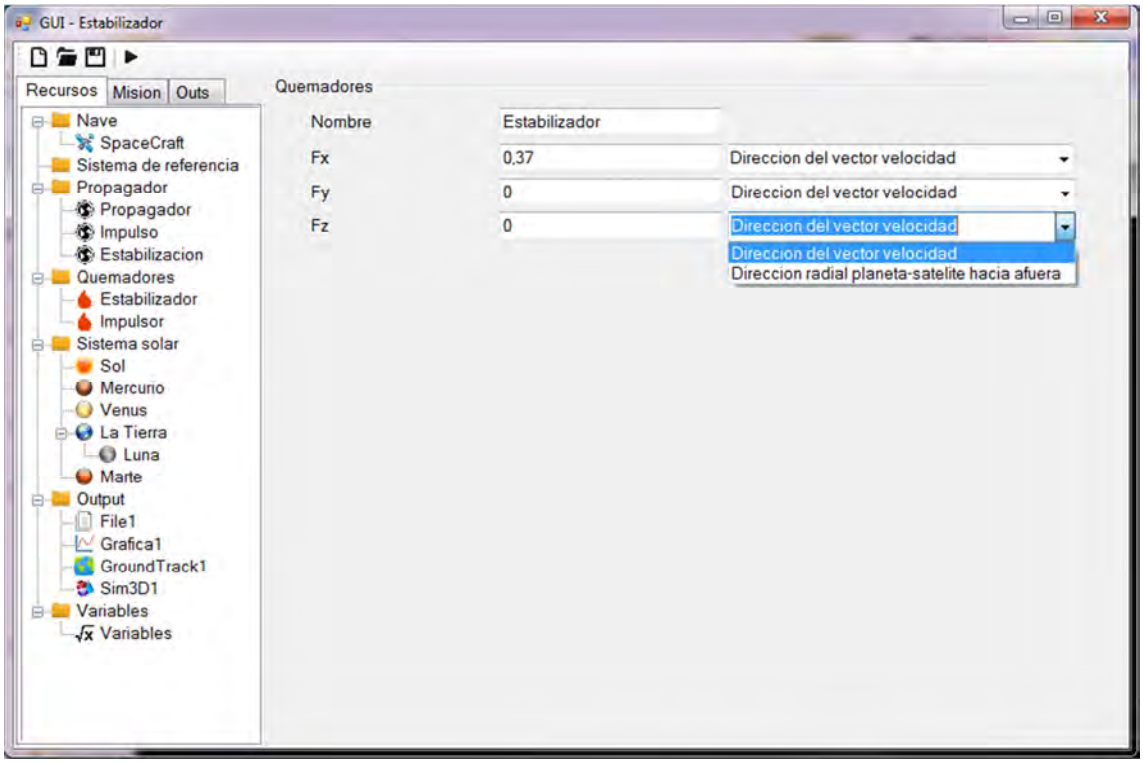


Figura VIII.11. Vista previa de apartado de Motores en la interfaz de usuario

Una vez seleccionadas las fuerzas y las direcciones de nuestros propulsores, deberán ser asociadas con un propagador. Con este fin se ha creado un propagador para cada motor y además uno extra para que se mueva sin incluir fuerzas adicionales. Pinchando en cada uno de los propagadores se podrá vincular a un motor en la pestaña Maniobras.

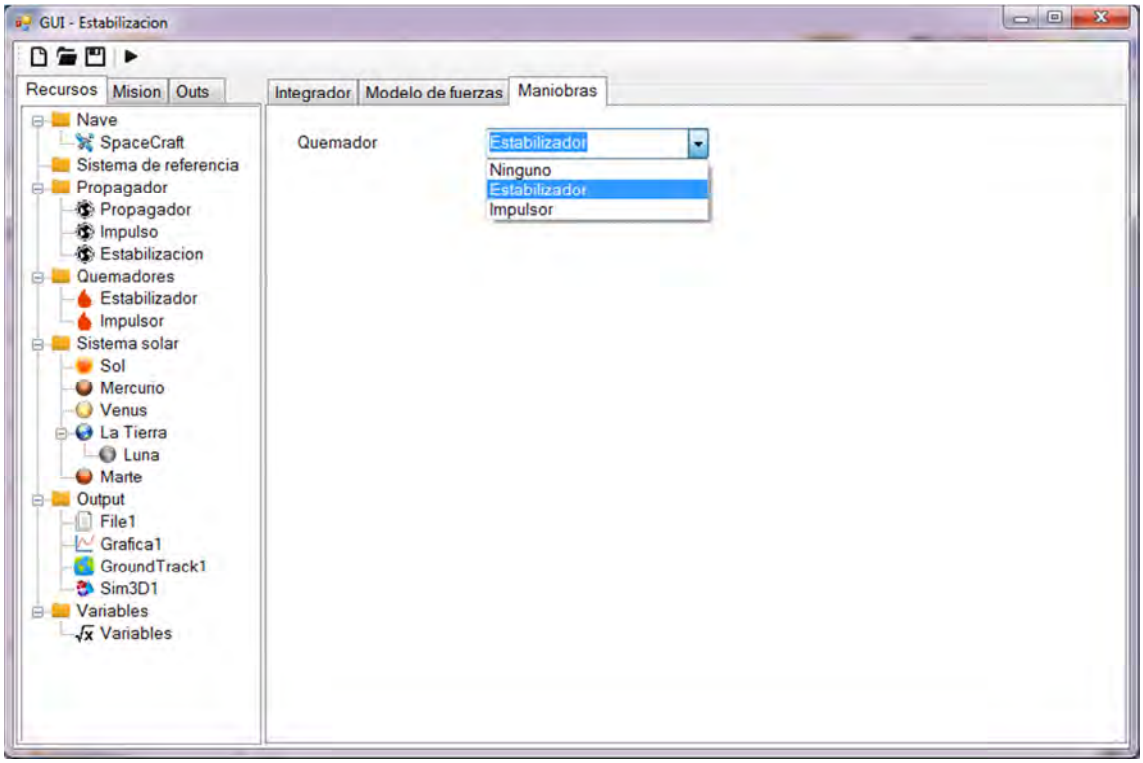


Figura VIII.12. Vista previa de apartado de Maniobras en la interfaz de usuario

Una vez que estén los propagadores asociados a sus respectivos motores, en la pestaña misión podremos añadir tantos tramos o steps como deseemos. Para cada uno de los steps se puede seleccionar un propagador –que a su vez llevará asociado un motor- además de las condiciones de parada para ese tramo. Entre las opciones que ofrecemos se encuentran la de alcanzar el periaxis, el apoaxis, o terminar el tiempo establecido.

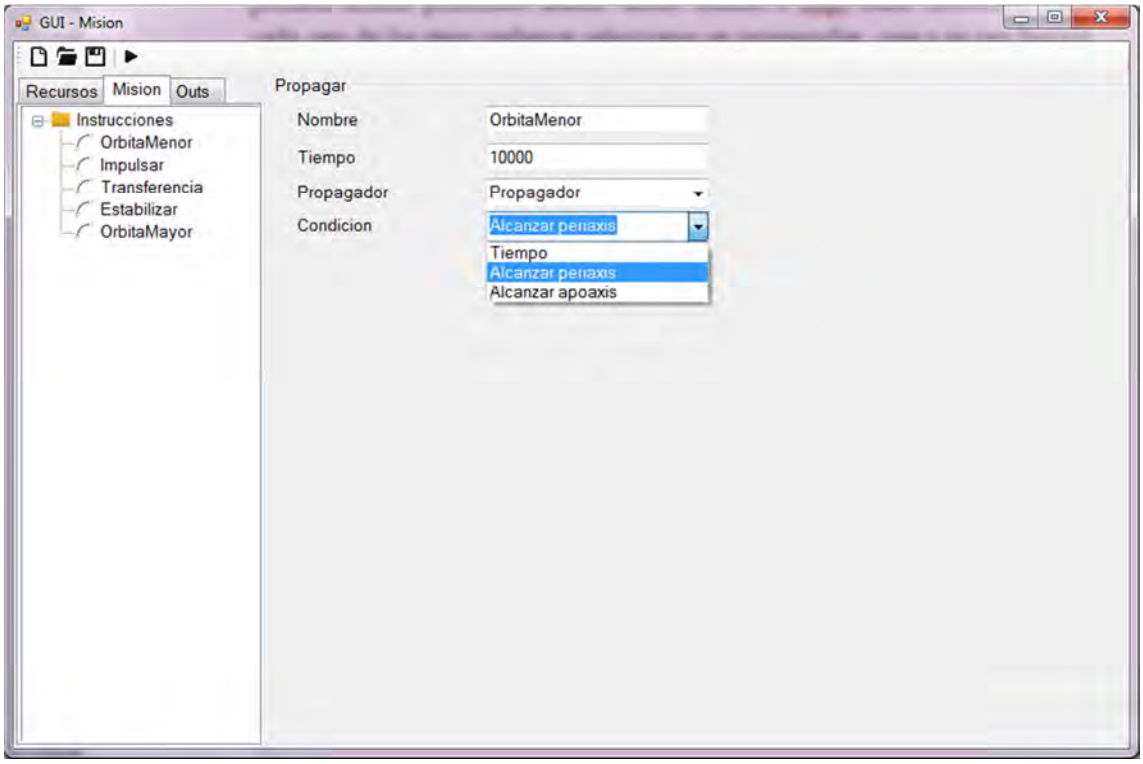


Figura VIII.13. Vista previa de apartado Mision en la interfaz de usuario

Una vez realizada la selección de steps con sus maniobras, ejecutamos la simulación. Esta vez se mostrará en tiempo real cuándo termina cada tramo y qué tiempo de simulación ha representado –recordemos que si seleccionamos alcanzar periaxis, el tiempo será una incógnita que el simulador nos devolverá-.

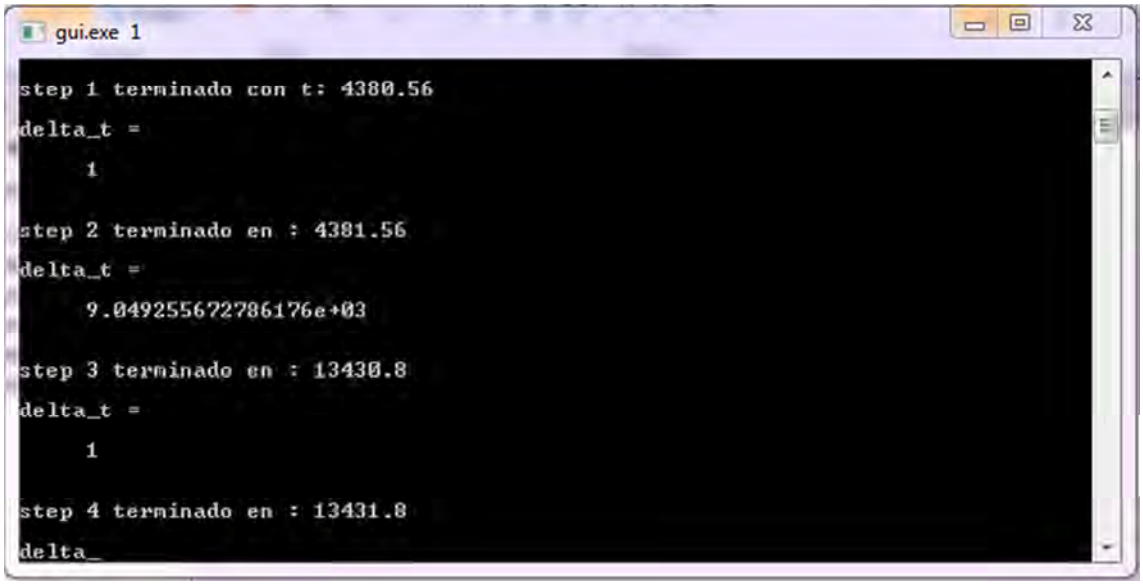


Figura VIII.14. Vista previa de módulo de simulación multi-step

Cuando termine la simulación nos devolverá a la GUI en modo Outputs y podremos ver las gráficas o archivos seleccionados como los que se muestran a continuación.

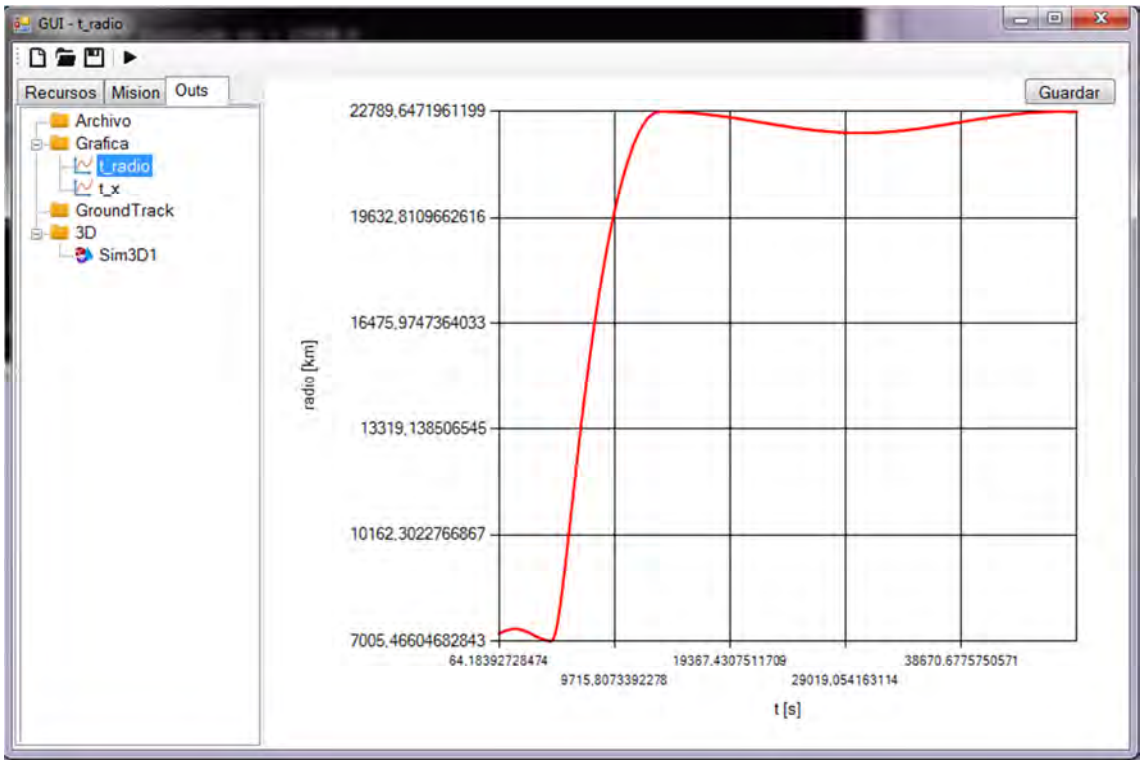


Figura VIII.15. Vista previa de resultados de tipo Gráfica para misión multi-step

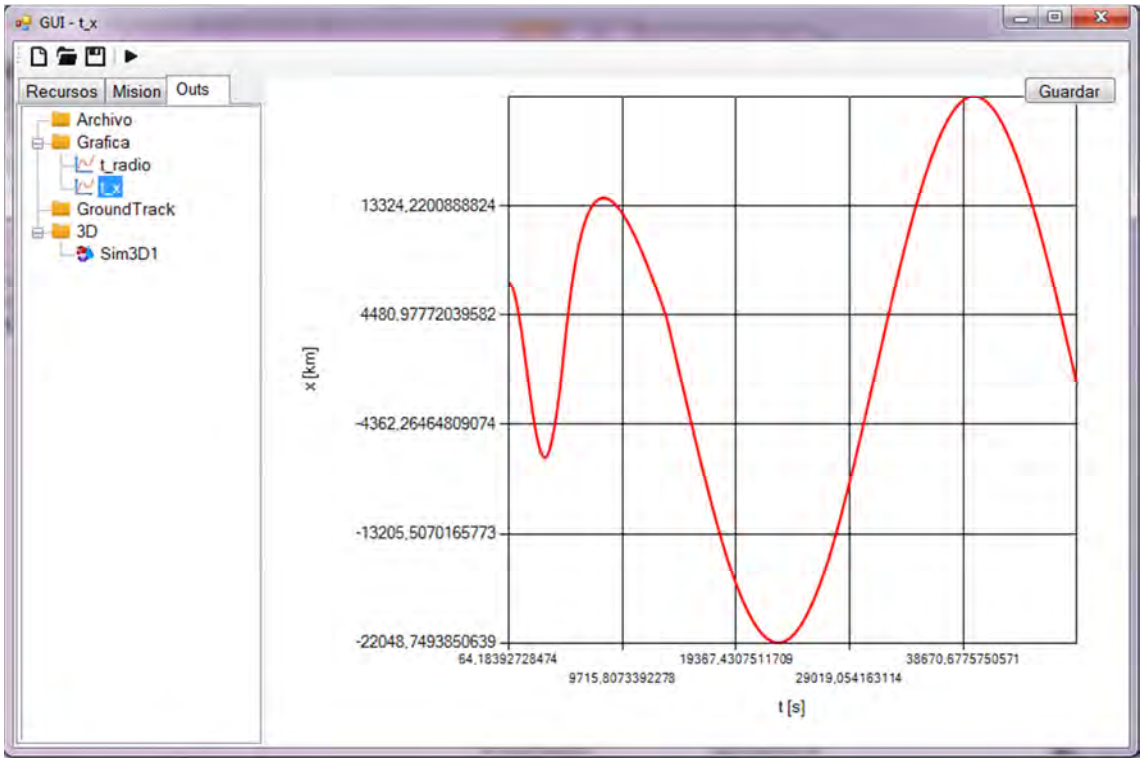


Figura VIII.16. Vista previa de resultados de tipo Gráfica para misión multi-step

Finalmente, para este tipo de misiones, como mejor se observan las maniobras será mediante la simulación en 3D donde podremos apreciar la trayectoria en su totalidad.

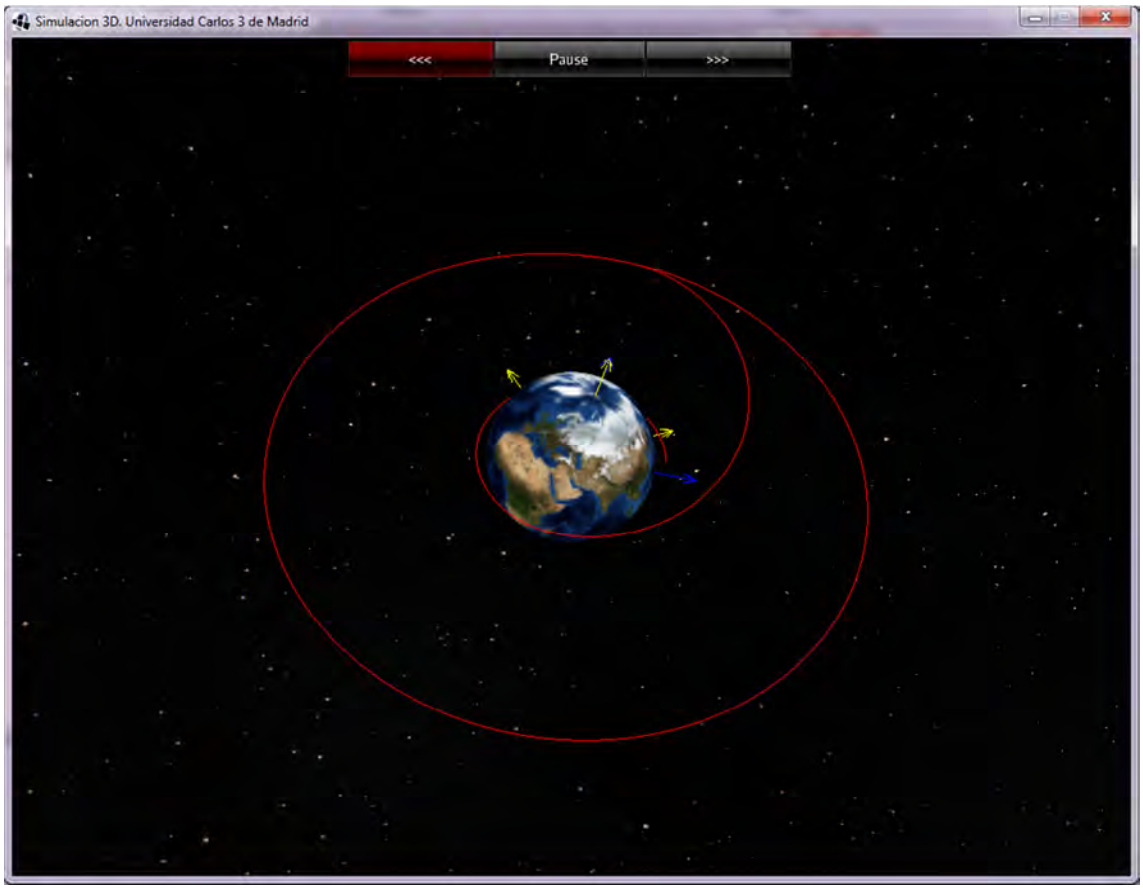


Figura VIII.17. Vista previa de resultados en módulo de simulación 3D para proceso multi-step

3. Leyes de control. Ejemplo de regulador de longitud de satélite geoestacionario

Para acompañar al programa se ha implementado un pequeño ejemplo de cómo utilizar el módulo de leyes de control. Este ejemplo dista de ser perfecto en cuanto a la obtención de resultados no obstante nos ayudará a entender cómo estructurar nuestro código para futuros desarrolladores.

Lo primero de todo que necesitamos hacer es activar el módulo de leyes de control en la interfaz de usuario, en el objeto propagador como muestra la imagen:

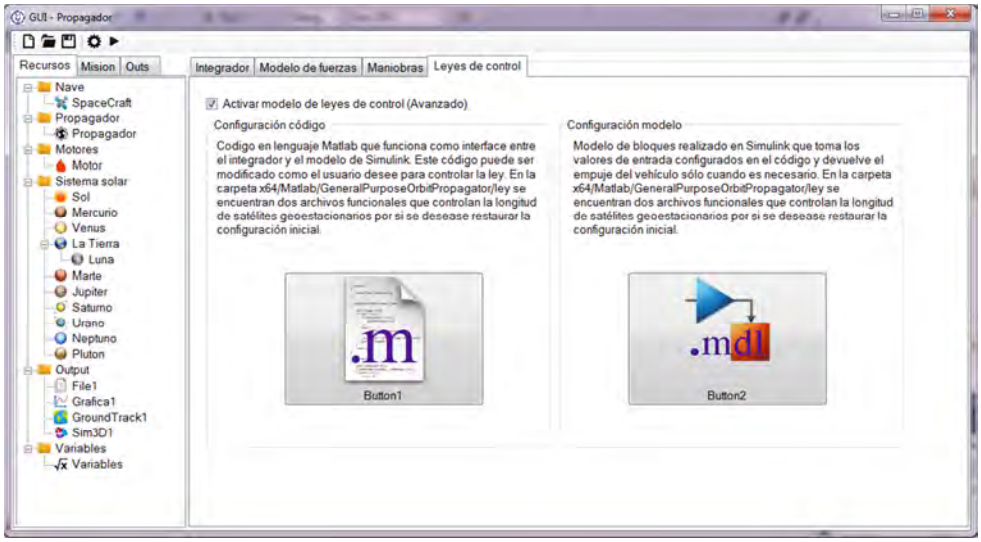


Figura VIII.18.Activación de módulo de leyes de control en la GUI

Si activamos este check, al realizar la simulación se procederá a evaluar la ley de control a cada ciclo.

El objetivo de este regulador es conseguir que satélites geoestacionarios que, por efecto de perturbaciones, se salen de la longitud preestablecida corrijan su posición mediante un empuje. Para el ejemplo se ha fijado la longitud del satélite en 70 grados y el margen para activar los motores de empuje en ± 5 grados. La ventana por tanto se encontrará entre los valores [65,75] grados.

Para reducir el tiempo de simulación hemos establecido que se realicen 100 comprobaciones cada día de modo que el tiempo de chequeo será $t_{check} = 864s$. Se ha seleccionado del mismo modo un tiempo de maniobra de 100 segundos. Esto implica que cuando el satélite se salga de la ventana se aplicará el empuje establecido en la ley de control durante 100 segundos para corregir la posición.

El código escrito dentro del archivo `leyes_control.m` es el siguiente:

```
function [ empuje ] = leyes_control(et, state, mu)
% et                tiempo de efeméride [s]
% state             vector de 7 filas: 3xPosicion + 3xVelocidad +
1xMasa

t_check =864;     %Frecuencia de checkeo de la ley de control [s]
t_man = 100;        %Tiempo de maniobra [s]

%% variables de tipo static que permanecen igual a cada paso
%% a no ser que sean cambiadas
persistent t_ult_check;
if(isempty(t_ult_check))
    t_ult_check = 0;
end

persistent t_inicio_man;
if(isempty(t_inicio_man))
    t_inicio_man = 0;
end

persistent a_ley;
if(isempty(a_ley))
    a_ley = [0;0;0];
end

if et-t_ult_check>=t_check
    t_ult_check = et;
    estado_fixed = j2000toFixed(state, et);
    [~, longitud, ~] = cspice_reclat(estado_fixed(1:3));
    longitud = longitud*180/pi;
    elem_orbitales = cspice_oscelt(state(1:6), et, mu);
    ta = elem_orbitales(6,:);
    ecc = elem_orbitales(2,:);
          %%Para cada bloque FromWorkspace del diagrama .mdl podemos
      hacer set una variable.
    set_variable_in_ley_control(longitud, 'long');
    set_variable_in_ley_control(ta, 'ta');
    set_variable_in_ley_control(ecc, 'ecc');
    %%realizamos la simulacion y asignamos las salidas a variables
    y = get_variable_out_ley_control('ley');
    flag = y(1);
    fuerza = y(2);
    %%Si hubiese más salidas y2 = y(2), y3 = y(3) etc
    if flag == 1
        m = state(7);
        dir = state(4:6)./norm( state(4:6));
        a_ley = fuerza*dir./m;
        t_inicio_man = et;
    end
end
if et - t_inicio_man>= t_man
    a_ley = [0;0;0];
end
empuje = a_ley;
end
```

En el código observamos que nuestro modelo tiene como variables de entrada la longitud, la anomalía verdadera y la excentricidad, previamente calculadas. A continuación se realiza la simulación y si ésta nos devuelve un `flag` igual a uno, la variable `a_ley` tomará el valor del empuje en la dirección de la velocidad y dividida por la masa –lo que equivale a la aceleración-. Una vez transcurrido el tiempo de maniobra, la variable `a_ley` será reiniciada a `[0;0;0]`.

El diagrama de bloques para este ejemplo se ha optado sencillamente por una función que toma variables de entrada y devuelve la salida. Esta sería la opción fácil aunque poco eficaz de realizar la ley puesto que si el módulo de leyes de control se creó fue para evitar la programación de funciones en Matlab y habilitar un modo mucho más visual y lógico para resolverlo. No obstante cumple su cometido y nos muestra cómo poder realizar la estructura del diagrama.

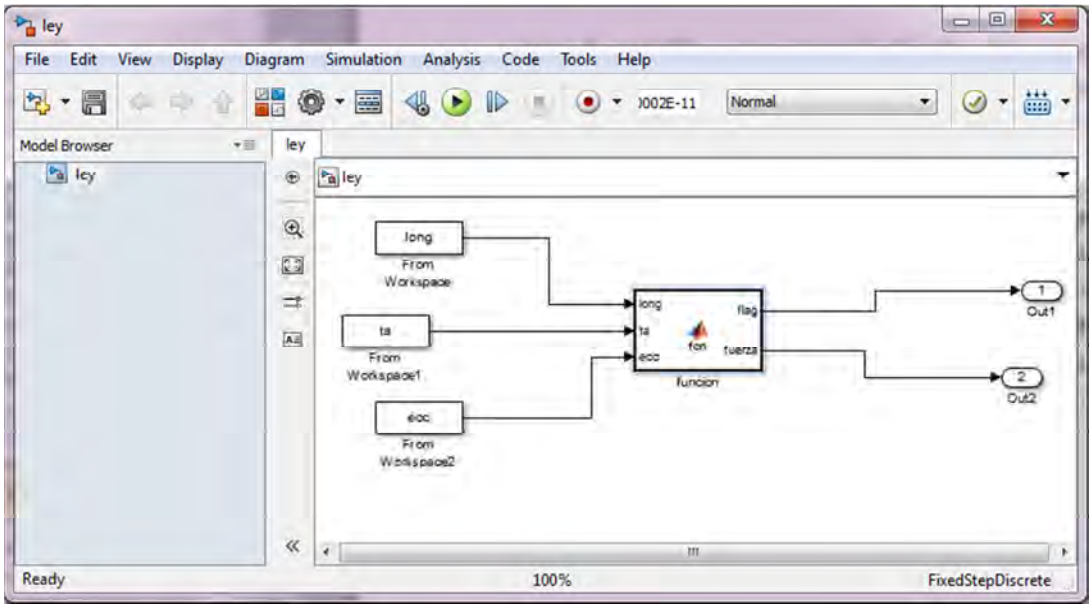


Figura VIII.19. Diagrama de bloques para controlar la longitud de un satélite

El código escrito dentro del bloque `función` del diagrama es el siguiente:

```
function [flag, fuerza]   = fcn(long, ta, ecc)

flag = 0;
fuerza = 0;
longitud_sat = 70;
ventana = 5;

long_sup = 70+ventana;
long_inf = 70-ventana;
```

```
if ecc == 0
    if long>long_sup
       flag = 1;
       fuerza = 0.02;
    end
else
   if long>long_sup && ta> acos(-ecc)
       flag = 1;
       fuerza = 0.02;
    end
end
if long<long_inf && ta> acos(-ecc)
   flag = 1;
   fuerza = -0.02;
end
```

Así pues las salidas `flag` y `fuerza` serán cero a no ser que la latitud se salga de la ventana prevista. Se han introducido otros parámetros de control como la excentricidad o la anomalía verdadera para evitar que a cada maniobra la excentricidad aumente. Esto se consigue realizando la maniobra en los lugares adecuados de la órbita que cumplen la condición que exponemos a continuación.

La condición viene de restringir las ecuaciones planetarias de Gauss a movimientos tangenciales y en particular la referente a la excentricidad:

$$\frac{de}{dt} = \frac{1}{v}\left[2(e + cosf)a_{dt} - \frac{r}{a}sinf\ a_{dn}\right]$$

Siendo f la anomalía verdadera. Puesto que nuestros empujes se realizarán siempre en la dirección tangencial de la órbita, el término a_{dn} se anularía quedándonos para producir variaciones negativas de la excentricidad, la maniobra ha de cumplir que $f > \arccos(-e)$.

El resultado de la aplicación de esta ley de control a una órbita con semieje mayor de $a = 42.000km$ y tiempo de simulación $t = 40\ días$ es el siguiente:

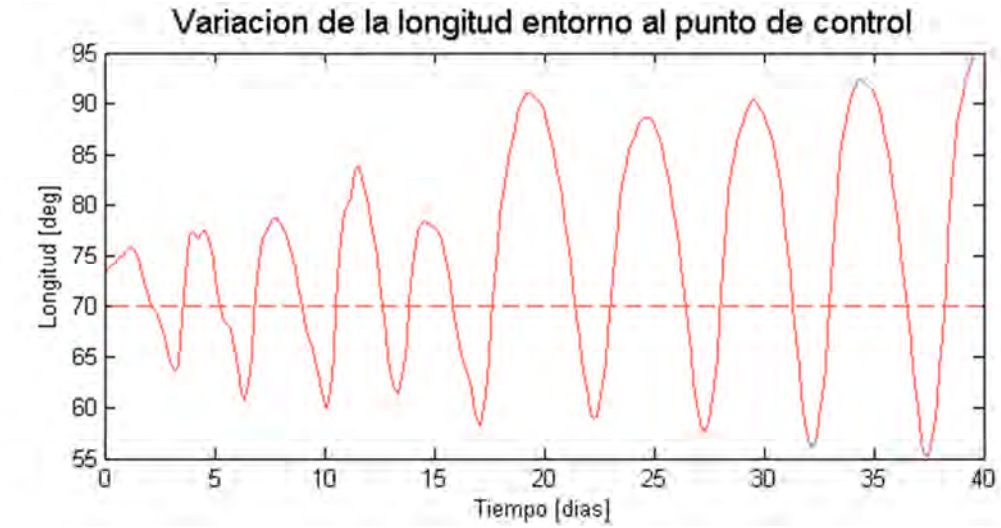


Figura VIII.20. Oscilación de longitud en torno a 70 grados

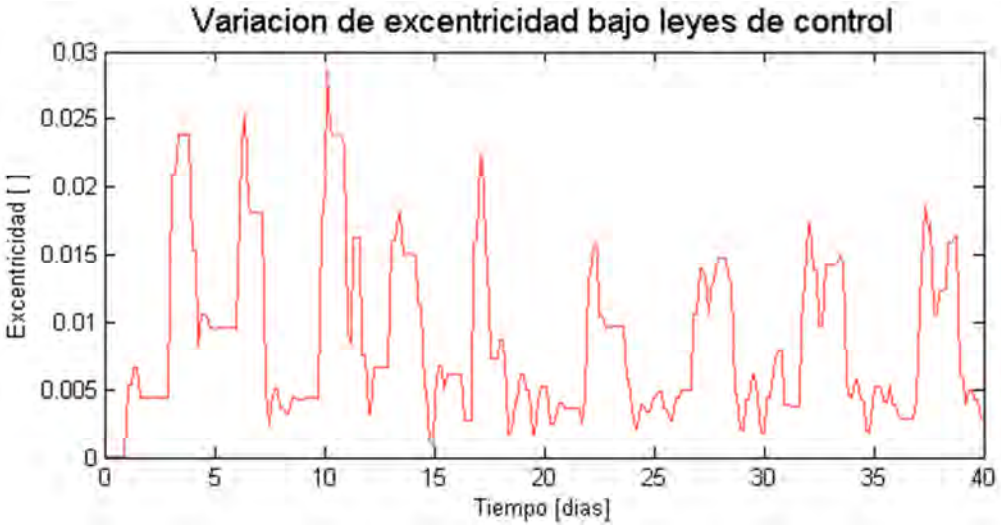


Figura VIII.21. Variación de la excentricidad con el tiempo

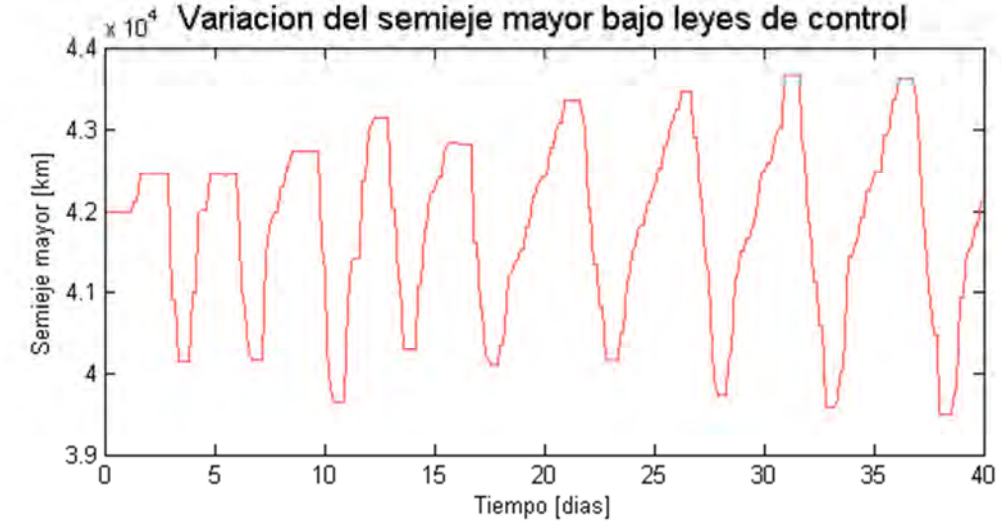


Figura VIII.22. Variación del semieje mayor con el tiempo

Cómo vemos se ha conseguido un control de todas las variables entorno a las condiciones preestablecidas. En la longitud observamos que, a pesar de oscilar entorno a $long = 70\ grados$, esta oscilación va creciendo. Y lo mismo ocurre con el resto de variables. Esto es debido a que la ley de control ha sido realizada como ejemplo de uso y ésta podría ser mejorada de muchas maneras distintas. Como ya se dijo con anterioridad, la calidad de la ley dependerá de la pericia del usuario que la programe. Hay que tener en cuenta que lo ideal que ocurriese con esta ley es que la excentricidad fuese casi todo el tiempo nula, las variaciones del semieje mayor fuesen mucho menores y el tiempo de respuesta al salirse de la ventana fuese menor.

Desde el punto de vista del proyecto se han creado las herramientas necesarias para que el usuario final interesado pueda desarrollar sus propias leyes de control. Este módulo se trata de un apartado experimental que podría ser mejorado en muchísimos aspectos y las posibilidades que ofrece son infinitas y se deja abierto a futuras modificaciones por parte de estudiantes.

IX. Futuras modificaciones

1. Claridad de código

Como ya expusimos al inicio del informe, una parte importante de este proyecto ha sido dejarlo preparado para futuras modificaciones por parte de nuevos alumnos que deseen trabajar con él. Desde el comienzo del desarrollo del software se ha tenido esta idea en mente y se han adoptado las siguientes pautas para facilitar el trabajo futuro:

- Anotaciones en el código con explicaciones de la mayoría de las funciones.
- Nombres de variables y funciones explicativos.
- Grandes funciones para realizar un trabajo complejo.
- Procesos secuenciales a la hora de iniciar el programa.
- Disposición del código homogénea.

Las anotaciones en el código son la mejor ayuda que puede tener un revisor del mismo. Para la mayoría de las funciones y las clases se han incluido varias líneas de explicación con las funcionalidades de las variables y con los resultados que arrojan al final.

Otra manera de facilitar el trabajo a futuros estudiantes es utilizar una nomenclatura explicativa para variables y funciones. De modo que si una función se encarga de mostrar el output de la gráfica del subelemento 'e' de la lista de outputs, la función se llamará

```
Private Sub mostrar_grupo_out_grafica(ByVal e As Integer)
```

Con los nombres de las variables se ha utilizado el mismo procedimiento y en su propia nomenclatura explican su naturaleza. Por otro lado, para los objetos que constituyen la parte visual de la GUI se ha utilizado un sistema de nomenclatura de tipo árbol-rama-hoja. Es decir, que para el cuadro de texto que está disponible en el elemento nave para designar la masa de la misma, se le ha denominado `txtNaveMasa` siendo la primera parte del nombre la naturaleza del objeto –en este caso un cuadro de texto-, la segunda parte el grupo al que pertenece y la tercera el subgrupo o directamente la variable que representa.

Todas estas son técnicas que han de utilizarse desde fases tempranas del desarrollo y que facilitan la compresión del código. Existen otras muchas técnicas distintas que también consiguen este objetivo.

Cuando tratamos de realizar un trabajo complejo y lo subdividimos en muchas funciones ganamos en versatilidad y eficiencia pero quizás complicamos la compresión al lector del código. Es por ello que se ha tomado la decisión de agrupar grandes funciones para realizar un trabajo concreto. Un ejemplo sería la funcion `Public Sub ajustarMarcos()` que se encarga de alinear todos los elementos de la GUI cada vez que ésta se redimensiona. Una simplificación de esta función habría sido crear 200 subfunciones para que alinee cada uno de los objetos únicamente que están a la vista. Desde el punto de vista de la eficiencia habría sido mucho mejor puesto que en nuestra función cada vez que redimensionamos el formulario se reajustan los objetos visibles y también los que no están visibles, no obstante creemos que una gran función para realizar este trabajo es más sencilla de gestionar que 200 pequeñas funciones más eficientes.

Otro método que se ha adoptado a la hora de desarrollar el código ha sido crear procesos secuenciales en el inicio del programa. Con esto, además de hacer el código más claro podemos gestionar los errores de una manera más eficaz, puesto que podemos aislar un gran bloque de la secuencia de inicio y observar los resultados de una manera más clara. En la GUI la secuencia de inicio es la siguiente:

```
Dim modoOutputs As Boolean = False
For Each argument As String In My.Application.CommandLineArgs
    If argument.ToString = "1" Then
       modoOutputs = True
    End If
Next
If modoOutputs Then
   cargarXML("temp\data_in.xml")
   cargarOuts()
   contenedorArbol.SelectedIndex = 2
Else
   mision = New recMision(dir)
   inicializarNodos()
   inicializarObjetos()
   ocultarGrupos()
   ajustarMarcos()
   actualizarArbolRecursos()
   actualizarArbolMision()
End If
```

La primera comprobación que se realiza es si a la GUI se la ha llamado con un parámetro en la línea de comandos. Si es así, la secuencia se iniciará en modo Outputs. Sino, se ejecutará normalmente.

A continuación podemos observar la secuencia de iniciación que contiene grandes funciones que realizan trabajos específicos y los que sería más sencillo

aislar y comprobar errores. Todo esto unido a la nomenclatura descriptiva de cada función consigue en principio gestionar un código más comprensible.

Finalmente, otra técnica utilizada en el desarrollo del proyecto ha sido la disposición del código igual en todos los puntos. Esto significa que para cualquiera de las clases utilizadas o de las funciones, aparecerá siempre, en primer lugar, una pequeña descripción de anotaciones, a continuación la declaración de variables que se utilizarán en esa clase o función y por último el desarrollo de la función. Con esto se pretende conseguir homogeneidad a la hora de utilizar o modificar cualquier función por parte de futuros usuarios.

Mediante estas técnicas esperamos que el futuro estudiante que desee modificar el código obtenga mayores facilidades para ello.

2. Ideas para implementar

Teniendo en cuenta los recursos disponibles, se han descartado funciones muy interesantes que en esta sección se van a detallar, siendo decisión del futuro desarrollador si implementarlas o hacer otras nuevas.

Existen infinidad de mejoras que se podrían realizar sobre este proyecto. Tanto mejoras de lo ya existente como nuevos elementos que conseguirían que el programa fuese mucho más completo. Las modificaciones atañen a distintos niveles. Es posible modificar una parte del código para obtener mejores resultados, o nuevos resultados. Pero también es posible crear partes nuevas, nuevos módulos o bloques que poder 'ensamblar' con los ya existentes y centralizarlos desde la GUI consiguiendo una herramienta mucho más completa que la que en este proyecto se ha desarrollado. Las posibilidades son infinitas y continuación exponemos algunas que podrían ser interesantes.

Conversión en tiempo real de coordenadas keplerianas a cartesianas y viceversa

Una de las funciones más útiles y cómodas que implementa el GMAT es la conversión en tiempo real de coordenadas. Si introduces los datos en coordenadas cartesianas y a continuación en el desplegable seleccionas coordenadas keplerianas, éstas se convierten automáticamente a este nuevo marco de referencia. También ocurre lo mismo haciéndolo a la inversa y realizándolo con cualquiera de las posibles coordenadas que dispone el GMAT.

Añadir base de datos de satélites y coordenadas

Sería muy interesante tener a mano en el apartado referente a la nave una pequeña base de datos de todas las coordenadas de la trayectoria de distintos satélites o astros conocidos de modo que mediante la selección de uno de éstos se auto configurase toda la ficha referente a la nave y a sus condiciones iniciales y época. Esta función podría completarse con la adición de nuevos modelos tridimensionales de cada una de las naves que se añadan a la base de datos.

Añadir la posibilidad de dos planetas en simulación 3D.

Hasta ahora la única posibilidad ideada respecto a la simulación en 3D es la creación de un planeta que será el centro de la imagen y del cual podremos acercarnos y alejarnos, así como girar. No obstante, y entrando en el punto descrito más arriba que ofrece la posibilidad de realizar misiones interplanetarias –como una misión de alunizaje- sería muy interesante tener la oportunidad de crear un entorno con varios planetas sobre los que se recoja la trayectoria de transferencia de uno a otro.

Añadir nuevas variables de cálculo.

Existen infinidad de posibilidades que se podrían añadir y tampoco sería muy cómodo tener que buscar la variable que necesitas entre cientos, pero estaría bien buscar un equilibrio entre el número de variables que el simulador puede calcular y un conjunto completo de posibilidades que recorriesen todos los aspectos geométricos, temporales etc.

Esto se integraría con la implementación de algún sistema de selección de ejes de referencia. Actualmente podemos solicitar al simulador que nos calcule la posición 'x' si buscamos la referencia en ejes inerciales, y 'x_fixed' si necesitamos ejes no inerciales. Esto podría mejorarse eliminando la opción 'fixed' e implementando algún tipo de selector que nos dejase tomar cualquier variable en ejes inerciales y en ejes no inerciales. De modo que reduciríamos el número de variables que se muestran y aumentaríamos las opciones disponibles. Actualmente solo hay dos sistemas, ambos sobre el plano ecuatorial por lo que se podrían añadir los referentes al plano de la elipse, y otros referentes a otros puntos del sistema –como el centro de la vía láctea, o cualquier otro planeta- consiguiendo un abanico mucho más completo en este sentido.

Calculo de propulsión inverso

Una de las opciones más interesantes que tiene GMAT es la del diseño de misiones. Con este programa es posible calcular parámetros de la misión. Un ejemplo sería diseñar una misión en la cual estamos orbitando un una órbita cualquiera y buscamos una transferencia hasta una órbita GEO. Los parámetros que el programa calcularía mediante iteración serían las fuerzas que añadirían los propulsores. La iteración sería ir probando valores de la fuerza tales que al finalizar la maniobra la órbita resultante fuese una GEO. Nuestro programa no permite hacer esto, solo permite darle valores a los propulsores y ver qué ocurrirá con ellos. Por lo tanto se hace muy interesante implementar esta funcionalidad en nuestra GUI.

Crear integrador para dejar de depender de Matlab

El mayor grado de dependencia que tiene nuestro proyecto con otro software de terceros es el relativo a la parte de integración. Como ya hemos explicado con anterioridad, la integración se realiza mediante una función ODE45 de Matlab. Esto implica además que para ejecutar el programa de un modo correcto es necesario tener instalado en el mismo ordenador una versión de Matlab compatible, software que no es precisamente económico. Si consiguiésemos eliminar esta dependencia conseguiríamos tiempos mucho menores de simulación además de suavizar enormemente las exigencias de nuestro ordenador a la hora de correr el programa.

La idea es crear una clase en la que podamos meter la función que posteriormente integraremos y mediante un método seleccionado integrar la expresión. Sería sin lugar a duda una tarea nada trivial, no obstante en mi opinión merecería la pena trabajar en ello.

Gestión de errores

Sería muy interesante que todos los módulos llevasen un gestor de errores que avisase en qué punto del código se ha producido ese error y el motivo. Esta gestión se podría realizar mediante archivos .log.

3. Errores conocidos

Para cerrar este capítulo se van a nombrar algunos errores conocidos que hemos observado en el código y en el comportamiento de los módulos que deberían ser revisados para mejorar el proyecto.

En el apartado de visualización 3D, se decidió añadir la visualización a través de la superposición de la órbita en el entorno de Google Earth que implica el siguiente proceso. Una vez calculada la trayectoria con el simulador se carga un módulo de Matlab denominado Google Earth ToolBox que básicamente convierte una serie de puntos dados en coordenadas esféricas en un archivo KML compatible para que lo ser abierto por el software de Google. Sin embargo esta funcionalidad se ha dejado en fase beta debido a que para trayectorias largas se observan deformaciones en las órbitas con respecto a lo que se observa en el bloque del simulador 3D creado por nosotros. Así por ejemplo, en la transferencia de Hoffman que se ha mostrado en el capítulo de Aplicaciones, ue en nuestro simulador se ve claramente que todas las maniobras están contenidas en el mismo plano, mientras que en la visualización mediante Google Earth no ocurre esto.

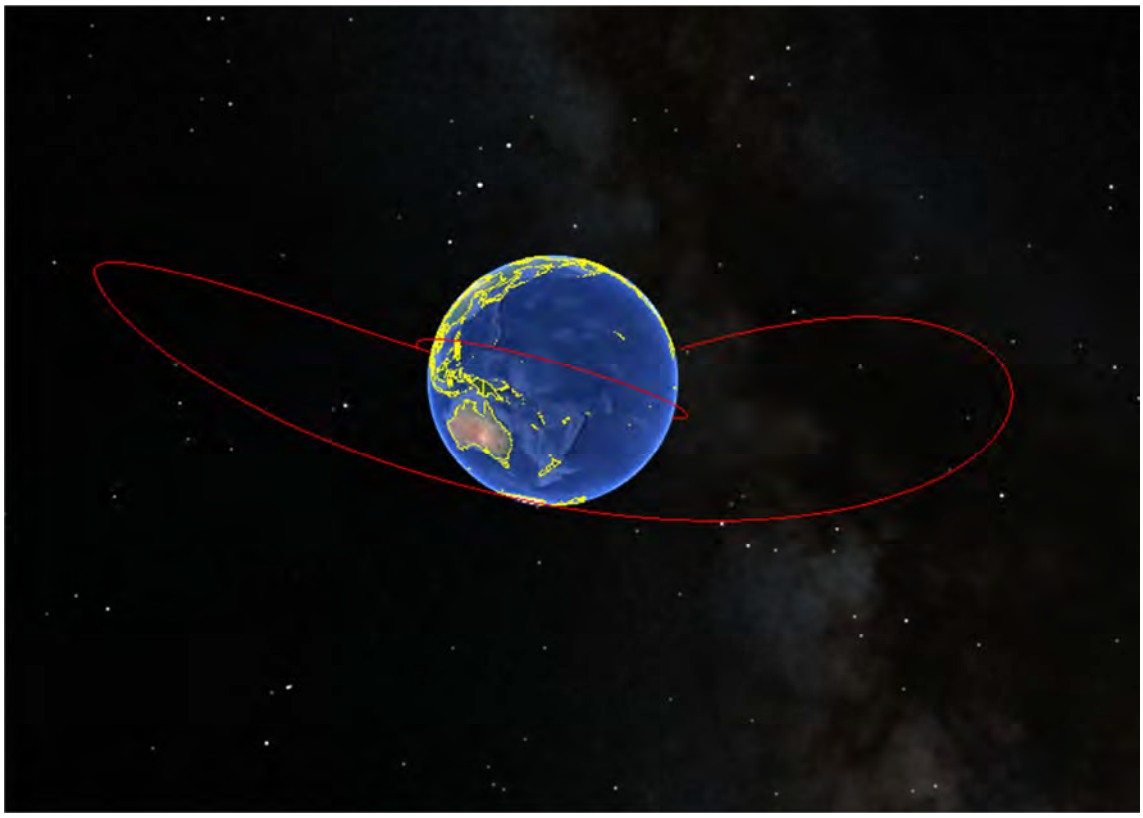


Figura IX.2. Resultado erróneo mostrado por Google Earth

El error puede venir desde varios frentes. El primero sencillamente desde la ToolBox que usamos para crear el archivo KML. Es posible que las coordenadas que necesita no sean las que le estamos enviando. Por otra parte, si nos fijamos en la naturaleza del error es probable que también se deba a que Google Earth trabaja con coordenadas en otro plano de referencia a diferencia de nuestros datos que son siempre referenciados al plano del ecuador. De cualquier modo es necesario retocar parte del código para corregir este problema y por ello se ha decidido dejar esta funcionalidad en Beta.

X. Informe de costes

Sumario de costes		
Materiales		
	Ordenador	990,00 €
Licencias software		
	Licencia comercial estándar Matlab R2013a	2.000,00 €
	Licencia Visual Studio 2013	647,00 €
	Li encia Office 2013	269,00 €
	Licencia Windows 7 Pro	119,00 €
Costes personales		
	Salario anual Ingeniero Aeroespacial Sénior	35.000,00 €
	Seguridad social (26,5%)	9.275,00 €
Coste total proyecto		48.300,00 €

Fuentes de información de costes:

- Licencia comercial de Matlab
  http://es.mathworks.com/pricing-licensing/index.html?intendeduse=comm&prodcode=ML
- Licencia Visual Studio 2013
  http://www.microsoftstore.com/store/mseea/es_ES/pdp/Visual-Studio-Professional-2013/productID.288483200
- Licencia Office 2013
  http://www.microsoftstore.com/store/mseea/es_ES/pdp/Office-Hogar-y-Empresas-2013/productID.263156400
- Licencia Windows 7 Pro
  http://www.microsoftstore.com/store/mseea/es_ES/pdp/Windows-7/productID.288573000
- Salario anual Ingeniero Aeroespacial
  http://www.prospects.ac.uk/aerospace_engineer_salary.htm
- Ordenador
  http://www.pccomponentes.com/

XI. Conclusiones

Con la realización de este proyecto se ha conseguido establecer las líneas generales de una herramienta que puede ser de gran utilidad para futuros estudiantes de esta rama. Se trata de una primera versión del programa que puede ser mejorada en infinidad de aspectos con nuevas funcionalidades o con correcciones de las que ya tiene. Desde un primer momento no se pretendió crear una herramienta que arrojase valores exactos del comportamiento de un cuerpo sometido a las distintas fuerzas gravitacionales, pero sí el elaborar un método para obtener buenas aproximaciones que sirviesen para comprender en una primera aproximación las leyes a las que están sometidos los astros.

El proyecto ha abarcado muchos campos interesantes de estudio y ha resultado muy didáctico desde el punto de vista personal. Partiendo de un análisis teórico inicial hemos ido estudiando todas las variantes que se podían presentar durante el uso del programa. De este primer análisis teórico se ha pasado a la aplicación práctica de todas estas fórmulas mediante herramientas numéricas y finalmente a la plasmación de los resultados por medio de gráficos y otras herramientas. Finalmente se han sometido a un análisis todos los resultados obtenidos por el proyecto siendo éstos muy satisfactorios.

Desde el punto de vista personal este proyecto ha resultado realmente enriquecedor debido a la cantidad de elementos que lo componían, cada uno ellos muy distinto del resto pero todos relacionados entre sí. Se trata de un proyecto que toca distintos campos pasando por la teoría de movimientos orbitales, el desarrollo de aplicaciones en distintos lenguajes o uso de interfaces entre módulos, entre otros muchos. Esta gran variedad de tareas ha conseguido la total implicación personal así como la compresión exhaustiva de todos los elementos que conforman el proyecto.

XII. Referencias

AGI. (s.f.). *STK AGI Software to model analyze and visualize space defense and intelligence systems*. Recuperado el 10 de 12 de 2014, de http://www.agi.com/products/stk/

Cornell University. Deparment of Astronomy. (s.f.). *Possible Operational Orbits*. Recuperado el 26 de 12 de 2014, de http://www.astro.cornell.edu/~berthoud/alpsat/chapter4a.html

Dr. Isaac Skog, P. H. (2012). *State of the art in Car Navigation: an Overview.*

ESA. (s.f.). *Earth Gravity revealed in unprecedented detail*. Recuperado el 19 de 12 de 2014, de http://www.esa.int/Our_Activities/Observing_the_Earth/GOCE/Earth_s_gravity_revealed_in_unprecedented_detail

FGZ Helmholtz Centre Postdam. (s.f.). *International Centre for Global Earth Models*. Recuperado el 01 de 02 de 2015, de http://icgem.gfz-potsdam.de/ICGEM/

Institute for Biodiversity and Ecosystem Dynamics. University of Amsterdam. (s.f.). *Google Earth Toolbox*. Recuperado el 01 de 02 de 2015, de http://ibed.uva.nl/

JHT's Planetary Pixel Emporium. (s.f.). *Repositorio de imágenes de planetas*. Recuperado el 14 de 12 de 2014, de http://planetpixelemporium.com/earth.html

Mathworks. (s.f.). *Matlab Engine*. Recuperado el 18 de 11 de 2014, de http://es.mathworks.com/help/matlab/programming-interfaces-for-c-c-fortran-com.html

McNish, L. (s.f.). *RASC Calgary Centre – Planetary Orbits*. Recuperado el 25 de 10 de 2014, de http://calgary.rasc.ca/orbits.htm

MonkeyEngine. (s.f.). *A cross-platform game engine for adventurous Java developers.* Obtenido de http://jmonkeyengine.org/

NASA. (s.f.). *General Mission Analysis Tool (GMAT) Software de la NASA para simulación de orbitas*. Recuperado el 10 de 12 de 2014, de http://gmatcentral.org/display/GW/GMAT+Wiki+Home]

NASA. (s.f.). *GMAT Coordinate Systems Documentation Resources*. Recuperado el 15 de 11 de 2014, de http://gmat.sourceforge.net/doc/R2014a/html/CoordinateSystem.html

NASA. (s.f.). *Repositorio imágenes de La Tierra*. Recuperado el 14 de 12 de 2014, de http://visibleearth.nasa.gov/view_cat.php?categoryID=1484

NASA. (s.f.). *The Navigation and Ancillary Information Facility*. Recuperado el 17 de 11 de 2014, de http://naif.jpl.nasa.gov/naif/toolkit.html

O. Montenbruck, E. G. (2000). *Satellite Orbtis, Models, Methods and applications.* Springer.

Orbiter. (s.f.). *Orbiter Space Flight Simulator*. Recuperado el 10 de 12 de 2014, de http://orbit.medphys.ucl.ac.uk/

Pardo, D. G. (2014). *Analysis and Implementation of Gravity Fields Models For Planet And Asteroids.* Madrid.

Pole Cats. (s.f.). *Coordinate Systems*. Recuperado el 13 de 01 de 2015, de https://rexuscats.wikispaces.com/Coordinate+Systems

Thomason, L. (s.f.). *Tiny XML*. Recuperado el 14 de 01 de 2015, de http://www.grinninglizard.com/tinyxml/

Vallado, D. A. (s.f.). *Fundamentals of Astrodynamics and Applications.* Space Technology Library.

XIII. Anexos

1. Conversión entre elementos orbitales y posición y velocidad.

A continuación se muestra como obtener las coordenadas del vector posición $\vec{r}$ y el vector velocidad $\vec{v}$ a partir de unos elementos orbitales dados.

El set tradicional de seis elementos dados es el siguiente:

- Semieje mayor a $[m]$
- Excentricidad e [ ]
- Argumento de perigeo ω $[rad]$
- Longitud de nodo ascendente Ω $[rad]$
- Inclinación i $[rad]$
- Anomalía media M_0 $[rad]$ referida a la época t_0 $[JD]$

Consideraremos el cálculo en una época t distinta a t_0 . Además supondremos conocido el parámetro de gravitación estándar referente a ese cuerpo central μ.

1. Calculo de anomalía media M $[rad]$

   Comenzamos calculando la variación de tiempo en segundos:

   $$\Delta t = 24 \cdot 3600(t - t_0)$$

   La anomalía media la calculamos como:

   $$M(t) = M_0 + \Delta t\sqrt{\frac{\mu}{a^3}}$$

   Con a la longitud del semieje mayor de la elipse y $\mu = G \cdot M$ siendo M la masa del cuerpo sobre el que se orbita. A continuación normalizamos $M(t)$ para que entre dentro del intervalo $[0, 2\pi)$.

2. Resolver ecuación de Kepler

   Planteamos un método iterativo para resolver la ecuación de Kepler y calcular $E(t)$:

   $$M(t) = E(t) - e \sin E(t)$$

   Siendo e la excentricidad de la elipse. El método utilizado en este proyecto es el método de Newton-Raphson. Para este método resolveremos iterativamente la siguiente sucesión estableciendo $E_0 = M$:

$$E_{j+1} = E_j - \frac{E_j - e\sin E_j - M}{1 - e\cos E_j}$$

Para cada ciclo de cálculo, estimamos la variación entre elementos consecutivos $\varepsilon = \left| E_{j+1} - E_j \right|$ y cuando converja lo suficiente ($\varepsilon < 10^{-9}$) terminaremos la iteración.

3. Calculo de anomalía verdadera

Calculamos la anomalía verdadera utilizando la función $arctan2$ definida matemáticamente como:

$$arctan2(y,x) = \begin{cases} \arctan\left(\frac{y}{x}\right) & x > 0 \\ \arctan\left(\frac{y}{x}\right) + \pi & y \geq 0, x < 0 \\ \arctan\left(\frac{y}{x}\right) - \pi & y < 0, x < 0 \\ +\frac{\pi}{2} & y > 0, x = 0 \\ -\frac{\pi}{2} & y < 0, x = 0 \\ indefinido & y = 0, x = 0 \end{cases}$$

La anomalía verdadera se calcula con la siguiente expresión:

$$v(t) = 2 \cdot arctan2\left(\sqrt{1+e}\sin\frac{E(t)}{2}, \sqrt{1-e}\cos\frac{E(t)}{2}\right)$$

4. Obtención de distancia al cuerpo

$$r_c(t) = a(1 - e\cos E(t))$$

5. Obtención de vector velocidad y posición en el plano de referencia

Antes de aplicar la matriz de giro del plano de la elipse, calculamos la velocidad y posición en la elipse sobre el plano de referencia:

$$\overrightarrow{r_{ref}}(t) = r_c(t)\begin{pmatrix} \cos v(t) \\ \sin v(t) \\ 0 \end{pmatrix}$$

$$\overrightarrow{v_{ref}}(t) = \frac{\sqrt{\mu a}}{r_c(t)} \begin{pmatrix} -\sin E(t) \\ \sqrt{1-e^2}\cos E(t) \\ 0 \end{pmatrix}$$

6. Obtención de vector velocidad y posición en el plano de la órbita.

Finalmente aplicamos la matriz de giro obteniendo:

$$\vec{r}(t) = r_{ref,x}(t) \begin{pmatrix} \cos\omega\cos\Omega - \sin\omega\cos i\sin\Omega \\ \cos\omega\sin\Omega + \sin\omega\cos i\cos\Omega \\ \sin\omega\sin i \end{pmatrix} + r_{ref,y}(t) \begin{pmatrix} -\sin\omega\cos\Omega - \cos\omega\cos i\sin\Omega \\ \cos\omega\cos i\cos\Omega - \sin\omega\sin\Omega \\ \cos\omega\sin i \end{pmatrix}$$

$$\vec{v}(t) = v_{ref,x}(t) \begin{pmatrix} \cos\omega\cos\Omega - \sin\omega\cos i\sin\Omega \\ \cos\omega\sin\Omega + \sin\omega\cos i\cos\Omega \\ \sin\omega\sin i \end{pmatrix} + v_{ref,y}(t) \begin{pmatrix} -\sin\omega\cos\Omega - \cos\omega\cos i\sin\Omega \\ \cos\omega\cos i\cos\Omega - \sin\omega\sin\Omega \\ \cos\omega\sin i \end{pmatrix}$$

2. Conversión entre vectores de posición y velocidad y elementos orbitales

A continuación mostramos el método de conversión del conjunto de los seis elementos orbitales normalmente utilizados a partir de los vectores de posición $\vec{r}(t)$ y velocidad $\vec{v}(t)$.

Supondremos conocido el parámetro de gravitación estándar referente a ese cuerpo central μ.

1. Cálculos previos

   Primeramente calcularemos el vector de momento orbital $\vec{h}$ así como el vector excentricidad $\vec{e}$

$$\vec{h} = \vec{r} \times \vec{v}$$

$$\vec{e} = \frac{\vec{v} \times \vec{h}}{\mu} - \frac{\vec{r}}{|\vec{r}|}$$

   A continuación construimos el vector $\vec{n}$ con las siguientes coordenadas de $\vec{h}$

$$\vec{n} = \left(-h_y\,, h_x\,, 0\right)^T$$

   Finalmente calculamos la anomalía verdadera como:

$$\upsilon = \begin{cases} \arccos\dfrac{\vec{e}\cdot\vec{r}}{|\vec{e}|\cdot|\vec{r}|}, & \vec{r}\cdot\vec{v} \geq 0 \\ 2\pi - \arccos\dfrac{\vec{e}\cdot\vec{r}}{|\vec{e}|\cdot|\vec{r}|}, & \vec{r}\cdot\vec{v} < 0 \end{cases}$$

   Donde la operación $\vec{e}\cdot\vec{r}$ denota el producto escalar de los vectores $\vec{e}\ y\ \vec{r}$

2. Cálculo de la inclinación de la orbita

$$i = \arccos\frac{h_z}{|\vec{h}|}$$

   Siendo h_z la tercera componente del vector de momento orbital.

3. Determinar la excentricidad de la órbita así como el anomalía excéntrica

$$e = |\vec{e}|$$

$$E = 2\arctan\frac{\tan\frac{v}{2}}{\sqrt{\frac{1+e}{1-e}}}$$

4. Obtención de la longitud de nodo ascendente y el argumento de periapsis

$$\Omega = \begin{cases} \arccos\frac{n_x}{|\vec{n}|}, & n_y \geq 0 \\ 2\pi - \arccos\frac{n_x}{|\vec{n}|}, & n_y < 0 \end{cases}$$

$$\omega = \begin{cases} \arccos\frac{\vec{n}\cdot\vec{e}}{|\vec{n}|\cdot|\vec{e}|}, & e_z \geq 0 \\ 2\pi - \arccos\frac{\vec{n}\cdot\vec{e}}{|\vec{n}|\cdot|\vec{e}|}, & e_z < 0 \end{cases}$$

5. Calculo de la anomalía media.

$$M = E - e\sin E$$

6. Calculo del semieje mayor

$$a = \frac{1}{\frac{2}{|\vec{r}|} - \frac{|\vec{v}|^2}{\mu}}$$

3. Configuración de entorno para uso de Matlab Engine

La librería Matlab Engine incluye funciones y rutinas que permiten ejecutar comandos típicos del entorno Matlab sobre otros lenguajes de programación para ser usados en aplicaciones de terceros, permitiendo capacitar a un software de todo el poder que este entorno de lenguaje formal matemático posee.

Los requisitos para poder usar Matlab Engine en un ordenador son primeramente tener instalado el software Matlab en ese ordenador ya que las funciones que incluyas en tu software no tienen la capacidad de realizar por si misma los comandos propios de Matlab, sino que poseen la capacidad de interactuar con el Matlab Engine que está ejecutándose en segundo plano en un proceso separado. Esto tiene como ventaja la reducción de código de la librería que deberás importar a tu software puesto que solo contendrá funciones de interface entre tu software y el Matlab Engine que corre en segundo plano.

En este anexo vamos a tratar de describir el proceso necesario para poder ejecutar comandos del Matlab Engine incluido en la version Matlab R2013a desde una aplicación escrita en lenguaje C desde el entorno de programación de Windows Visual Studio 2013. Si deseamos desarrollar una aplicación con estas capacidades deberemos de tener instalado este IDE de Microsoft y configurarlo como explicamos en el anexo 4.

1. Instalación de un compilador compatible con Matlab

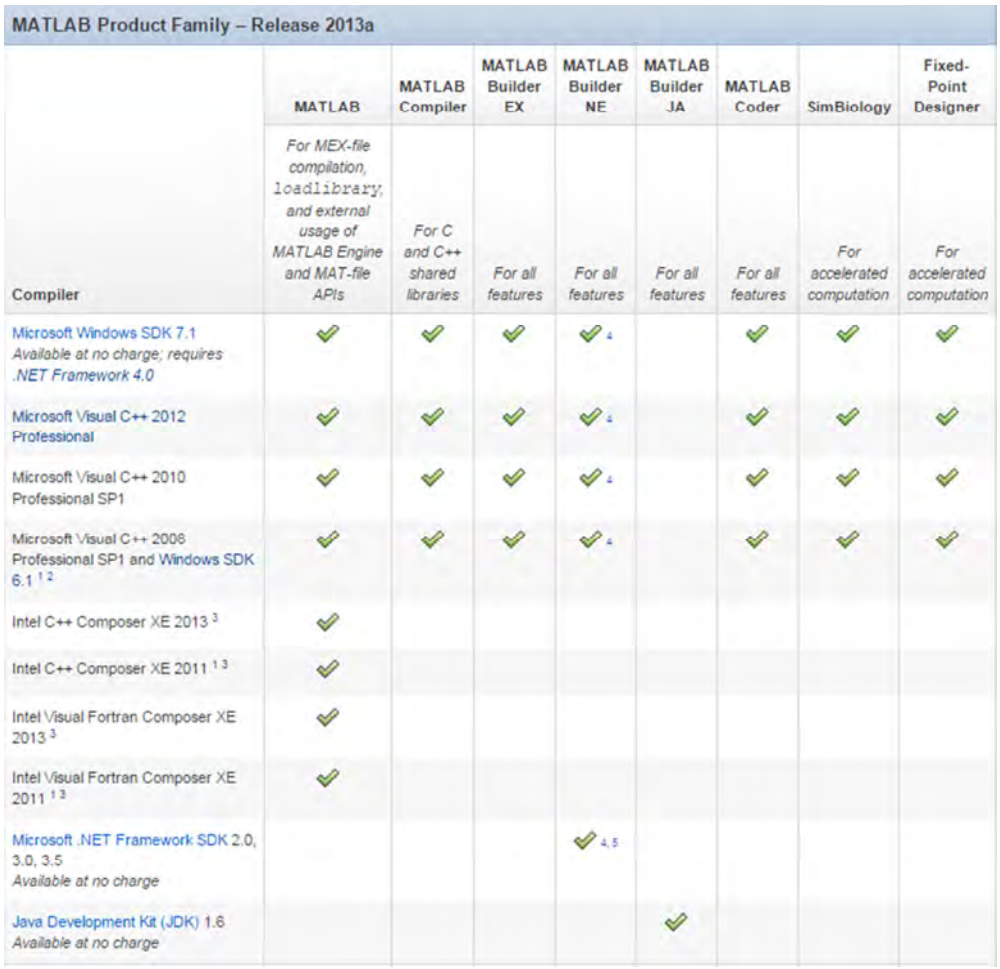
Este paso sólo será necesario si no se tiene instalado previamente un compilador compatible con Matlab. Para comprobarlo abra Matlab y ejecute el siguiente comando:

```
mex -setup
```

Le preguntará entonces si desea buscar compiladores instalados en la computadora a lo que deberá responder que Sí. A continuación se le listarán compiladores instalados en el PC. Si no apareciese ninguno deberá de realizar la instalación que se detalla a continuación. En caso contrario seleccione uno de la lista y pase al punto número 2.

Una vez instalado Matlab, descargamos un compilador compatible para nuestra versión de Matlab. En nuestro caso tenemos los compiladores disponibles en la siguiente URL:

http://es.mathworks.com/support/compilers/R2013a/index.html?sec=win64



MATLAB Product Family – Release 2013a

Compiler	MATLAB *For MEX-file compilation, `loadlibrary`, and external usage of MATLAB Engine and MAT-file APIs*	MATLAB Compiler *For C and C++ shared libraries*	MATLAB Builder EX *For all features*	MATLAB Builder NE *For all features*	MATLAB Builder JA *For all features*	MATLAB Coder *For all features*	SimBiology *For accelerated computation*	Fixed-Point Designer *For accelerated computation*
Microsoft Windows SDK 7.1 *Available at no charge; requires .NET Framework 4.0*	✓	✓	✓	✓ [4]		✓	✓	✓
Microsoft Visual C++ 2012 Professional	✓	✓	✓	✓ [4]		✓	✓	✓
Microsoft Visual C++ 2010 Professional SP1	✓	✓	✓	✓ [4]		✓	✓	✓
Microsoft Visual C++ 2008 Professional SP1 and Windows SDK 6.1 [1] [2]	✓	✓	✓	✓ [4]		✓	✓	✓
Intel C++ Composer XE 2013 [3]	✓							
Intel C++ Composer XE 2011 [1] [3]	✓							
Intel Visual Fortran Composer XE 2013 [3]	✓							
Intel Visual Fortran Composer XE 2011 [1] [3]	✓							
Microsoft .NET Framework SDK 2.0, 3.0, 3.5 *Available at no charge*				✓ [4,5]				
Java Development Kit (JDK) 1.6 *Available at no charge*					✓			

Figura XII.1. Compiladores disponibles

Para el ejemplo hemos descargado el compilador Microsoft Windows SDK 7.1 que requiere tener instalado el paquete .NET Framework 4.0 de Windows. Una vez descargado se instala manteniendo la configuración por defecto.

A continuación desde la consola de Matlab ejecutamos el siguiente comando:

```
mex -setup
```

El sistema nos preguntará si deseamos que Matlab busque los compiladores disponibles. Le diremos que sí. A continuación nos mostrará compiladores disponibles que tenemos instalados en nuestra computadora. Seleccionamos el que acabamos de instalar: Microsoft Software Development Kit (SDK) 7.1.

2. Configuración del entorno de Matlab Engine

A continuación es necesario activar Matlab Engine para que se ejecute en segundo plano y sea capaz de interpretar las funciones que se ejecutan desde nuestro código C. Necesitaremos introducir una variable del sistema nueva. Para ello ejecutamos el comando en Matlab:

```
fullfile(matlabroot,'bin',computer('arch'))
```

Esto nos devolverá una ruta de archivos que en nuestro caso es:

```
C:\Program Files\MATLAB\R2013a\bin\win64
```

A continuación nos dirigimos al"Control panel" de Windows y entramos en "System and security" -> "System" -> "Advance system settings" en donde deberemos pinchar sobre "Environment Variables"

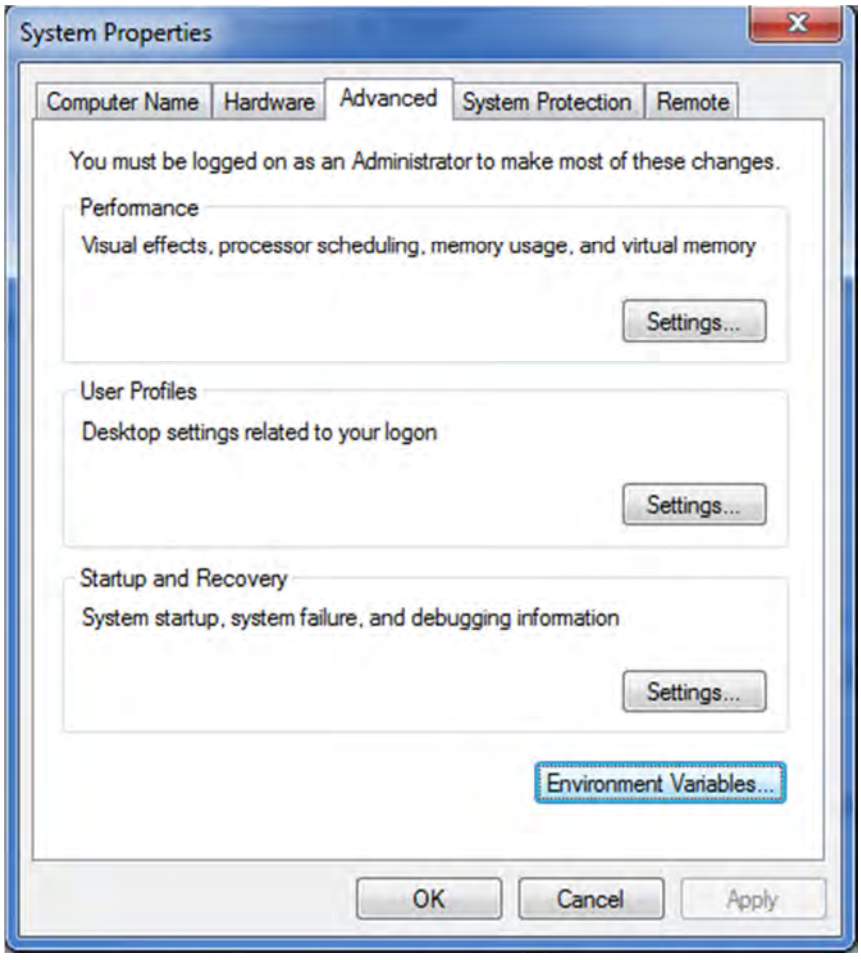


Figura XII.2. Menu de System properties de Windows

Buscamos el campo "Path" en el scroll de "System variables" y hacemos doble click. En el campo "Variable value" deberemos **añadir al final del mismo un**

punto y coma (;) y a continuación la ruta que nos devolvió desde Matlab sin dejar ningún espacio. **Asegúrese de que no elimina lo que estaba anteriormente en este campo** y que simplemente añade algo del estilo "`;C:\Program Files\MATLAB\R2013a\bin\win64`" al final del mismo.

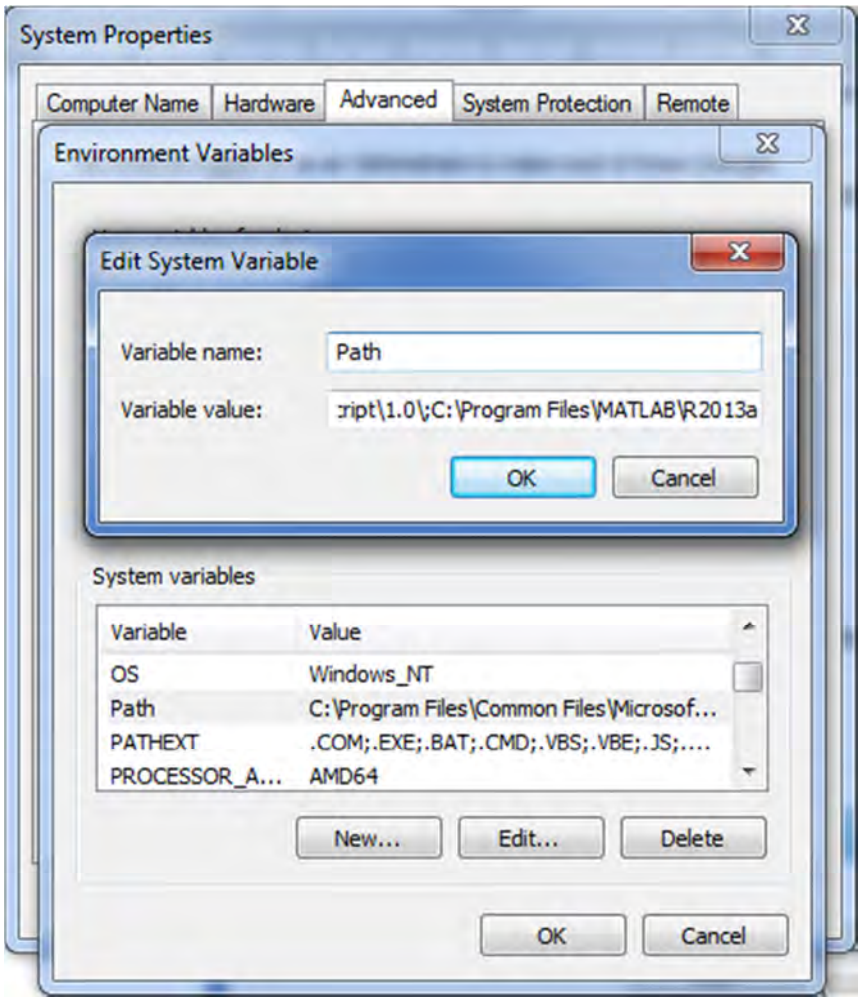


Figura XII.3. Menú de Paths en Windows.

Una vez añadido al final la ruta pulsamos OK en ambas ventanas y salimos del "Control panel"

Finalmente necesitamos registrar el Matlab Engine como servicio COM Server de Windows. Para ello cerramos Matlab y sobre el icono de Matlab pulsamos botón derecho y "Run as administrator". Se volverá a abrir Matlab y escribimos el comando:

```
cd(fullfile(matlabroot,'bin',computer('arch')))
!matlab /regserver
```

Al pulsar ENTER se nos abrirá una ventana de consola de Matlab que deberemos cerrar.

Una vez hecho esto el Matlab Engine ya estará correctamente configurado y listo para ser utilizado por nuestro proyecto.

4. Configuración de IDE para uso de Matlab Engine.

Una vez instalado el Visual Studio 2013 lo abrimos y creamos un nuevo proyecto: "File" -> "New Project"

En el asistente seleccionamos "Other Languages" -> "Visual C++" -> "Win32 Console Application". Le ponemos el nombre de "PruebaMatlabEngine" y pulsamos OK

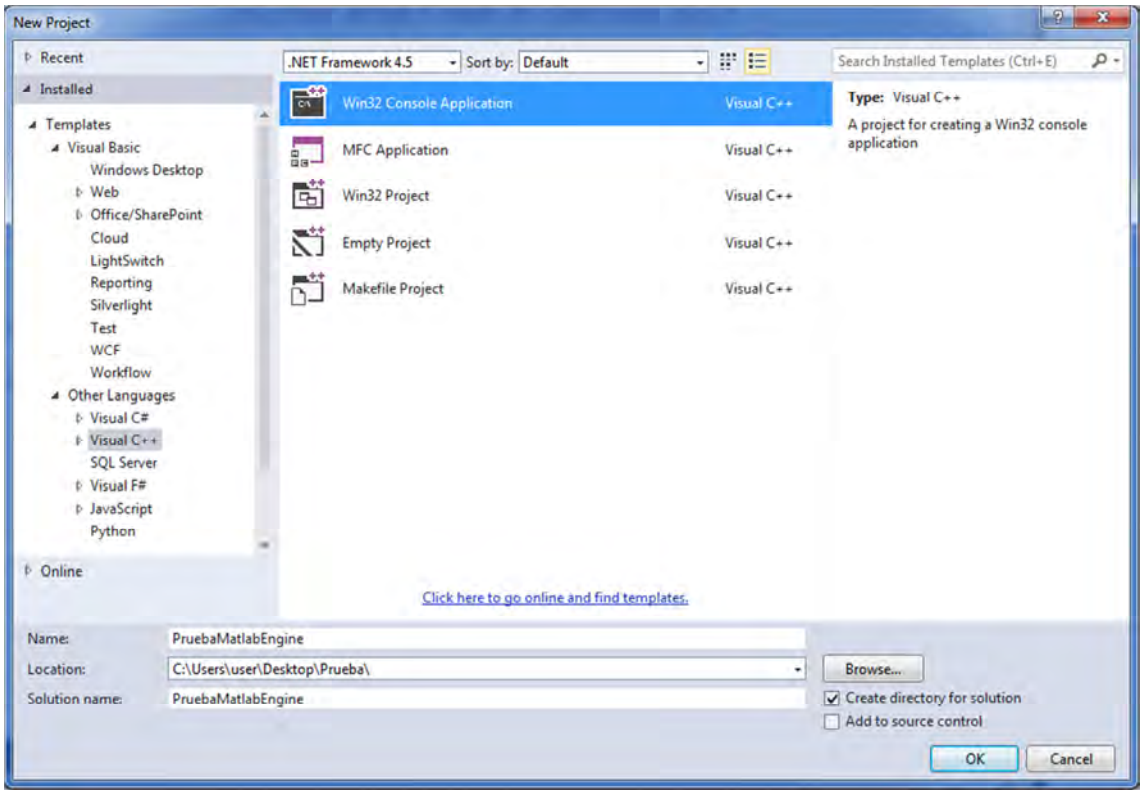


Figura XII.4. Menú de Proyecto nuevo en Visual Studio 2013

En el asistente "Win32 Application Wizard" pulsamos en "Finish".

Una vez cerrado el asistente deberemos primero cambiar la configuración de Bits de la aplicación a la que corresponda según el Matlab que hayamos instalado. Por defecto, las aplicaciones vienen para plataformas de 32 bits. Si

nuestro Matlab es de 64 bits necesitaremos cambiarlo. Para comprobar la versión que tiene instalada ha de abrirse Matlab y pulsar sobre "Help" -> "About Matlab".

En el entorno de Visual Studio accedemos a "Project" -> "PruebaMatLabEngine Properties". Pulsamos en la pestaña "Configuration properties" y a continuación arriba a la derecha en "Configuration Manager".

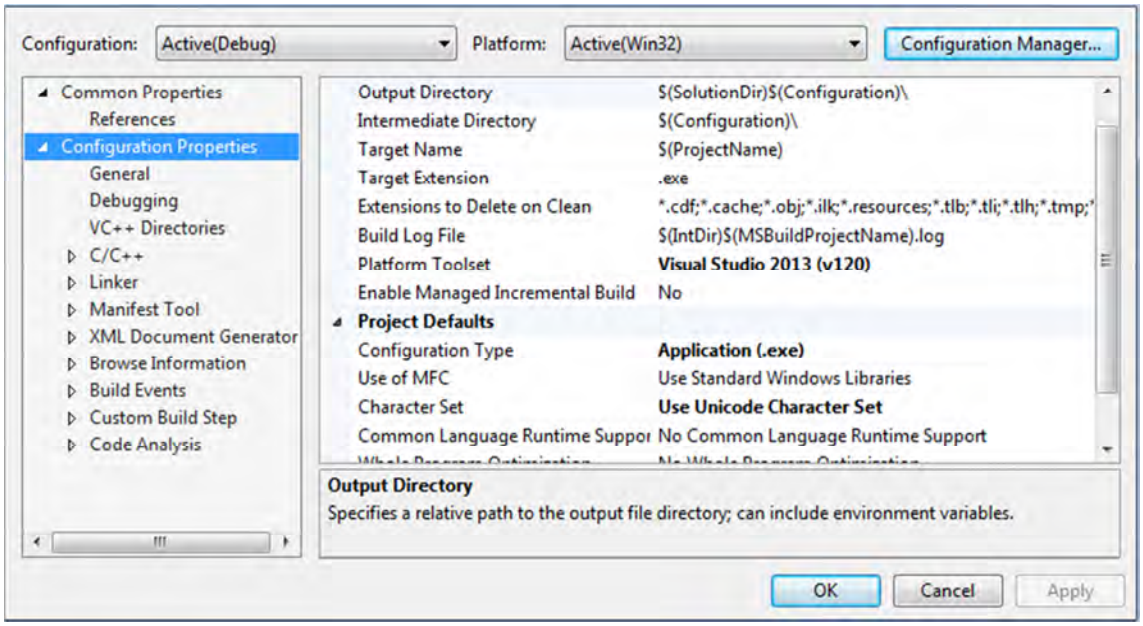


Figura XII.5. Menú de configuración de aplicación en Visual Studio 2013

En la pestaña "Active solution platform" pinchamos sobre "<New…>". En el desplegable de tipo de plataforma seleccionamos "x64". El resto lo dejamos por defecto. Pulsamos OK volviendo al menú anterior. Finalmente en el desplegable "Active solution platform" debemos dejar seleccionado "x64" que deberá coincidir con la opción "Platform" de nuestro proyecto más abajo. Pulsamos "Close" y volvemos al menú de configuración principal.

Una vez cambiada la plataforma de nuestra aplicación a 64bits deberemos incluir algunos directorios nativos de Matlab a nuestro proyecto para que importe las funciones. Para ello accedemos a "Configuration properties" -> "C/C++" -> "General" -> "Additional Include Directories" -> "<Edit…>". Pinchamos sobre el icono de la carpeta "New Line" y añadimos la dirección a la carpeta "`extern\include\`" de nuestra instalación de Matlab que en nuestro caso sería "`C:\Program Files\MATLAB\R2013a\extern\include`". Salimos dando a OK. El resultado debería ser lo que se muestra en la siguiente figura.

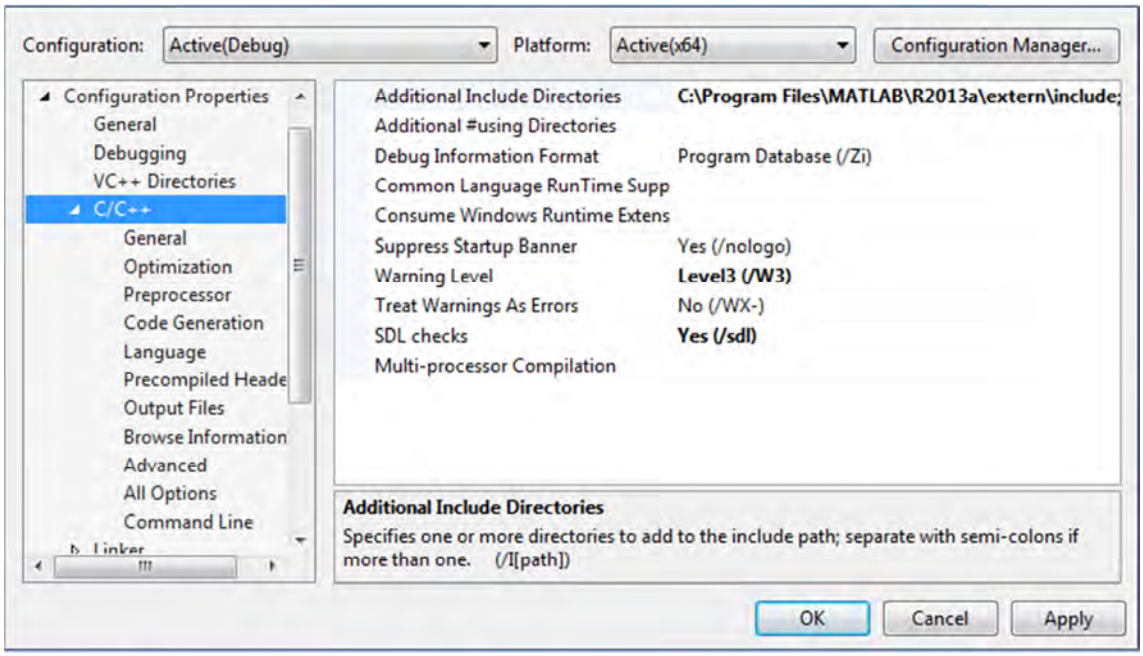


Figura XII.6. Menú de configuración de aplicación en Visual Studio 2013

Para continuar vamos a "Configuration properties" -> "Linker" -> "General" -> "Additional Library Directories" -> "<Edit…>". Pulsamos sobre el icono de la carpeta "New Line" y añadimos el directorio "`extern\lib\win64\microsof`" de nuestra carpeta de instalación de Matlab. En nuestro caso sería "`C:\Program Files\MATLAB\R2013a\extern\lib\win64\microsoft`". Pulsamos OK y volvemos al menú de configuración. Deberíamos tener la configuración como se muestra en la siguiente figura.

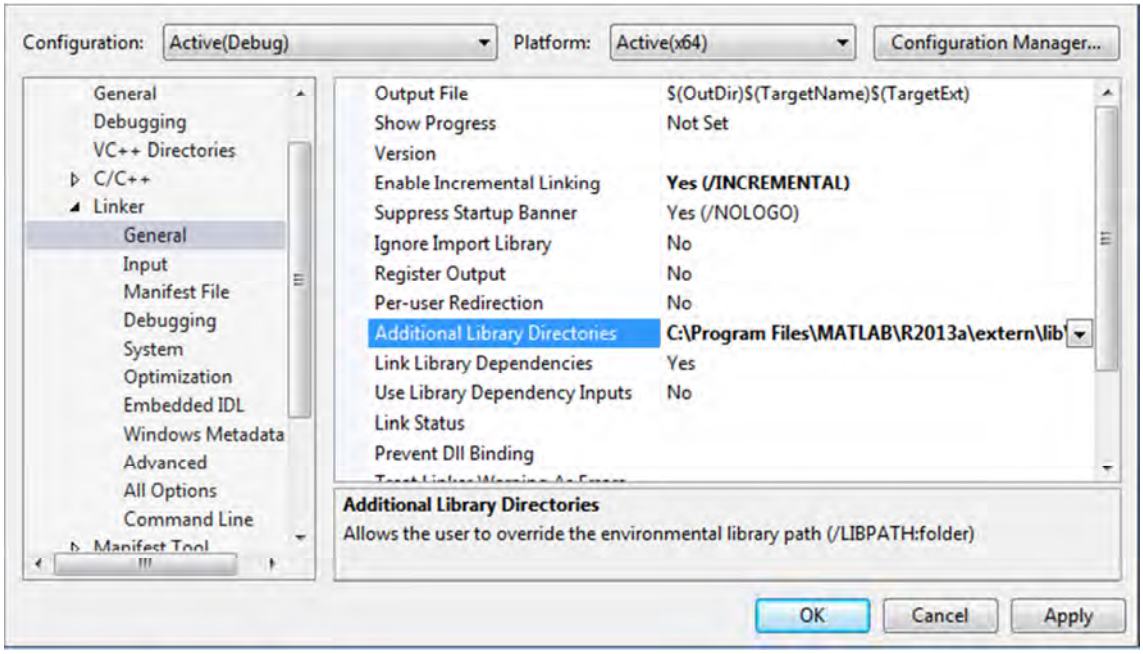


Figura XII.7. Menú de configuración de aplicación en Visual Studio 2013

Para finalizar vamos a "Configuration properties" -> "Linker" -> "Inputs" -> "Additional Dependencies" -> "<Edit…>" y escribimos "libeng.lib libmx.lib" como en la figura.

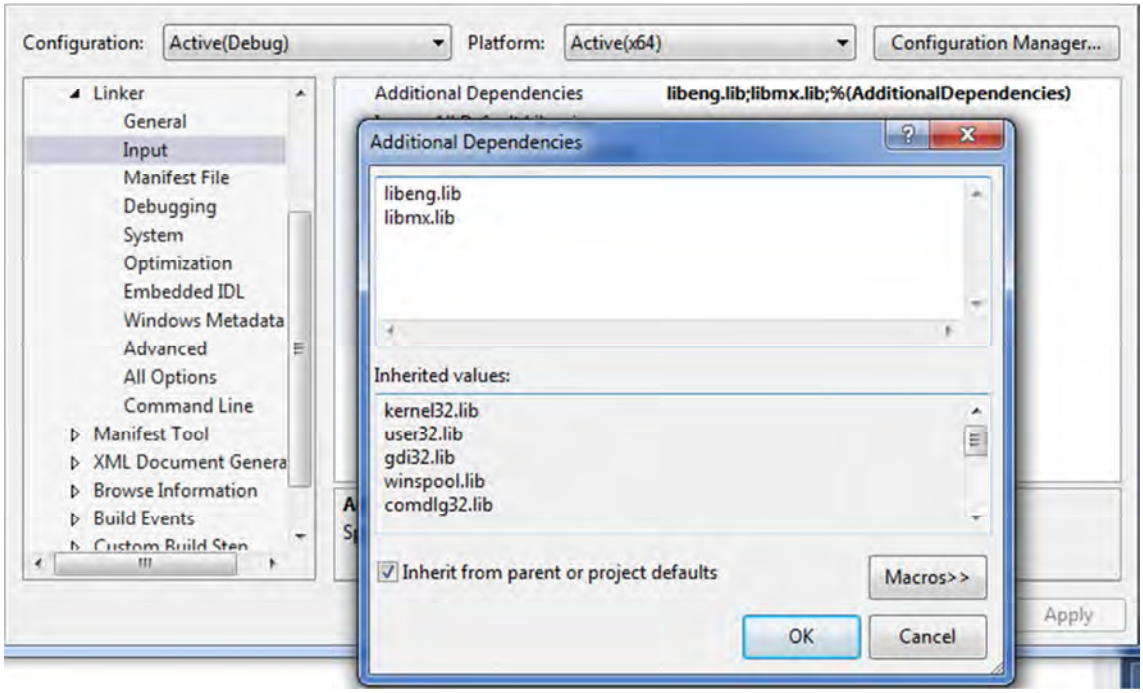


Figura XII.8. Menú de configuración de aplicación en Visual Studio 2013

Finalmente pulsamos OK dos veces y salimos de la configuración. Para terminar incluiremos en nuestro proyecto la librería "`engine.h`" para poder utilizar todos los métodos. Para ello escribimos en la parte superior:

```
#include "engine.h"
```

Al final el código del proyecto deberá presentar la siguiente apariencia.

```
// PruebaMatlabEngine.cpp


#include "stdafx.h"
#include "engine.h"

int _tmain(int argc, _TCHAR* argv[])
{
	return 0;
}
```

Un ejemplo de proyecto con los comandos más básicos sería el siguiente.

```
// PruebaMatlabEngine.cpp

#include "stdafx.h"
#include <windows.h>
#include <stdlib.h>
#include <stdio.h>
#include "engine.h"

#define BUFSIZE 256 //Tamaño de buffer

int _tmain(int argc, _TCHAR* argv[])
{
	Engine *mE;   //puntero Matlab Engine
	char buffer[BUFSIZE + 1];

	buffer[BUFSIZE] = '\0';    //iniciamos buffer

	printf("%s", "Iniciando Matlab Engine\n");
	if (!(mE = engOpen(""))) { //iniciamos conexion con Matlab Engine
		fprintf(stderr, "\nNo es posible iniciar MatlabEngine\n");
		return EXIT_FAILURE;
	}

	engEvalString(mE, "x=7");  //enviamos al MatlabEngine el comando
	engOutputBuffer(mE, buffer, BUFSIZE);   // le pedimos que guarde las
                                                // respuestas en la variable
                                                // buffer a partir de aqui
	engEvalString(mE, "y=x^2");//enviamos comando
	printf("%s", buffer);      //mostramos respuesta
	engClose(mE); //cerramos conexion con el Matlab Engine

	printf("Pulsa para terminar\n\n");
	fgetc(stdin);

	return EXIT_SUCCESS;
}
```

Al ejecutar este código obtendríamos la siguiente respuesta en la consola de Windows:

Figura XII.9. Resultado devuelto por la consola al ejecutar el comando

Estamos pues capacitados ya para ejecutar cualquier comando nativo de Matlab desde nuestra aplicación de C

5. Configuración de Google Earth para utilizarlo como simulador 3D

Para poder utilizar el complemento de visualización de la órbita en Google Earth es necesario realizar unos ajustes primeramente.

El primero paso consiste en descargar el programa desde la página web de Google. Una vez descargado entramos en el programa y abrimos el menú de Herramientas -> Opciones.

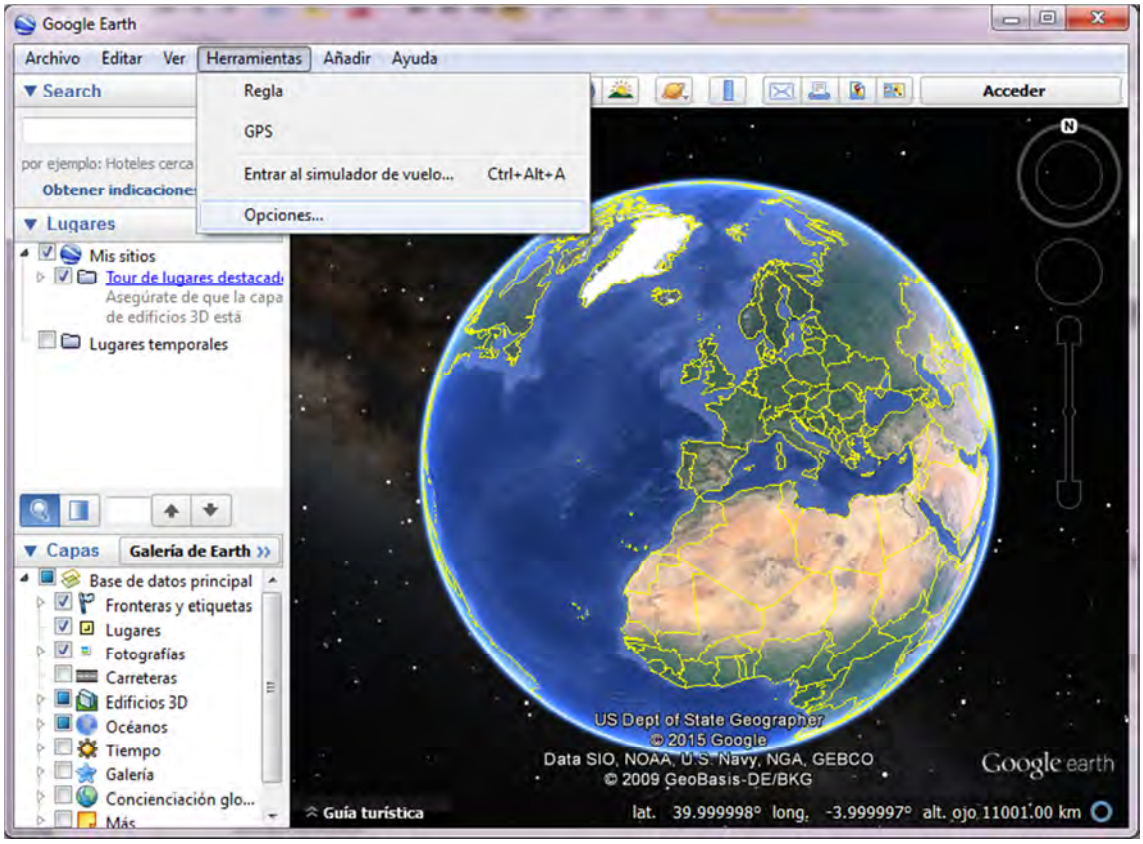


Figura XII.10. Vista previa de Google Earth

En el menú que nos aparece, en la pestaña de "Vista 3D" configuramos los siguientes parámetros. En "Modo de gráficos" debemos seleccionar "OpenGL". En el apartado de "Mostrar latitud/longitud" debemos de cambiar las medidas a grados decimales. Finalmente en el apartado de unidades de medida debemos seleccionar la opción de "Metros/Kilométros". Al final, nuestra configuración debe quedar de la siguiente forma:

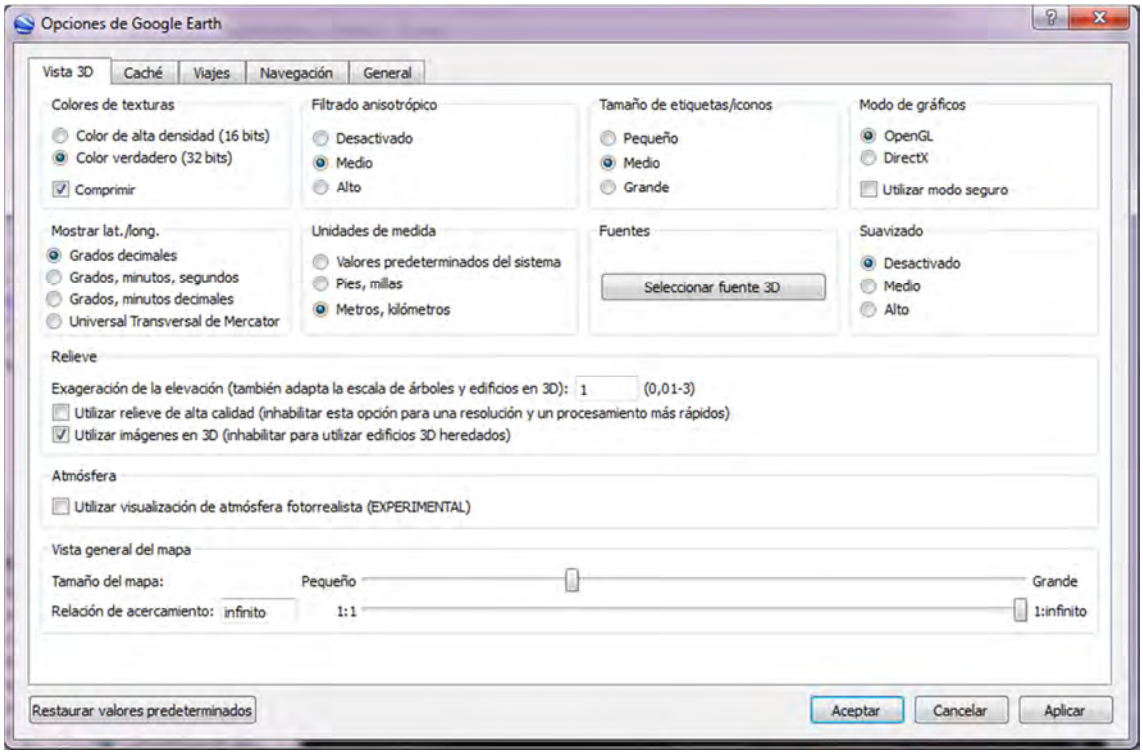


Figura XII.11. Menú de configuración de Google Earth

Para terminar pulsamos Aceptar y salimos del programa.