

E.T.S. de Ingeniería Industrial,
Informática y de Telecomunicación

Simulación de sistemas mecánicos holónomos
Validación de librería ``Dynamic_Library''



Máster Universitario en
Ingeniería Industrial

Complemento al Trabajo Fin de Máster

Vadim Coselev Condrea

Irene Miquelez Madariaga

Jorge Elso Torralba

Pamplona, 21/06/2024

upna

Universidad Pública de Navarra
Nafarroako Unibertsitate Publikoa

Este espacio se deja intencionalmente en blanco.

Índice general

1	Introducción	2
1.1	Justificación y objetivos	2
2	Métodos de resolución numérica	4
2.1	Sistemas mecánicos holónomos	4
2.2	Elección del método de resolución	5
2.3	Método Runge Kutta de orden 4 (RK4)	5
3	Modelo numérico de torre grúa	7
4	Caso práctico: Modelado numérico y simulación de torre grúa	9
4.1	Discretización del modelo	9
4.1.1	Separación en sólidos	10
4.1.2	Discretización de sólidos rígidos	10
4.2	Cálculo de parámetros de sólidos rígidos	12
4.2.1	Lectura de vértices y volúmenes	12
4.2.2	Definición de densidades	12
4.2.3	Cálculo de parámetros: masas y tensores de inercia	12
4.2.4	Características de la torre grúa	14
4.2.5	Comparativa de discretización con torre grúa Liebherr 550 EC-H 40	16
4.3	Construcción del modelo dinámico	17
4.3.1	Inicialización y configuración del sistema	17
4.3.2	Unión del sistema	18
4.4	Implementación RK4 en Matlab	19
4.4.1	Parámetros de función RK4 con <i>inputs</i>	19
4.5	Simulación	20
4.6	Animación en Blender	21
4.6.1	Matrices de transformación	21
4.6.2	Animación	22
5	Resultados de simulación	25
5.1	Movimiento pendular	25
5.2	Movimiento de carro	27
5.3	Giro de la pluma	28
5.4	Coste computacional	30
6	Conclusiones	31
7	Lineas futuras	32
8	Anexo	34
8.1	Clase “Body” Matlab	34
8.2	Script Matlab: Cálculo de tensores de inercia y masas	36
8.3	Script Matlab: Sistema dinámico de torre grúa y simulación	37
8.4	Función Matlab: Método RK4 con entradas	42
8.5	Script Python: Animación de la torre grúa en Blender	43

Índice de figuras

3.1	Modelo de torre grúa propuesto en [1].	7
3.2	Modelo de torre grúa propuesto en [2].	7
3.3	Diseño de pato con vóxels de diferente tamaño [3].	8
4.1	Modelo tridimensional de torre grúa. [4]	9
4.2	Modelo “.stl” de la pluma.	10
4.3	Modelo “.stl” del carro.	10
4.4	Vóxeles y volumen de sólidos rígidos que representan el sistema de la torre grúa. 4.1	11
4.5	Puntos y volumen del sólido “Base”.	12
4.6	Función “pcshow” sobre el sólido “Trolley”.	13
4.7	“pcshow” de sólidos discretizados de torre grúa.	14
4.8	Vista lateral de torre grúa [4].	15
4.9	Vista en planta de la torre grúa [4].	15
4.10	Liebherr 550 EC-H 40 [5]. Modelo de torre grúa <i>hammerhead</i> comercializada por la empresa Liebherr.	16
4.11	Datos de la torre del modelo Liebherr 550 EC-H 40 [5].	17
4.12	Contrapeso en el modelo tridimensional.	17
4.13	Datos del contrapeso en la torre grúa modelo Liebherr EC-H-40 [5].	17
4.14	Grados de libertad y acciones de entrada en modelo de torre grúa.	17
4.15	Componentes b_k^i y c_k^i de los sólidos “Jib”, “Trolley” y “Load” respecto a las coordenadas generalizadas θ_1, r, β_2 y β_1	18
4.16	Componentes $b_{k_v}^i$ y $c_{k_v}^i$ de los sólidos “Jib”, “Trolley” y “Load” respecto a las coordenadas generalizadas θ_1, r, β_2 y β_1	18
4.17	Logo de Blender	21
4.18	Python logo.	21
4.19	Ejemplos de matrices de transformación.	22
4.20	Archivos CSV para <i>Jib</i> , <i>Trolley</i> y <i>Load</i>	22
4.21	Directorio de simulación Blender.	23
5.1	Condiciones iniciales simulación torre grúa movimiento pendular.	25
5.2	Coordenadas generalizadas angulares ($\theta_1, \beta_2, \beta_1$) en simulación de movimiento pendular.	26
5.3	Secuencia de caída y subida de la carga	26
5.4	Coordenada generalizadas θ_1 en simulación de movimiento pendular.	27
5.5	Coordenadas generalizadas lineales (r) en simulación de movimiento pendular.	27
5.6	Coordenadas generalizadas lineales (r) en simulación de aceleración de la pluma.	28
5.7	Coordenadas generalizadas angulares ($\theta_1, \beta_2, \beta_1$) en simulación de aceleración de carro.	28
5.8	Secuencia de aceleración positiva carro.	28
5.9	Coordenadas generalizadas angulares ($\theta_1, \beta_2, \beta_1$) en simulación de aceleración de la pluma.	29
5.10	Coordenadas generalizadas lineales (r) en simulación de aceleración de la pluma.	29
5.11	Secuencia de giro pluma.	29
5.12	Comparativa en tiempo de cómputo de las simulaciones: movimiento pendular, F_{in} y M_{in}	30

Índice de Códigos

4.1	Script de discretización y cálculo de volumen archivos formato “.stl”	10
4.2	Volumen y coordenadas de los puntos discretizados del carro.	11
4.3	Archivo “convert.py” de la librería “stl_to_voxel[6]” original.	11
4.4	Archivo “convert.py” de la librería “stl_to_voxel[6]” modificado.	11
4.5	Lectura de posiciones de los vértices y volumen del sólido “Base”.	12
4.6	Lectura de archivos “.txt” en Matlab.	12
4.7	Función “pcshow” sobre el sólido “Trolley”.	13
4.8	Función “pcshow” sobre el sólido “Trolley”.	13
4.9	Código para implementar RK4 [7].	19
4.10	Funciones para el cálculo de $[A]$, $\dot{[A]}$ y Q .	19
4.11	Función para método RK4 método matricial.	20
4.12	Parámetros de configuración de la simulación.	20
4.13	Bucle de ejecución método RK4.	21
4.14	Bucle de ejecución método RK4 con exportación de matrices de transformación.	21
4.15	Matriz de transformación guardada en archivo CSV.	22
4.16	Código para importar STL a espacio de trabajo Blender.	22
4.17	Comando para cambiar el centro del sólido al cursor.	23
4.18	Importación de sólidos de torre grúa a Blender.	23
4.19	Importación matrices de transformación a través de archivos CSV a Blender.	24
4.20	Generación de animación en Blender.	24
8.1	Clase “Body” en Matlab.	34
8.2	Script para el cálculo de los tensores de inercia en CG de sólidos.	36
8.3	Script de configuración del sistema mecánico de torre grúa y simulación.	37
8.4	Función para método RK4 con entradas.	42
8.5	Programación completa para la importación y animación de la torre grúa.	43

Abstract

This supplement to the final project focuses on the simulation of holonomic mechanical models, aiming to model a construction tower crane and develop the necessary controllers. Various numerical resolution methods are explored, and the fourth-order Runge-Kutta method (RK4) is implemented in Matlab. To validate the *Dynamic Library*, a numerical model of the tower crane is constructed, resulting in a simulation and an animation in Blender. As a result, the library is validated, and an accurate mechanical model for the tower crane is obtained.

Keyword— RK4, discretization, voxel

Resumen

Este complemento a trabajo final se enfoca en la simulación de modelos mecánicos holónomos, con el objetivo de modelar una torre grúa de construcción y desarrollar los controladores necesarios. Se exploran diversos métodos de resolución numérica y se implementa el método de Runge-Kutta de cuarto orden (RK4) en Matlab. Para validar la librería *Dynamic Library*, se construye un modelo numérico de la torre grúa, del cual se obtiene una simulación y una animación en Blender. Como resultado, se valida la librería y se encuentra un modelo mecánico preciso para la torre grúa.

Palabras clave— RK4, discretización, voxel

1 Introducción

Este proyecto se enmarca dentro de una de las líneas de trabajo del grupo de investigación de Sistemas Dinámicos y Control, centrada en el problema de la torre grúa de construcción, con el objetivo de controlar su movimiento y reducir las cargas mecánicas experimentadas por sus componentes. En [8] se muestra la relación que existe entre el modelado físico de sistemas y el diseño de control y, cómo una buena metodología de modelado puede mejorar el desempeño en proyectos de diseño de controladores.

Otra de las partes fundamentales a la hora de diseñar controladores es realizar simulaciones para observar como reacciona el sistema ante un control propuesto. Esto ayuda a mejorar los diseños e incluso a disminuir costes debido a que las simulaciones, en gran parte de los casos, tienen un menor coste que las implementaciones reales. El objetivo principal de esta memoria es validar la librería *Dynamic_Library* [8] a través de la simulación de un modelo mecánico. Este desarrollo pretende servir de base para líneas futuras de investigación que estudien controladores para modelos mecánicos holónomos y, en concreto, para grúas de construcción.

1.1 Justificación y objetivos

En esta sección se hará una justificación de los problemas que existen actualmente en el modelado de sistemas mecánicos, en relación a diseño de controladores y, los objetivos que este proyecto intenta cumplir.

Justificación

En [8] se detalla cómo la librería *Dynamic_Library* puede obtener las ecuaciones diferenciales de movimiento (EDM) simbólicas para sistemas de sólidos rígidos empleando un péndulo como ejemplo. No obstante, en sistemas complejos, las soluciones simbólicas se vuelven más complicadas de obtener. Como alternativa, estas ecuaciones pueden ser calculadas numéricamente, una funcionalidad que la librería ofrece a través de la función *unify_system*.

Para integrar la librería en entornos de la ingeniería, es crucial validar su capacidad para obtener EDM numéricas de manera precisa. Esta validación se llevará a cabo mediante la inspección de simulaciones. Aunque existen métodos técnicos, como el cálculo de balances de energía y el análisis de gráficas temporales, se ha optado por ilustrar el movimiento de una torre grúa mediante una animación. La representación de los sistemas a través de animaciones tiene la ventaja de que se pueden comparar con nuestras intuiciones sobre la física del movimiento de los objetos.

Dado que el modelo se presenta numéricamente, la resolución de sus EDM también se realiza del mismo modo. La variedad de técnicas disponibles para este propósito implica la necesidad de seleccionar y aplicar la más adecuada para este caso específico.

En el contexto de este estudio, resulta interesante validar la librería utilizando un modelo de torre grúa. Además de avanzar en el objetivo principal de esta investigación, esto permite una comparación directa con el trabajo previo inmediato [1].

Objetivos

Para tener en mente la finalidad de este trabajo, las justificaciones de la Sección 1.1 se van a plasmar en una serie de objetivos que se intentarán cumplir.

- Proponer métodos de resolución numérica de ecuaciones diferenciales e implementar uno para la simulación del sistema.

- Construir un modelo mecánico de una torre grúa. Este modelo debe ser numérico para implementar métodos de resolución numérica.
- Realizar la simulación del modelo contrastando el resultado a través de una animación.

2 Métodos de resolución numérica

La simulación de sistemas mecánicos es un tema amplio que integra muchas ramas de la mecánica y las matemáticas. Dependiendo de la precisión que se requiera del modelo, existirán aproximaciones que se pueda hacer y otras que no. A continuación se describen los métodos de resolución de EDM de mayor relevancia y se discuten las ventajas del seleccionado.

2.1 Sistemas mecánicos holónomos

En el ejemplo clásico de un péndulo simple en el que las EDM vienen dadas por la Ecuación

$$\ddot{\theta} + \frac{g}{L} \sin(\theta) = 0, \quad (2.1)$$

si se consideran pequeñas oscilaciones, $\sin(\theta) \approx \theta$, la Ecuación 2.1 se aproxima a

$$\ddot{\theta} + \frac{g}{L} \theta = 0. \quad (2.2)$$

Hoy en día, existen métodos de modelado que varían desde el tratamiento de sólidos rígidos hasta modelos más flexibles, así como técnicas combinadas para lograr precisión computacional eficiente. Dado que la librería *Dynamic_Library* está diseñada para modelos bajo la suposición de sólidos rígidos y holónomos, en esta sección se explorarán diversas formas de resolver sistemas de ecuaciones del tipo

$$[\mathbf{A}] \ddot{\mathbf{q}} = [\mathbf{b}], \quad (2.3)$$

que definen los sistemas que puede estudiar la librería en cuestión.

Métodos Analíticos

Los métodos analíticos proporcionan soluciones exactas para ecuaciones diferenciales, aunque sus aplicaciones están limitadas a problemas simples y bien definidos.

- Los métodos de integración directa incluyen técnicas como la separación de variables, series de Fourier y transformadas de Laplace, adecuados para ecuaciones diferenciales lineales y algunas no lineales simples.
- Los métodos perturbativos se utilizan en sistemas con no linealidades ligeras, donde la solución se approxima mediante un desarrollo en serie alrededor de un punto.

Métodos Numéricos

Para sistemas complejos donde las soluciones analíticas no son viables, se emplean métodos numéricos para obtener soluciones aproximadas.

- Para ecuaciones diferenciales ordinarias (ODEs), el método de Euler y la familia de métodos Runge-Kutta son muy utilizados. Aunque el método de Euler es simple, tiene limitaciones en precisión; los métodos Runge-Kutta, en cambio, son más precisos y convergen más rápidamente al resultado [9].
- El método de los elementos finitos (MEF) es crucial en problemas estructurales complejos. Aquí, el dominio se divide en elementos finitos y se resuelven las EDM para cada elemento.

2.2 Elección del método de resolución

Debido a que los modelos a simular son numéricos (uno de los objetivos del proyecto), el método de resolución también debe serlo. La elección del método numérico concreto se basa en criterios diferentes como el tamaño del sistema, la presencia de fuerzas externas o el grado de precisión requerido.

En el conjunto de sistemas que representa la Ecuación 2.3, la matriz $[A]$ es invertible y, por lo tanto, pueden transformarse en sistemas de ecuaciones diferenciales ordinarias (ODEs) de segundo orden como muestra la Ecuación

$$\ddot{\mathbf{q}} = [A]^{-1} [\mathbf{b}]. \quad (2.4)$$

De hecho, la Ecuación 2.4 puede reducirse a sistemas ODEs de primer orden mediante el cambio de variables $(\mathbf{q}, \dot{\mathbf{q}}, \ddot{\mathbf{q}}) \rightarrow (\mathbf{u}, \mathbf{v}, \dot{\mathbf{v}})$. Este sistema final se muestra en la Ecuación

$$\begin{bmatrix} \dot{\mathbf{v}} \\ \dot{\mathbf{u}} \end{bmatrix} = \begin{bmatrix} [A(\mathbf{u}, \mathbf{v})]^{-1} [\mathbf{b}] \\ \mathbf{v} \end{bmatrix}. \quad (2.5)$$

La Ecuación 4.3 revela que el método de Euler y la familia de métodos Runge-Kutta son las técnicas adecuadas para la resolución de estos problemas, dado que se requiere un método numérico para un sistema de ODEs.

Para la implementación práctica, Matlab recomienda comenzar con su función “ode45”, que ejecuta un método Runge-Kutta de cuarto orden (RK4). Stoneking en [10] propone un ejemplo de algoritmo de resolución que utiliza un método RK4 y aplica a sistemas mecánicos, especialmente en el ámbito aeroespacial. Por lo tanto, parece razonable utilizar un método RK4 para la resolución numérica del sistema de torre grúa.

La elección del método puede estudiarse en mayor detalle. Existen métodos numéricos para resolver ecuaciones diferenciales de segundo orden, lo que hace innecesaria la reducción de orden, así como métodos para ecuaciones diferenciales algebraicas (DAE), que no requieren la inversión de matrices. Es crucial controlar los errores numéricos y la estabilidad para obtener buenos resultados en la simulación. Dado que el objetivo de este trabajo es validar la librería con un caso práctico, se ha elegido un método bien documentado y de fácil implementación. Los métodos RK pueden realizarse con pasos fijos o variables, y se ha optado por un método de paso fijo por su sencillez de programación.

2.3 Método Runge Kutta de orden 4 (RK4)

La presentación habitual de los métodos Runge-Kutta se realiza para ODEs definidas de la forma

$$x'(t) = f(t, x(t)). \quad (2.6)$$

Si bien esta forma es totalmente correcta, para el caso que atañe este proyecto, es necesaria la introducción de entradas al sistema. En el caso de los sistemas estudiados, los estados futuros no dependen únicamente de los estados actuales, sino también de las entradas al sistema. Por esto mismo, se presentan las relaciones como

$$x'(t) = f(x(t), u(t))[11]. \quad (2.7)$$

Teniendo las entradas en cuenta, el método Runge-Kutta de orden “s” viene presentado por

$$x_{n+1} = x_n + h \sum_{i=1}^s b_i g_i, \quad (2.8)$$

y los coeficientes que componen el método se pueden calcular como

$$g_i = f \left(x_n + h \sum_{j=1}^s a_{ij} g_j, u(nh + c_i h) \right). \quad (2.9)$$

Habitualmente los métodos RK se caracterizan con una colección de vectores y matrices que se representan en la “Tabla de Butcher”. La Ecuación

$$\frac{\mathbf{c} \mid [\mathbf{A}]}{\mathbf{b}} = \begin{array}{c|ccccc} c_1 & a_{11} & \dots & a_{1s} \\ \vdots & \vdots & & \vdots \\ c_s & a_{s1} & \dots & a_{ss} \\ \hline b_1 & \dots & b_s \end{array}. \quad (2.10)$$

presenta una “Tabla de Butcher” genérica y, para el método RK4, se puede particularizar como

$$\begin{array}{c|cccc} 0 & & & & \\ 1/2 & 1/2 & & & \\ 1/2 & 0 & 1/2 & & [9]. \\ 1 & 0 & 0 & 1 & \\ \hline & 1/6 & 1/3 & 1/3 & 1/6 \end{array} \quad (2.11)$$

Los coeficientes que se extraen de la tabla de la Ecuación 2.11 pueden calcularse como muestra la Ecuación

$$\begin{aligned} g_1 &= f(x_n, u(nh + c_1 h)), \\ g_2 &= f\left(x_n + \frac{1}{2}hg_1, u(nh + c_2 h)\right), \\ g_3 &= f\left(x_n + \frac{1}{2}hg_2, u(nh + c_3 h)\right), \\ g_4 &= f(x_n + hg_3, u(nh + c_4 h)). \end{aligned} \quad (2.12)$$

El principal inconveniente de este método es que requiere conocer el valor de la entrada en $nh + c_1 h$, $nh + c_2 h$, $nh + c_3 h$ y $nh + c_4 h$ para su calculo. Es decir, son necesarios datos futuros de la entrada para poder realizar la resolución numérica. En caso de que el sistema integre un controlador del que se pueda conocer la salida en función del estado del sistema, esto es posible, pero realizar una simulación con una entrada arbitraria no lo es. La aproximación que se realiza habitualmente es suponer que el paso h es lo suficientemente pequeño para que la entrada u no varíe mucho en los instantes de cálculo. De esta forma los coeficientes del método se pueden aproximar a

$$\begin{aligned} g_1 &= f(x_n, u_n), \\ g_2 &= f\left(x_n + \frac{1}{2}hg_1, u_n\right), \\ g_3 &= f\left(x_n + \frac{1}{2}hg_2, u_n\right), \\ g_4 &= f(x_n + hg_3, u_n). \end{aligned} \quad (2.13)$$

3 Modelo numérico de torre grúa

Para realizar la simulación se debe construir un modelo mecánico con el cual contrastar la librería y, debido a que el contexto es el diseño de controladores para una torre grúa, es una buena idea construir un modelo de la misma. En desarrollos previos [1], se habían propuesto las EDM de una torre grúa con ecuaciones simbólicas. Si bien estas ecuaciones pueden ser utilizadas para obtener el modelo numérico, esta torre grúa se compone únicamente de tres grados de libertad (gdl) como muestra la Figura 3.1.

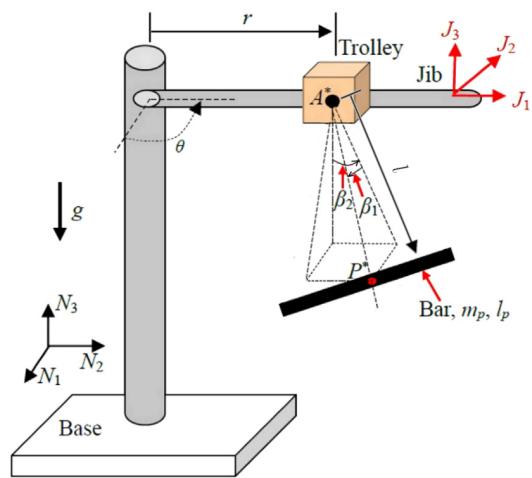
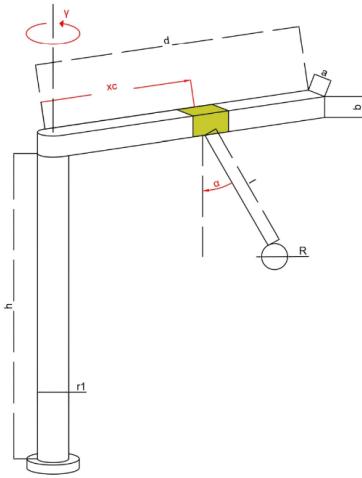


Figura 3.1: Modelo de torre grúa propuesto en [1]. Figura 3.2: Modelo de torre grúa propuesto en [2].

Los gdl del modelo se pueden ampliar con el modelo de la Figura 3.2. Este sistema, ya estudiado a través de la librería *Dynamic_Library* en [8], tiene cinco gdl que permiten a la barra (Bar) girar en el espacio tridimensional completo.

Ambos modelos presentados, tienen la particularidad de que los sólidos que los componen son figuras geométricas simples. Para este tipo de figuras, las propiedades como, por ejemplo, el tensor de inercia \mathcal{I} , se encuentran tabuladas en la bibliografía, lo que facilita una primera aproximación del modelo o el cálculo simbólico de las EDM. El inconveniente de este tipo de modelado es que para cada sistema mecánico se debe definir de forma particular las figuras simples por las que se va a aproximar. Cuanto más parecida sea la geometría que se han modelado al caso real, mayor será el acercamiento a la realidad. Lo contrario es igual de valido, cuanto más simple sea la forma, menos características del sistema real representa ese sólido.

Una alternativa al cálculo manual de parámetros de los sólidos es el cálculo numérico de estas magnitudes a través de la discretización de diseños tridimensionales. La Figura 3.3, muestra dos modelos 3D de un pato. Cada uno de estos modelos se ha formado a partir de unidades más pequeñas denominadas voxel (del inglés *volumetric pixel*). Cada voxel del diseño, tiene un volumen asociado y, cuanto más pequeño sea el voxel, más se ajusta el modelo discretizado al continuo. Volviendo a la Figura 3.3, el pato de la izquierda se ha discretizado con un tamaño de voxel mas grande que el de la derecha, por lo tanto, el modelo discretizado izquierdo será menos preciso que el derecho.

El cálculo de parámetros mecánicos de los sólidos, también se puede discretizar. La Ecuación

$$m_{Sol} = \rho V_{Sol}, \quad (3.1)$$

presenta la masa de un sólido continuo con una distribución de masa uniforme. Si este sólido se divide en unidades más pequeñas, la ecuación discreta toma la siguiente forma

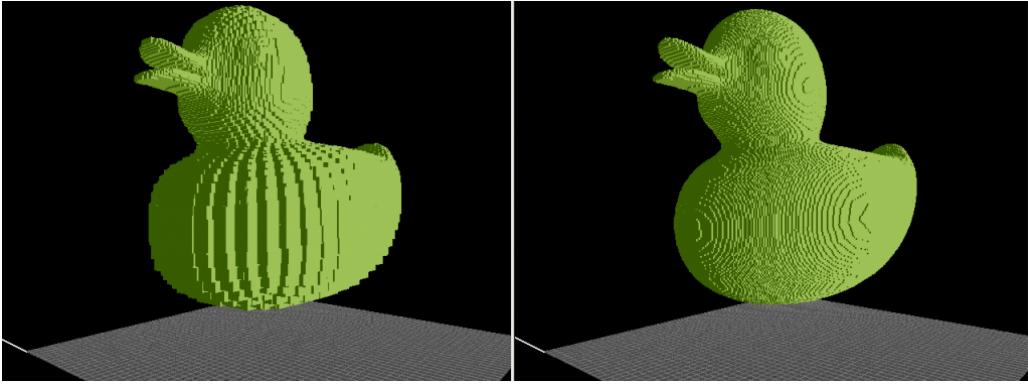


Figura 3.3: Diseño de pato con véxels de diferente tamaño [3].

$$m_{Sol} = \sum_{k=1}^N \rho V_i. \quad (3.2)$$

Con el cálculo del tensor de inercia del sólido ocurre algo similar. La Ecuación

$$\mathcal{I} = \begin{bmatrix} I_{11} & I_{12} & I_{13} \\ I_{21} & I_{22} & I_{23} \\ I_{31} & I_{32} & I_{33} \end{bmatrix} = \begin{bmatrix} \int_S (y^2 + z^2) dm & -\int_S xydm & -\int_S xzdm \\ -\int_S xydm & \int_S (x^2 + z^2) dm & -\int_S yzdm \\ -\int_S xzdm & -\int_S yzdm & \int_S (x^2 + y^2) dm \end{bmatrix}, \quad (3.3)$$

muestra el cálculo del tensor de inercia de un sólido continuo. Si se realiza la discretización del sólido, el tensor de inercia se calcula como

$$\mathcal{I} = \begin{bmatrix} I_{11} & I_{12} & I_{13} \\ I_{21} & I_{22} & I_{23} \\ I_{31} & I_{32} & I_{33} \end{bmatrix} = \begin{bmatrix} \sum_{k=1}^N m_k (y_k^2 + z_k^2) & -\sum_{k=1}^N m_k x_k y_k & -\sum_{k=1}^N m_k x_k z_k \\ -\sum_{k=1}^N m_k x_k y_k & \sum_{k=1}^N m_k (x_k^2 + z_k^2) & -\sum_{k=1}^N m_k y_k z_k \\ -\sum_{k=1}^N m_k x_k z_k & -\sum_{k=1}^N m_k y_k z_k & \sum_{k=1}^N m_k (x_k^2 + y_k^2) \end{bmatrix}. \quad (3.4)$$

4 Caso práctico: Modelado numérico y simulación de torre grúa

Para obtener un modelo numérico de la torre grúa y simularlo utilizando el método RK4, primero se debe conseguir un diseño 3D y separarlo en un conjunto de sólidos rígidos. El modelo 3D que se va a utilizar se ha obtenido del portal *Sketchfab* [4] y, la separación en sólidos rígidos, se va a realizar con ayuda del software libre *Blender*. Posteriormente, se debe calcular numéricamente los parámetros de los sólidos que componen la torre grúa. Para este fin, se hace uso de las librerías de Python *stl-to-voxel* [6] y *STL Volume Model Calculator* [12] para discretizar los sólidos rígidos. Un *script* en Matlab realiza el cálculo de las masas y los tensores de inercia a partir de las discretizaciones.

Para el entorno de simulación se hará uso de las funciones generadas por la librería *Dynamic_Library* de las cuales se obtienen las componentes \mathbf{b}_i y \mathbf{c}_i (parciales de las velocidades [8]). Estas funciones de Matlab se incluyen dentro de un bucle que realiza el método de resolución numérica Runge-Kutta de orden 4.

Para llevar a cabo la animación es necesario el cálculo de matrices de transformación 4x4. El propio bucle que realiza la resolución numérica calcula estas matrices para cada instante de tiempo. Finalmente, las matrices de transformación se importan en Blender para conseguir la animación final.

4.1 Discretización del modelo

Para la torre grúa se ha obtenido el modelo 3D que se muestra en la Figura 4.1[4]. Este modelo se va a separar en diferentes sólidos, los cuales definirán el sistema de sólidos rígidos a discretizar.



Figura 4.1: Modelo tridimensional de torre grúa. [4]

Los sólidos rígidos se van a aproximar a través de vértices, con la misma densidad y volumen para cada uno de ellos. De esta forma, se puede obtener numéricamente las masas y los tensores de inercia.

4.1.1 Separación en sólidos

Un formato de modelos 3D muy extendido es el STL (*Standard Tessellation Language*), debido a que las librerías de Python que se van a utilizar son para este formato, se deberá convertir el formato original a STL. El 3D originalmente se codifica en FBX, a través de Blender se ha conseguido separar cada uno de los sólidos rígidos y extraerlos como archivos STL. En las Figuras 4.2 y 4.3 se muestran los modelos de la pluma y del carro de la grúa.

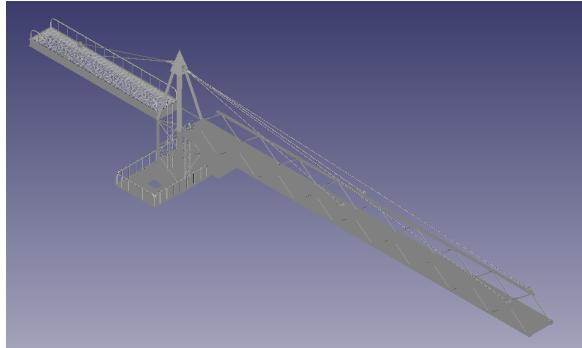


Figura 4.2: Modelo “.stl” de la pluma.

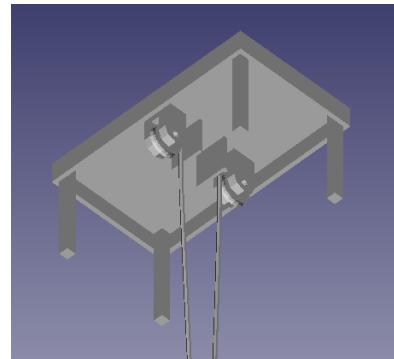


Figura 4.3: Modelo “.stl” del carro.

En total, la torre grúa se ha separado en 5 sólidos rígidos:

- Torre
- Pluma
- Bloque de Mortero
- Carro
- Carga

4.1.2 Discretización de sólidos rígidos

Cada uno de los sólidos se debe discretizar en véxoles. Para ello se hace uso de la librería *stl_to_voxel*[6] en Python, la cual permite realizar la discretización. Debido a que es necesario el volumen para el cálculo de la masa, también se hará uso de la librería *volume_calculator*[12] que calcula el volumen de modelos cerrados en formato STL. El Código 4.1, realiza para cada uno de los sólidos la discretización y el cálculo de volumen.

```
1 import stltovoxel
2 import volume_calculator
3
4 body_names = ["Base", "Jib_Concrete", "Jib_Steel", "Load", "Trolley"]
5 resolutions = [500, 100, 100, 100, 100]
6 for body_name, resolution_body in zip(body_names, resolutions):
7     stl_name = body_name + ".stl"
8     directory_stl = "Tower_Crane_Model/Tower_crane/Source/stl/"
9     stltovoxel.convert_file(directory_stl + stl_name, "txt/" + body_name + ".txt",
10     resolution=resolution_body)
11     Volume_Calculator = volume_calculator.STLUtils()
12     Volume_Calculator.loadSTL(directory_stl + stl_name)
13     stl_Volume = Volume_Calculator.calculate_Only_Volume()
14     with open("txt/" + body_name + "_Volume.txt", 'w') as file:
15         file.write(str(stl_Volume))
```

Código 4.1: Script de discretización y cálculo de volumen archivos formato “.stl”

El resultado de este script son los archivos mostrados en la Figura 4.4.

Base.txt	12/04/2024 10:26	Documento de tex...	1.160 KB
Base_Volume.txt	12/04/2024 10:26	Documento de tex...	1 KB
Jib_Concrete.txt	12/04/2024 10:26	Documento de tex...	16.954 KB
Jib_Concrete_Volume.txt	12/04/2024 10:26	Documento de tex...	1 KB
Jib_Steel.txt	12/04/2024 10:26	Documento de tex...	771 KB
Jib_Steel_Volume.txt	12/04/2024 10:26	Documento de tex...	1 KB
Load.txt	12/04/2024 10:26	Documento de tex...	1.860 KB
Load_Volume.txt	12/04/2024 10:26	Documento de tex...	1 KB
Trolley.txt	12/04/2024 10:26	Documento de tex...	298 KB
Trolley_Volume.txt	12/04/2024 10:26	Documento de tex...	1 KB

Figura 4.4: Vóxeles y volumen de sólidos rígidos que representan el sistema de la torre grúa. 4.1

Los archivos *_Volume.txt* contienen el volumen del sólido en cuestión en las medidas del archivo STL. En este caso el modelo original se ha diseñado en metros, por lo tanto, el volumen de los sólidos se obtiene en metros cúbicos. El resto de archivos contienen las posiciones cartesianas de cada uno de los vóxeles creados para cada uno de los sólidos. En el Código 4.2 se muestra el volumen y las coordenadas de algunos vóxeles del sólido referente al carro.

```

1 % TROLLEY VOLUME
2 0.33837368835953785
3 % TROLLEY POINTS
4 29.353337825261757 1.360742861012229 39.80576050858855
5 29.567422143665468 1.360742861012229 39.80576050858855
6 29.353337825261757 1.4030120807455293 39.80576050858855
7 ...
8 30.124041371515112 2.586550233277939 43.9166145079808
9 30.166858235195853 2.586550233277939 43.9166145079808

```

Código 4.2: Volumen y coordenadas de los puntos discretizados del carro.

Si bien se ha mostrado que la librería *stl_to_voxel*[6] ha realizado la exportación de datos a un archivo TXT, la librería original no tiene esta característica. Los formatos originales a los que se exporta el conjunto de vóxeles son XYZ, SVX o NPY. No ha sido posible importar ninguno de estos formatos a Matlab, por lo tanto, se ha introducido en la librería la capacidad de exportar en TXT la posición de los vóxeles. El Código 4.3 muestra la librería original con la cual no se tiene la capacidad de exportar en TXT, mientras que el Código 4.4 muestra la librería modificada para cubrir esta necesidad.

```

1 if output_file_extension == '.png':
2     vol = np.pad(vol, pad)
3     export_pngs(vol,
4     output_file_path, colors)
5     elif output_file_extension == '.xyz':
6         export_xyz(vol,
7         output_file_path, scale, shift)
8     elif output_file_extension == '.svx':
9         export_svx(vol,
10        output_file_path, scale, shift)
11     elif output_file_extension == '.npy':
12         export_npy(vol,
13         output_file_path, scale, shift)

```

Código 4.3: Archivo “convert.py” de la librería “*stl_to_voxel*[6]” original.

```

1 if output_file_extension == '.png':
2     vol = np.pad(vol, pad)
3     export_pngs(vol,
4     output_file_path, colors)
5     elif output_file_extension == '.xyz':
6         export_xyz(vol,
7         output_file_path, scale, shift)
8     elif output_file_extension == '.svx':
9         export_svx(vol,
10        output_file_path, scale, shift)
11     elif output_file_extension == '.npy':
12         export_npy(vol,
13         output_file_path, scale, shift)
14     elif output_file_extension == '.txt':
15         export_xyz(vol,
16         output_file_path, scale, shift)

```

Código 4.4: Archivo “convert.py” de la librería “*stl_to_voxel*[6]” modificado.

4.2 Cálculo de parámetros de sólidos rígidos

Finalizada la discretización de la torre grúa, se deben calcular los parámetros numéricos de cada uno de los sólidos rígidos en los que se ha dividido.

4.2.1 Lectura de véxeles y volúmenes

El cálculo de las masas y los tensores de inercia se realiza a través de Matlab. El Código 4.6 muestra la función *readmatrix* la cual lee los archivos TXT generados en Python y los convierte en arrays en Matlab. En la Figura 4.5, se observa el array de véxeles y, el volumen, leidos para el sólido “base”. Este sólido en cuestión se ha discretizado para 20443 véxeles, cada uno de los cuales se localizan por las 3 coordenadas cartesianas.

```
1 Base_Points = readmatrix("txt\Base.txt");
2 Base_Volume = readmatrix("txt\Base_Volume.txt");
```

Código 4.5: Lectura de posiciones de los véxeles y volumen del sólido “Base”.

	Base_Points	20443x3 double
	Base_Volume	9.6977

Figura 4.5: Puntos y volumen del sólido “Base”.

4.2.2 Definición de densidades

En el modelo, solo se considerarán dos materiales: el acero y el mortero de cemento. Esto se debe a que el objetivo no es obtener una representación altamente realista de la torre grúa, sino validar la librería dinámica. No obstante, cabe destacar que este método permite incluir tantos materiales como se desee. La densidad de cada uno de los materiales es:

- acero: $\rho = 7,850 \text{ kg/m}^3$ y
- mortero de cemento: $\rho = 2,400 \text{ kg/m}^3$.

4.2.3 Cálculo de parámetros: masas y tensores de inercia

Para facilitar generalizar el procedimiento de cálculo de los tensores de inercia, se ha programado la clase *Body* en Matlab, la cual, se inicializa con los parámetros:

- nombre,
- puntos,
- volumen y
- densidad.

Esta clase ayuda a realizar el cálculo de los tensores de inercia en el centro de gravedad del sólido, requisito de la librería dinámica. La idea del cálculo de parámetros se va a explicar a través del carro de la torre grúa. Para este sólido se tienen que importar los TXT de las coordenadas de los véxeles y del volumen del sólido, la función *radmatrix* se encarga de importar estos archivos como arrays a Matlab. Posteriormente se debe definir la densidad, parámetro que se inicializa de forma manual. Finalmente, se crea el objeto “Base” de la clase *Body*. Este procedimiento se expone en el Código 4.6.

```
1 Trolley_Points = readmatrix("txt\Trolley.txt");
2 Trolley_Volume = readmatrix("txt\Trolley_Volume.txt");
3 Steel_Density = 7850; %Kg/m3
4 Trolley_Body = body("Trolley",Trolley_Points,Trolley_Volume,Steel_Density);
```

Código 4.6: Lectura de archivos “.txt” en Matlab.

Una de las funciones que se ha implementado en la clase es el uso de la función `pcshow` de Matlab para poder visualizar la localización de los véxeles de los que se compone el sólido. El Código 4.8 muestra el comando para ejecutar la función y, la Figura 4.6, es la ventana en la que se observa la nube de puntos que compone el sólido. Si se quiere una visualización más precisa, en la que se muestren los véxeles, se debería calcular el tamaño de cada uno de los cubos, y representarlo en la posición que le corresponda.

```
1 Trolley_Body.pcshow_body;
```

Código 4.7: Función “`pcshow`” sobre el sólido “Trolley”.

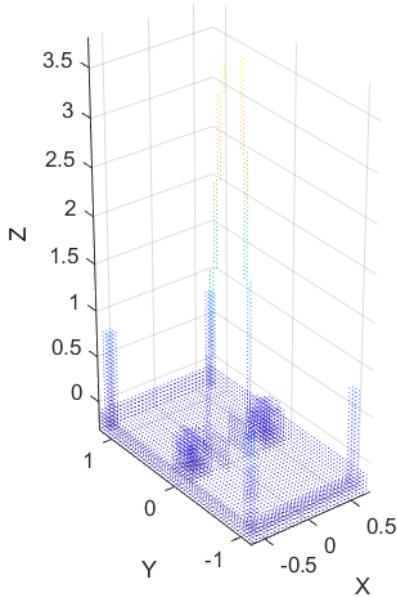


Figura 4.6: Función “`pcshow`” sobre el sólido “Trolley”.

Se han introducido las propiedades de localización centro de masas, tensor de inercia y masa en la clase `Body`, por lo tanto, el usuario puede acceder a estos valores. El Código 4.8 muestra como acceder a estos valores para los sólido “Trolley”. Con `Trolley_Body.CG` se obtiene el centro de masas del sólido, `Trolley_Body.I` muestra el tensor de inercia en el centro de masas y, `Trolley_Body.m` devuelve la masa del sólido.

```
1 ##### CODIGO #####
2 Trolley_Body.CG
3 ##### SALIDA #####
4 29.5160; 1.4246; 43.6045
5 ##### CODIGO #####
6 Trolley_Body.I
7 ##### SALIDA #####
8 ans =
9 1.0e+03 *
10
11 2.5231 -0.0015 -0.0037
12 -0.0015 1.5859 0.0232
13 -0.0037 0.0232 2.0106
14 ##### CODIGO #####
15 Trolley_Body.Mass
16 ##### SALIDA #####
17 2.6562e+03
```

Código 4.8: Función “`pcshow`” sobre el sólido “Trolley”.

El script completo para cada uno de los sólidos rígidos del sistema se muestra en el Código 8.2. En este ejecutable además se ha introducido la visualización con “`pcshow`” de cada uno de los sólidos

rígidos, así como el conjunto. La Figura 4.7, muestra todos los sólidos junto con el centro de masas de cada uno de ellos. La pluma y el mortero de cemento se han juntado en uno mismo debido a que, en cuanto a movimiento físico se refiere, ambos se moverán solidarios.

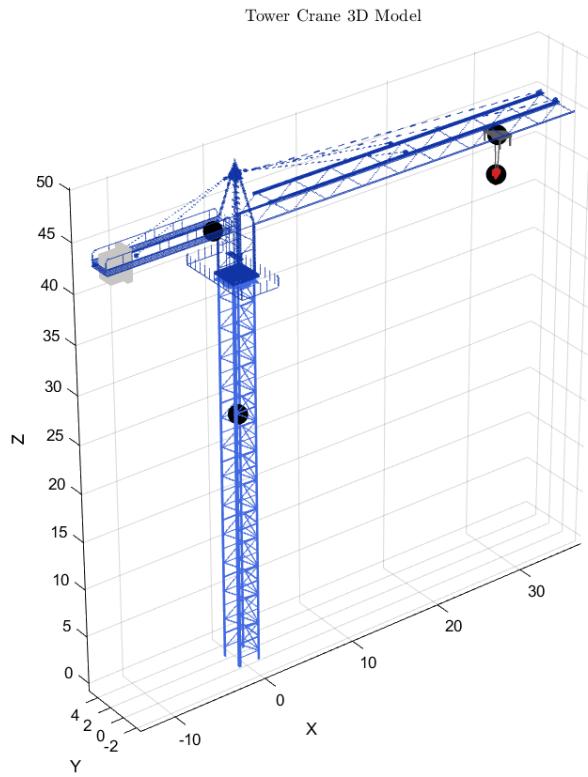


Figura 4.7: “pcshow” de sólidos discretizados de torre grúa.

4.2.4 Características de la torre grúa

En esta sección se exponen los parámetros calculados para la torre grúa que se está estudiando. Además, se detallan características específicas de las torres grúa que permiten clasificarlas según sus distintos diseños. Parámetros como la altura de la torre y la longitud de la pluma son esenciales, ya que nos proporcionan información crucial sobre su aplicación. Por ejemplo, una grúa con una torre más alta y una pluma más larga puede ser ideal para la construcción de rascacielos, mientras que una con dimensiones más modestas podría ser más adecuada para proyectos residenciales o industriales de menor envergadura.

Altura de la torre y tamaño de pluma

En las Figuras 4.8 y 4.9 se observan el lateral y la planta de la torre grúa que se está estudiando. Esta torre grúa en específico está diseñada con 44 metros de altura y 60 metros de pluma. Comúnmente, torres grúas de estas características pertenecen al grupo de torres grúas horizontales o *hammerhead*. Este tipo de grúas están diseñadas para manejar proyectos de construcción de gran envergadura. Un ejemplo real de grúa *hammerhead* es la Liebherr 550 EC-H 40 [5], la cual es utilizada en construcción de edificios altos, puentes y otras estructuras grandes. En la Figura 4.10 se representa un esquema de la torre grúa Liebherr 550 EC-H 40.

Masas de los sólidos rígidos

Para cada uno de los sólidos rígidos en los que se ha dividido la torre grúa se ha calculado la masa. Esta masa además de ser necesaria para el cálculo del centro de masas y las ecuaciones del movimiento, es un buen parámetro para tener en mente las magnitudes con las que se opera en esta clase de estructuras. Se indican a continuación las masas de calculadas para cada uno de los sólidos:

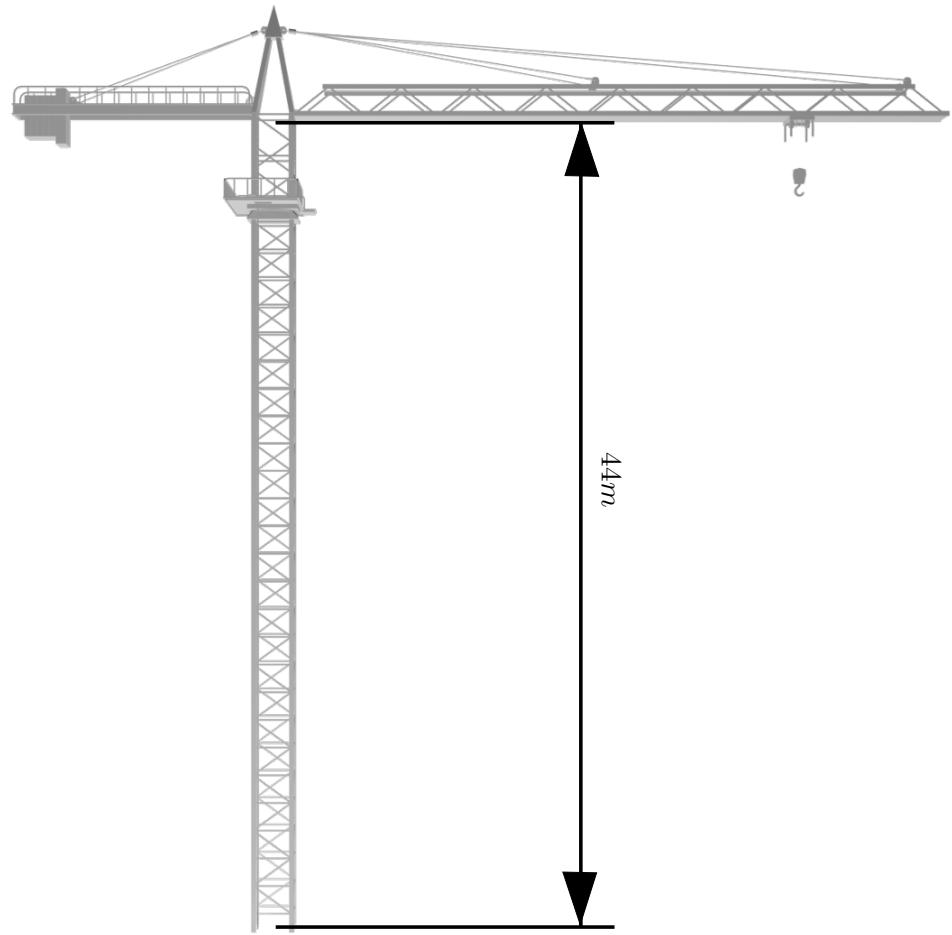


Figura 4.8: Vista lateral de torre grúa [4].

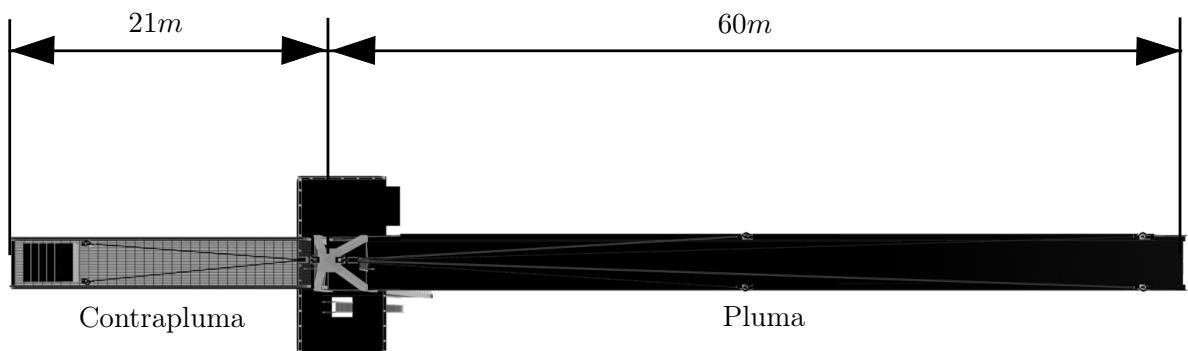


Figura 4.9: Vista en planta de la torre grúa [4].

- base de la torre grúa, 76 t;
- pluma y contrapluma, 175.5 t;
- contrapeso de mortero de cemento, 21.84 t,
- carro, 2.65 t y
- la carga, 1.11 t.

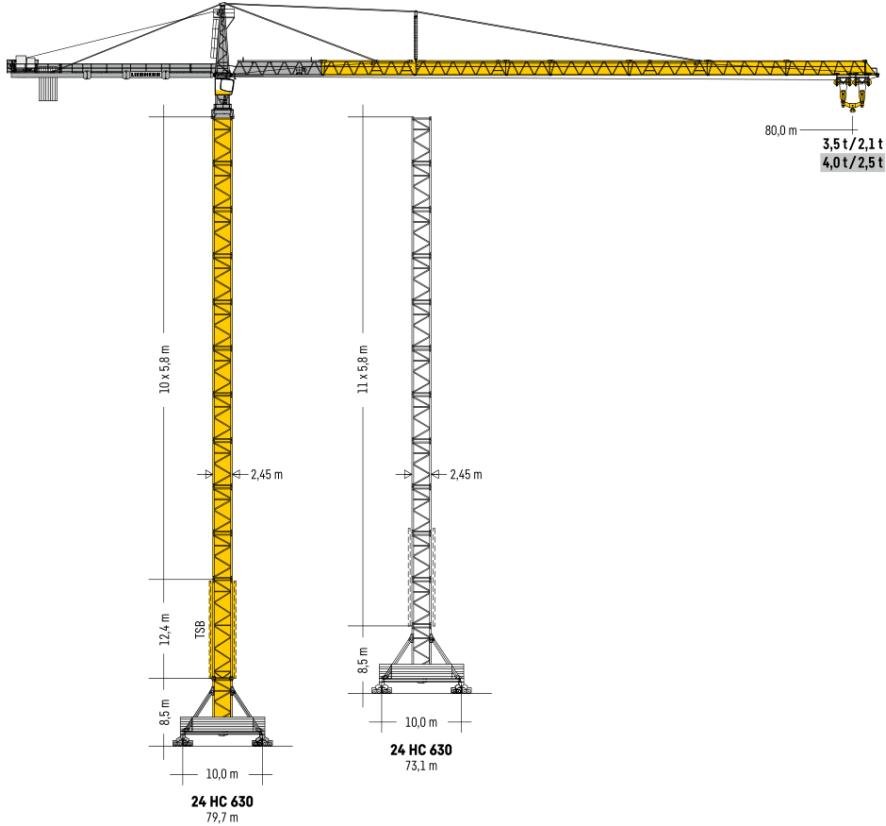


Figura 4.10: Liebher 550 EC-H 40 [5]. Modelo de torre grúa *hammerhead* comercializada por la empresa Liebherr.

4.2.5 Comparativa de discretización con torre grúa Liebher 550 EC-H 40

Para verificar que las magnitudes calculadas son coherentes con la realidad, se realizará una comparación entre el modelo adquirido en *Sketchfab* y la torre grúa Liebherr 550 EC-H 40 [5]. Dado que existen cinco sólidos a comparar, y no en todos Liebherr proporciona datos sobre la masa, la comparación se centrará en la torre y el contrapeso. Para diferenciar entre las dos torres grúa, se referirá al modelo 3D adquirido como “modelo” y a la torre grúa 550 EC-H 40 como “550”.

Comparativa de la torre (tramo base)

La Figura 4.11 presenta la relación entre la altura de la torre 550 y la masa de la misma. La torre 550 se compone de una base (tramo base) y de los elementos de la torre (tramo torre). En caso de tener una grúa con una altura de 44 metros (modelo estudiado en este proyecto), es necesaria una base y cinco tramos de torre. En total, una torre 550 con la altura del modelo tendría una masa de 47 t, lo cual en comparación a las 76 t obtenidas por discretización del modelo 3D, es considerablemente menor.

Es importante tener en cuenta que la diferencia puede deberse a que el fabricante puede estar usando diferentes materiales o aleaciones para la base y los tramos de torre, lo cual afectaría sus masas individuales. Además, el diseño tridimensional adquirido de *Sketchfab* no incluye información detallada sobre los materiales ni las densidades utilizadas. La suposición de una densidad constante y sólo dos tipos de material para toda la torre grúa contribuye a la diferencia entre la masa de una 550 real y la calculada para el modelo.

Comparativa del contrapeso

El contrapeso del modelo se compone de seis bloques de mortero de cemento con un ancho de 1.67 metros. Si se observa la Figura 4.12, uno de los bloques tiene una altura más grande que el resto. Debido a que estos cálculos son únicamente orientativos, todos los bloques se van a considerar del tipo

P _i	A _i	Beschreibung • Description • Description • Descripción • Descrizione • Descripción • Descrição • Описание	24 HC 630 TS-0580c	6,28 m (L)	2,45 m (H)
13	13	Turmstück • Tower section • Élement de mât • Elemento de torre • Tramo torre • Торре • Башенная секция			
14	1	Grundturmstück • Base tower section • Mât de base • Elemento di torre base • Tramo base • Peça de base da torre • Секция основания	24 HC 630 TSB-1242c	12,42 m (L)	2,67 m (H)

6450 kg^d

14500 kg^d

Figura 4.11: Datos de la torre del modelo Liebherr 550 EC-H 40 [5].

A de la 550 mostrado en la Figura 4.13. Teniendo en cuenta seis bloques de 2.88 t cada uno, la masa total del contrapeso es de 17.28 t, lo cual, se acerca considerablemente a las 21.84 t del modelo.

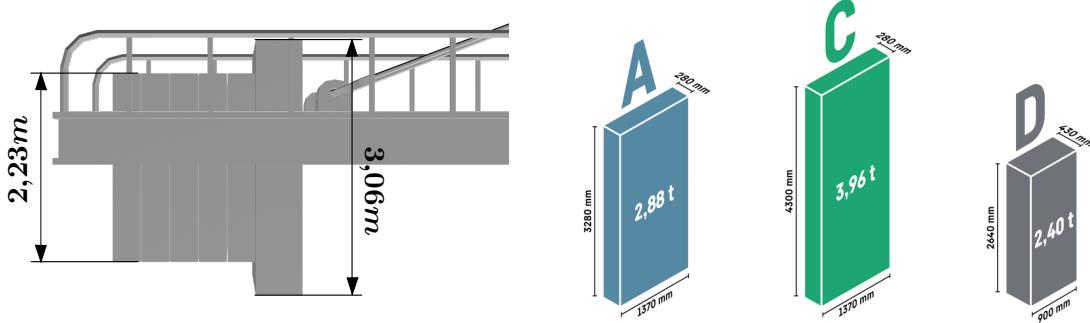


Figura 4.12: Contrapeso en el modelo tridimensional.

Figura 4.13: Datos del contrapeso en la torre grúa modelo Liebherr EC-H-40 [5].

4.3 Construcción del modelo dinámico

El modelo de la torre grúa se va a configurar a través de la librería “Dynamic_Library [13]”.

4.3.1 Inicialización y configuración del sistema

El sistema se define con cuatro grados de libertad través de las coordenadas generalizadas $\mathbf{q} = [\theta_1, r, \beta_2, \beta_1]^T$ como se muestra en la Figura 4.14. Sobre cada uno de los sólidos rígidos actúa la acción de la gravedad \mathbf{g} , se introduce un momento de entrada M_{in} en la pluma (Jib) y una fuerza de entrada F_{in} en el carro.

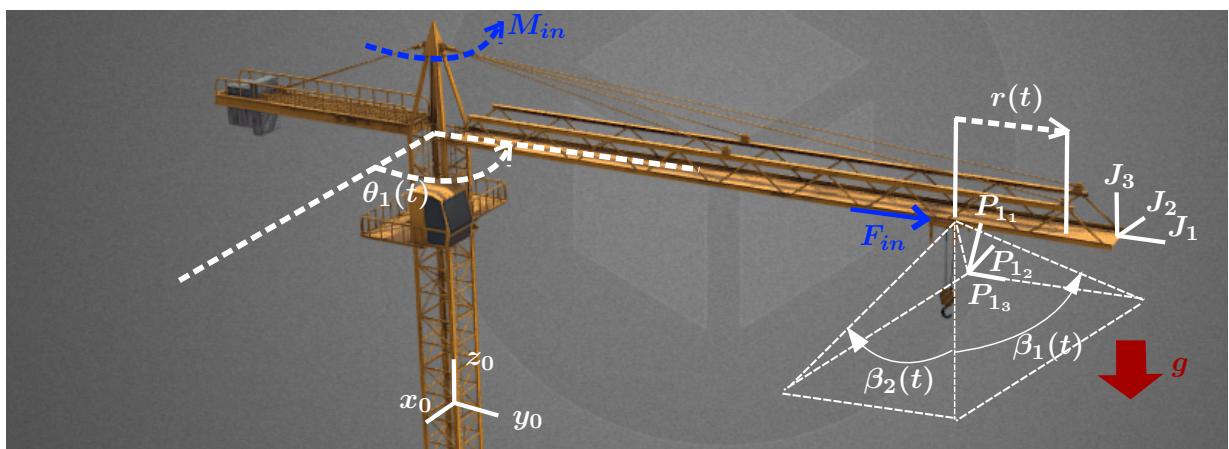


Figura 4.14: Grados de libertad y acciones de entrada en modelo de torre grúa.

La inicialización del sistema se realiza mediante el Código 8.3 que cumple las siguientes funciones:

- Incluir todas las coordenadas generalizadas \mathbf{q} en el sistema
- Crear las bases necesarias para el cálculo cinemático
- Posicionar los puntos para la localización de los centros de masas
- Introducir la acción de la gravedad
- Introducir el momento de entrada,
- Introducir la fuerza de entrada,
- Definir los sólidos rígidos a través de sus masas y sus tensores de inercia

4.3.2 Unión del sistema

Llegados a este punto, el siguiente paso a realizar es la unión del sistema. La función *system_unify* creada en la librería, realiza la unión numérica del sistema. Este tipo de unión requiere que todos los parámetros del sistema estén definidos numéricamente y que las únicas incógnitas dentro del sistema sean las coordenadas generalizadas \mathbf{q} . Con estos pasos previos, la librería es capaz de generar automáticamente funciones de Matlab que calculan la matriz $[\mathbf{A}]$, disminuyendo considerablemente el coste computacional.

La función *unify_system* es la encargada de generar automáticamente funciones que calculan las componentes de la matriz $[\mathbf{A}]$, la cual se calcula según

$$[\mathbf{A}^i]_{kj} = m_i \mathbf{b}_k^i \cdot \mathbf{b}_j^i + [\mathcal{I}_{G_i} \mathbf{c}_k^i]^T \cdot \mathbf{c}_j^i, \quad (4.1)$$

para cada uno de los sólidos. También es necesaria la derivada temporal de la matriz $[\mathbf{A}]$, la cual se compone a través de

$$[\mathbf{A}_v^i]_{kj} = m_i \mathbf{b}_{k_v}^i \cdot \mathbf{b}_j^i + [\mathcal{I}_{G_i} \mathbf{c}_{k_v}^i]^T \cdot \dot{\mathbf{c}}_j^i = [\mathbf{A}^i]_{kj}, \quad (4.2)$$

para cada uno de los sólidos. Las funciones en formato “.m” se generan automáticamente y se guardan en la carpeta “fun_handles”. Estas, tienen la forma que muestran las Figuras 4.15 y 4.16 y, son las que se utilizan para ensamblar numéricamente la matriz $[\mathbf{A}]$ y su derivada.

```

Virtual_Velocity_Components_uv_Jib_1.m
Virtual_Velocity_Components_uv_Jib_2.m
Virtual_Velocity_Components_uv_Jib_3.m
Virtual_Velocity_Components_uv_Jib_4.m
Virtual_Velocity_Components_uv_Load_1.m
Virtual_Velocity_Components_uv_Load_2.m
Virtual_Velocity_Components_uv_Load_3.m
Virtual_Velocity_Components_uv_Load_4.m
Virtual_Velocity_Components_uv_Trolley_1.m
Virtual_Velocity_Components_uv_Trolley_2.m
Virtual_Velocity_Components_uv_Trolley_3.m
Virtual_Velocity_Components_uv_Trolley_4.m

```

Figura 4.15: Componentes \mathbf{b}_k^i y \mathbf{c}_k^i de los sólidos “Jib”, “Trolley” y “Load” respecto a las coordenadas generalizadas θ_1 , r , β_2 y β_1

```

d_Virtual_Velocity_Components_uv_Jib_1.m
d_Virtual_Velocity_Components_uv_Jib_2.m
d_Virtual_Velocity_Components_uv_Jib_3.m
d_Virtual_Velocity_Components_uv_Jib_4.m
d_Virtual_Velocity_Components_uv_Load_1.m
d_Virtual_Velocity_Components_uv_Load_2.m
d_Virtual_Velocity_Components_uv_Load_3.m
d_Virtual_Velocity_Components_uv_Load_4.m
d_Virtual_Velocity_Components_uv_Trolley_1.m
d_Virtual_Velocity_Components_uv_Trolley_2.m
d_Virtual_Velocity_Components_uv_Trolley_3.m
d_Virtual_Velocity_Components_uv_Trolley_4.m

```

Figura 4.16: Componentes $\mathbf{b}_{k_v}^i$ y $\mathbf{c}_{k_v}^i$ de los sólidos “Jib”, “Trolley” y “Load” respecto a las coordenadas generalizadas θ_1 , r , β_2 y β_1

Debido a que la Ecuación 2.3 muestra una ecuación diferencial de segundo orden, *Dynamic_Library* realiza automáticamente la reducción a un sistema de primer orden. En las Figuras 4.15 y 4.16 los nombres de cada una de las funciones viene acompañado por un “uv”, haciendo hincapié en que las parciales de las velocidades se calculan a través de las variables \mathbf{u} y \mathbf{v} .

4.4 Implementación RK4 en Matlab

En la sección 2.2 se ha comentado que el método RK4 tiene mucha documentación que puede ser consultada, de hecho, la programación en Matlab viene inspirada en la propuesta por Brunton [7], como muestra en el Código 4.9. En este código las funciones matemáticas antes descritas en la Ecuación 2.13 vienen representadas por las variables f_i . Estas variables hacen uso del paso dt , el instante t_k y el valor inicial x_0 para obtener el valor en el instante de tiempo siguiente $xout$.

```

1 function xout = rk4singlestep(fun, dt, tk, xk)
2
3 f1 = fun(tk,xk);
4 f2 = fun(tk+dt/2,xk+(dt/2)*f1);
5 f3 = fun(tk+dt/2,xk+(dt/2)*f2);
6 f4 = fun(tk+dt, xk+dt*f3);
7
8 xout = xk +(dt/6)*(f1+2*f2+2*f3+f4);
9
10 end

```

Código 4.9: Código para implementar RK4 [7].

4.4.1 Parámetros de función RK4 con *inputs*

De forma análoga a lo presentado en la sección teórica del método Runge Kutta (Sección 2.3), este código se va a modificar para aceptar entradas en el sistema. En esta sección se van a detallar las variables de entrada que debe considerar el RK4 modificado para este caso particular.

Además de las entradas (*inputs*), en cada iteración se deberá calcular la matriz $[A]$, así como el vector $[b]$. El vector $[b]$ se construye a partir de la derivada temporal de la matriz de coeficientes, $\dot{[A]}$, y el vector Q , que indica las fuerzas generalizadas del sistema. Con estas construcciones y tras una reducción de orden, la Ecuación 2.3 se puede expresar de la forma

$$\begin{bmatrix} [A(Sist)(u,v)] & \mathbf{0} \\ \mathbf{0} & 1 \end{bmatrix} \begin{bmatrix} \dot{v} \\ \dot{u} \end{bmatrix} = \begin{bmatrix} Q(Sist)(u,v) - [A(Sist)(u,v)]v \\ v \end{bmatrix}. \quad (4.3)$$

Para el sistema todas las fuerzas generalizadas se tratan de la misma forma, pero para estudios de control, es conveniente separar las acciones en externas τ y en entradas Q_{in} como

$$Q = \tau + Q_{in}. \quad (4.4)$$

El cálculo de la matriz $[A]$, su derivada, y las fuerzas generalizadas, se materializa en la librería a través de las funciones *Get_A_tk_xk*, *Get_d_A_tk_xk*, *Get_tau_tk_xk* y *Get_input_tk_xk* como muestra el Código 4.10.

```

1 A_uv_ode = @Crane_System.Get_A_tk_xk;
2 d_A_uv_ode = @Crane_System.Get_d_A_tk_xk;
3 tau_uv_ode = @Crane_System.Get_tau_tk_xk;
4 input_uv_ode = @Crane_System.Get_input_tk_xk;

```

Código 4.10: Funciones para el cálculo de $[A]$, $\dot{[A]}$ y Q .

Estas funciones hacen uso de las funciones generadas en la carpeta “fun_handles” para realizar el cálculo de la matriz $[A]$ y el vector b . El símbolo “@” se introduce para crear un puntero a esas funciones o *handle*. Este *handle* tiene la capacidad de encapsular dentro de una variable una función, lo cual facilita introducir todas las funciones *Get_* dentro del bucle de iteración que se creará para el método método RK4.

Finalmente, se crea la función *rk4singlestep_Ab* con las siguientes entradas:

- el *handle* de la función que calcula la matriz $[A]$,
- el *handle* de la función que calcula la matriz $[A_v]$,
- el *handle* de la función que calcula el vector τ ,

- el *handle* de la función que calcula el vector \mathbf{Q}_{in} ,
- el *handle* de la función que calcula el vector \mathbf{v} ,
- el paso dt ,
- el tiempo t_k ,
- el valor inicial x_0 ,
- el valor en cada instante de los *inputs* y
- las variables simbólicas que representan $[\mathbf{v}, \mathbf{u}]^T$.

El Código 4.11 muestra el uso de esta función en el bucle de iteración para la simulación. El Código 8.2 muestra la función completa del método RK4 con entradas.

```
1 function xout = rk4singlestep_Ab(A, d_A, tau, in, v, dt, tk, xk, external_xin, vars
)
```

Código 4.11: Función para método RK4 método matricial.

4.5 Simulación

Antes de comenzar la simulación se deben especificar parámetros de la misma como:

- tiempo de simulación (*sim_time*),
- resolución de simulación (*dt*),
- valores iniciales cinemáticos (x_0) y
- los valores iniciales de las fuerzas y momentos ($M_{in_x_in}$ y $F_{in_x_in}$).

```
1 %% Compute trajectory
2 frames_per_second = 60;
3 dt = 1/frames_per_second;
4 sim_time = 8;
5 tspan = [0:dt:sim_time];
6 %x0 = [0;0;0;0;pi/4;0;-pi/4;-pi/4];
7 x0 = [0;0;0;0;0;0;0;0];
8 X=[];
9 X(:,1)=x0;
10 xin=x0;
11 external_xin = [];
12 sz = size(tspan);
13 M_in_x_in = [zeros(sz)+10000];
14 F_in_x_in = [[zeros(1,121);zeros(1,121)+0;zeros(1,121)+0],[zeros(1,120);zeros(1,120)+0;zeros(1,120)+0]]
```

Código 4.12: Parámetros de configuración de la simulación.

El Código 4.12 muestra la configuración de los parámetros que son necesarios para comenzar la simulación. Debido a que posteriormente se desea realizar una animación, la resolución se indica a través de los *frames* por segundo que se desean.

La función *rk4singlestep_Ab* resuelve la ecuación diferencial paso a paso y, debido a esto, se ejecuta en un bucle de iteraciones. En el Código 4.13 se muestra un bucle con el que se actualiza en cada iteración el tiempo (time) con el que se ejecuta el método RK4. En el, introducen los valores en el instante inicial de cada iteración *xin*, se obtienen las soluciones del paso siguiente *xout* y, son estas mismas soluciones de salida, las entradas para la proxima iteración.

```

1   for i=1:tspan(end)/dt
2     time = i*dt;
3     xout = rk4singlestep_Ab(A_uv_ode, ...
4                               d_A_uv_ode, ...
5                               tau_uv_ode, ...
6                               [input_uv_ode], ...
7                               v_ode, ...
8                               dt, ...
9                               time, ...
10                             xin, ...
11                             external_xin(:,i), ...
12                             vars);
13
14   X = [X xout];
15   xin = xout;
16 end

```

Código 4.13: Bucle de ejecución método RK4.

4.6 Animación en Blender

Blender es un programa dedicado al modelado, renderizado y animación para diversas plataformas. Se ha elegido Blender para la animación debido a que es un software libre con una amplia comunidad de usuarios. El lanzamiento inicial de Blender fue en 1994 [14] y, hasta el día de hoy, su uso continúa en aumento.

Además de ser una herramienta de software libre, Blender se utiliza en este proyecto porque ofrece una API en Python. Esta API permite acceder a los datos de los diferentes objetos en el espacio de trabajo, modificarlos y ejecutar comandos, lo que facilita la manipulación y automatización del modelo.



Figura 4.17: Logo de Blender



Figura 4.18: Python logo.

4.6.1 Matrices de transformación

El movimiento y transformación de objetos en Blender, se realiza a través de matrices de transformación. Las matrices de transformación 4x4 son herramientas esenciales en gráficos 3D, ya que permiten realizar operaciones de traslación, rotación, escalado y perspectiva de manera eficiente. Estas matrices son de tamaño 4x4 para poder representar todas las transformaciones en un espacio homogéneo [15], lo que facilita las combinaciones de transformaciones. En la Figura 4.19 se muestran las operaciones básicas así como su representación en matriz de transformación 4x4.

Estas matrices de transformación se deben calcular a partir de la simulación realizada en la Sección 4.5, y se deben importar en Blender en cada instante de tiempo. Para su cálculo, se propone el Código 4.14. En este código se amplía el bucle de iteración del método RK4 para calcular las matrices de transformación ($[T]$), este cálculo viene a partir de las funciones $T_{Jib_uv_ode}$, $T_{Trolley_uv_ode}$ y $T_{Load_uv_ode}$.

```

1   for i=1:tspan(end)/dt
2     time = i*dt;
3     xout = rk4singlestep_Ab(A_uv_ode, ...
4                               d_A_uv_ode, ...
5                               tau_uv_ode, ...
6                               [input_uv_ode], ...
7                               v_ode, ...
8                               dt, ...
9                               time, ...

```

$$\begin{array}{c}
 \begin{matrix} \text{X} & 0 & 0 & 0 \\ 0 & \text{Y} & 0 & 0 \\ 0 & 0 & \text{Z} & 0 \\ 0 & 0 & 0 & 1 \end{matrix} \quad \begin{matrix} 1 & 0 & 0 & \text{X} \\ 0 & 1 & 0 & \text{Y} \\ 0 & 0 & 1 & \text{Z} \\ 0 & 0 & 0 & 1 \end{matrix} \quad \begin{matrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{matrix} \\
 \text{Scale} \qquad \qquad \text{Translation} \qquad \qquad \text{No Change} \\
 (\text{Identity})
 \end{array}$$

$$\begin{array}{c}
 \begin{matrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\varphi) & -\sin(\varphi) & 0 \\ 0 & \sin(\varphi) & \cos(\varphi) & 0 \\ 0 & 0 & 0 & 1 \end{matrix} \quad \begin{matrix} \cos(\varphi) & 0 & \sin(\varphi) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\varphi) & 0 & \cos(\varphi) & 0 \\ 0 & 0 & 0 & 1 \end{matrix} \quad \begin{matrix} \cos(\varphi) & -\sin(\varphi) & 0 & 0 \\ \sin(\varphi) & \cos(\varphi) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{matrix} \\
 \text{Rotation along X} \qquad \qquad \text{Rotation along Y} \qquad \qquad \text{Rotation along Z} \\
 (\varphi = \text{Angle})
 \end{array}$$

Figura 4.19: Ejemplos de matrices de transformación.



Figura 4.20: Archivos CSV para *Jib*, *Trolley* y *Load*.

```

1 0.9999999999932432,-1.16248370992035e
2 -05,0,-1.94240076359682
3 1.16248370992035e
4 -05,0.9999999999932432,0,1.44567333610671
5 0,0,1,43.9496
6 0,0,0,1

```

Código 4.15: Matriz de transformación guardada en archivo CSV.

```

10 xin, ...
11 external_xin(:,i), ...
12 vars);
13 X = [X xout];
14 xin = xout;
15 T_Jib = T_Jib_uv_ode(time,xout);
16 writematrix(T_Jib,'Jib.csv','Delimiter','comma','WriteMode','append');
17 T_Trolley = T_Trolley_uv_ode(time,xout);
18 writematrix(T_Trolley,'Trolley.csv','Delimiter','comma','WriteMode','append');
19 T_Load = T_Load_uv_ode(time,xout);
20 writematrix(T_Load,'Load.csv','Delimiter','comma','WriteMode','append');
21 end

```

Código 4.14: Bucle de ejecución método RK4 con exportación de matrices de transformación.

Las matrices se escriben en un archivo CSV para importar a través de Python. El formato CSV (*Comma-Separated Values*, o Valores Separados por Comas) almacena datos tabulares (como una hoja de cálculo o una base de datos) en formato de texto plano. En la Figura 4.20 se muestran los tres archivos CSV guardados de la simulación del Código 4.14. El Código 4.15 muestra una matriz de transformación guardada en uno de los archivos CSV de la Figura 4.20.

4.6.2 Animación

La finalidad de este apartado es mostrar como, a través de la API de Python, se puede crear una animación del sistema mecánico de la torre grúa.

Importación de archivos STL

Blender permite la importación de modelos 3D en formato STL [16], de hecho, el comando para realizar esto se muestra en el Código 4.16. En este Código, “model_name” es una variable tipo *string* que representa el nombre del archivo STL a importar. La Figura 4.21 muestra el directorio de simulación en el que se puede ver los diferentes archivos STL de los sólidos que representan el modelo.

```
1 bpy.ops.import_mesh.stl(filepath=bpy.path.abspath("//stl//"+model_name+".stl"))
```

Código 4.16: Código para importar STL a espacio de trabajo Blender.

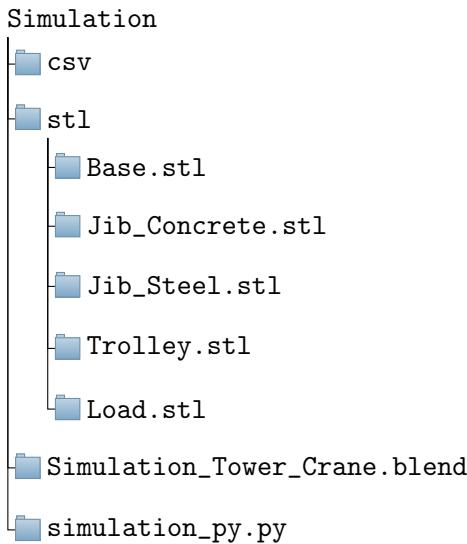


Figura 4.21: Directorio de simulación Blender.

Debido a que el centro de gravedad de cada uno de los sólidos no se importa con el archivo STL, se debe introducir manualmente. El posicionamiento del origen de cada sólido en Blender se realiza a través de un elemento denominado “cursor”. Esta idea se entiende mejor entendiendo el ejemplo del Código 4.17, que permite cambiar el origen de un sólido determinado. El cursor es simplemente un punto en el espacio de trabajo de Blender que se puede cambiar de posición a través de la propiedad *cursor.location*. En el ejemplo este cursor se mueve al punto [0,3513 1,3800 24,9501] del espacio. El centro de un sólido únicamente se puede llevar al cursor y esto se realiza con la función *solid.origin_set*. No existe forma directa de posicionar el centro de masas de los sólidos rígidos en la API de Blender si no es a través del cursor.

```
1 bpy.context.scene.cursor.location = [0.3513, 1.3800, 24.9501]
2 bpy.ops.solid.origin_set(type= 'ORIGIN_CURSOR')
```

Código 4.17: Comando para cambiar el centro del sólido al cursor.

El Código 4.18 es el código final con el que se importan los archivos STL y se posiciona cada uno de sus centros de gravedad en el espacio. En cada iteración el cursor se mueve a las posiciones indicadas por el vector “GC_Models”.

```
1 ##### IMPORT OF 3D Models
2 #Import stl files and change origin of each body to CG.
3 stl_names = ["Base", "Jib_Concrete", "Jib_Steel", "Trolley", "Load"]
4 GC_Models = [[0.3513, 1.3800, 24.9501], # CG Base
5               [-12.0440, 1.3862, 43.9635], # CG Jib_Concrete
6               [8.1592, 1.5052, 43.9357], # CG Jib_Steel
7               [29.5160, 1.4246, 43.6045], # CG Trolley
8               [29.4569, 1.3992, 39.6491]] # CG Load
9 for model_name,CG in zip(stl_names,GC_Models):
10     bpy.ops.import_mesh.stl(filepath=bpy.path.abspath("//stl//"+model_name+".stl"))
11
12     bpy.context.scene.cursor.location = CG
13     Body_Object = bpy.data.objects[model_name]
14     Body_Object.select_set(True)
15     bpy.ops.object.origin_set(type='ORIGIN_CURSOR')
16     Body_Object.select_set(False)
17
18     bpy.context.scene.cursor.location = [0,0,0]
```

Código 4.18: Importación de sólidos de torre grúa a Blender.

Importación de matrices de transformación

Cada uno de los sólidos importados a través del formato STL, tienen una propiedad denominada *matrix_world*, que define la matriz de transformación desde la base “mundo”. Esta base “mundo”, es la canónica desde la cual se realizan el resto de transformaciones. La generación de una animación radica en que la matriz de transformación de un sólido respecto a la base canónica (mundo), varíe con el tiempo.

Para importar las matrices de transformación generadas en la simulación desde la API de Python, se usan los comandos del Código 4.19. La función *genfromtxt* importa los archivos CSV como arrays y los asocia a las variables *T_Nombre_Del_Sólido_sim*.

```
1 T_Jib_sim = genfromtxt(bpy.path.abspath("//csv//Jib.csv"), delimiter=',')
2 T_Trolley_sim = genfromtxt(bpy.path.abspath("//csv//Trolley.csv"), delimiter=',')
3 T_Load_sim = genfromtxt(bpy.path.abspath("//csv//Load.csv"), delimiter=',')
```

Código 4.19: Importación matrices de transformación a través de archivos CSV a Blender.

Posteriormente, los arrays creados que contienen las matrices de transformación en cada instante de tiempo se utilizan para generar la animación. El Código 4.20 muestra como, para cada fotograma (*keyframe*), el sólido “Jib” se cambia de posición (*location*) y de orientación (*rotation_euler*).

```
1 for nT in enumerate(np.rollaxis(T_Jib_sim_Tensor, 2)):
2     n_frame = nT[0]+1
3     Jib_0bj.matrix_world = Matrix(nT[1])
4     bpy.context.view_layer.update()
5     Jib_0bj.keyframe_insert("location", frame=n_frame)
6     Jib_0bj.keyframe_insert("rotation_euler", frame=n_frame)
```

Código 4.20: Generación de animación en Blender.

El Código 8.5 muestra la programación completa para la importación y animación de la torre grúa.

5 Resultados de simulación

El conjunto de la librería dinámica, el modelo numérico y el método de resolución RK4 culminan en la simulación y animación de sistemas mecánicos holónomos. En esta sección se presentarán varias simulaciones del modelo de la torre grúa que permiten verificar y validar el correcto funcionamiento de la librería y el método de resolución numérica.

5.1 Movimiento pendular

La primera simulación a realizar tiene únicamente en cuenta la acción gravitatoria que actúa sobre la grúa. En caso de que las condiciones iniciales sean nulas, y la carga se mantenga paralela a la gravedad, no existirá movimiento en el sistema. Como este no es un caso de interés, se proponen las condiciones iniciales que se observan en la Figura 5.1, siendo estas $[\dot{\theta}_1, \dot{r}, \dot{\beta}_2, \dot{\beta}_1, \theta_1, r, \beta_2, \beta_1] = [0, 0, 0, 0, 0, 0, \pi/4, -\pi/4]$. Como se observa a través de la Figura 5.1, el convenio de variables utilizado en la Figura 4.14 se cumple, por lo tanto, se expone aquí la primera validación de la librería: las condiciones iniciales se cumplen en la simulación. Es importante visualizar este convenio directamente en el modelo tridimensional de la grúa. En la Figura 5.2 se observa que β_2 comienza en $\pi/4$ y β_1 en $-\pi/4$, pero con estos datos, a menos que se introduzcan en las matrices de cambio de base y se calcule la orientación respecto a la base canónica, no se puede saber de antemano si la simulación cumple con el convenio establecido.

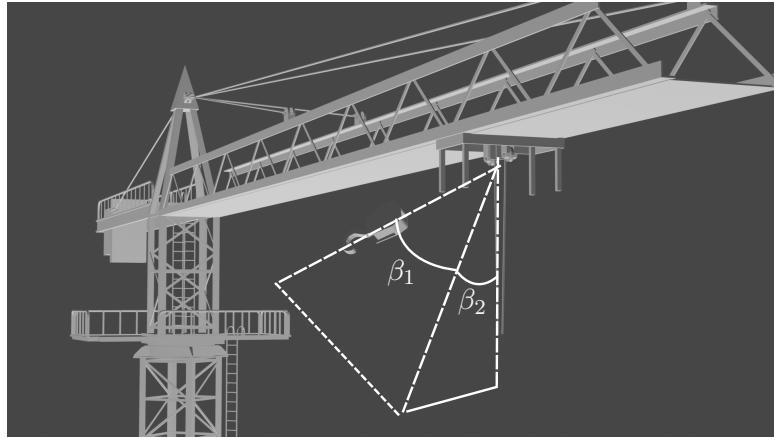


Figura 5.1: Condiciones iniciales simulación torre grúa movimiento pendular.

La siguiente validación se basa en visualizar la caída de la carga. Debido a que esta carga se comporta como un péndulo, la intuición permite predecir que comenzará a caer, hasta llegar a su punto más bajo y, posteriormente subirá. En la Figura 5.2 se observa que alrededor del primer segundo de simulación, β_2 y β_1 se anulan, es decir, en el entorno de 1 segundo es cuando la carga se encuentra en su punto más bajo. A partir de ahí, la carga comienza a elevarse aumentando (en valor absoluto) los ángulos β_2 y β_1 hasta llegar a otro máximo local (β_2 mínimo pero máximo en valor absoluto). Este pico, se produce en torno a los 2 segundos. Esta idea de bajada y subida de la carga se observa también en la secuencia de fotogramas presentada en la Figura 5.3. En el tiempo que transcurre desde el inicio de la simulación ($t = 0$) hasta el segundo 1,83 a la carga le da tiempo a caer y volver a subir. Esta validación deja claro que las leyes físicas que produce la librería son correctas. En [8] se había validado el cálculo

de las ecuaciones del movimiento simbólicamente para un péndulo y, en esta memoria se realiza para la carga de una torre grúa.

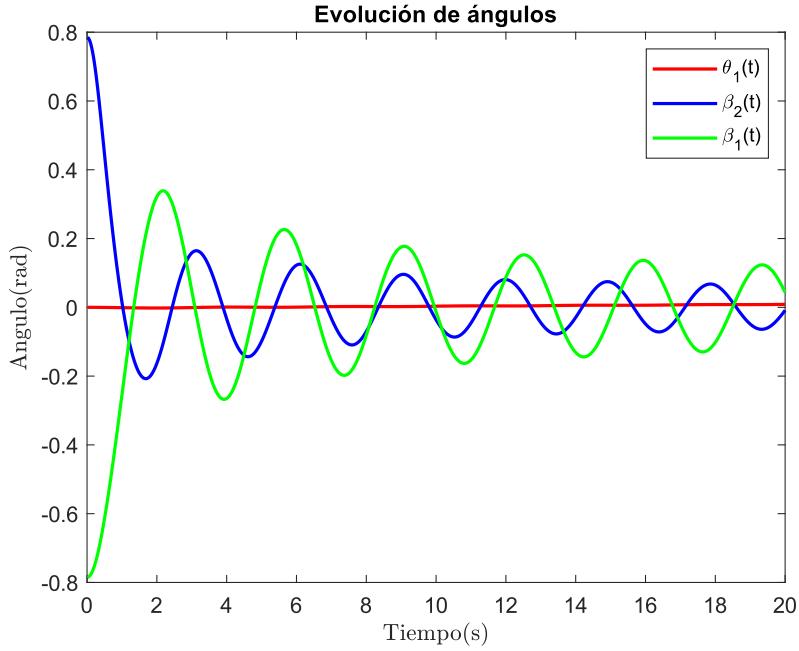


Figura 5.2: Coordenadas generalizadas angulares ($\theta_1, \beta_2, \beta_1$) en simulación de movimiento pendular.

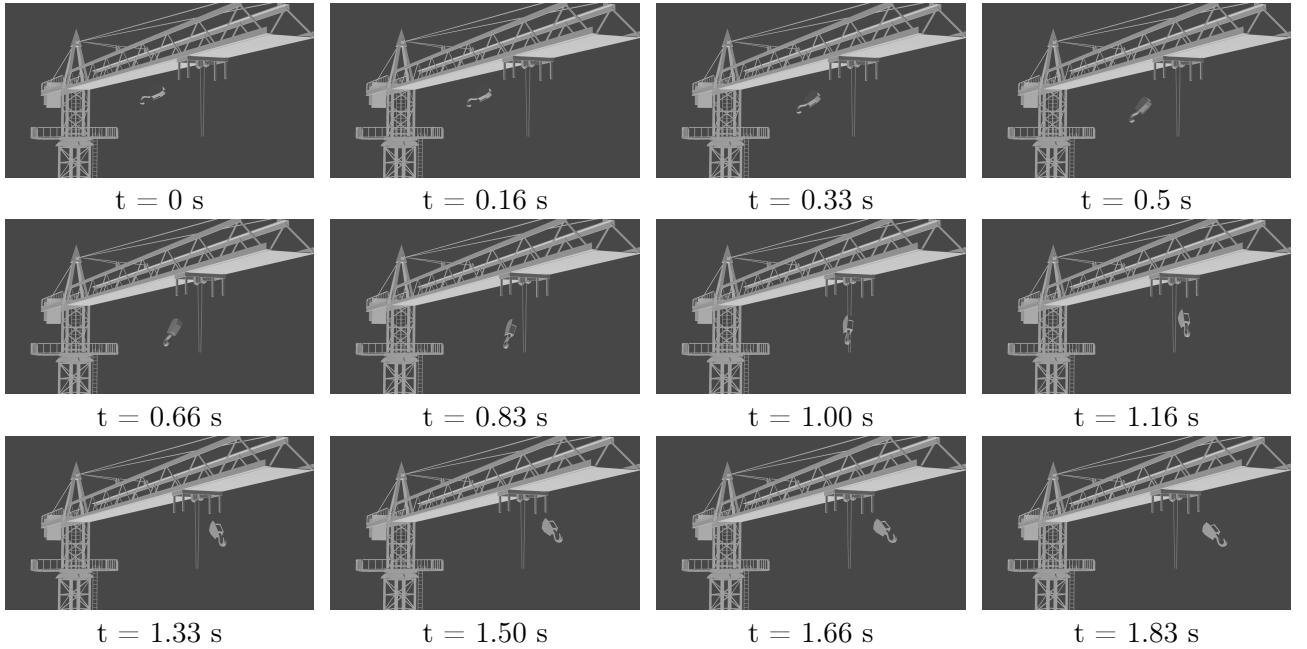


Figura 5.3: Secuencia de caída y subida de la carga

Es también interesante estudiar el movimiento de los demás sólidos en el modelo. La Figura 5.4, muestra cómo durante la caída la tensión del péndulo sobre el carro, y de este sobre la pluma, provoca que la torre gire en sentido negativo. Algo similar ocurre con el carro; la Figura 5.5 ilustra que la tensión entre el péndulo y el carro durante la caída hace que el carro retroceda temporalmente. Durante la subida, sucede lo contrario: la pluma gira en sentido positivo y el carro avanza. Este movimiento pendular se repite, haciendo que el carro avance progresivamente y que la pluma gire cada vez más en sentido positivo.

En la Sección 3 se mostró que es posible aproximar la torre grúa mediante figuras geométricas simples. Si estas figuras fueran simétricas, el movimiento de la torre bajo las condiciones iniciales

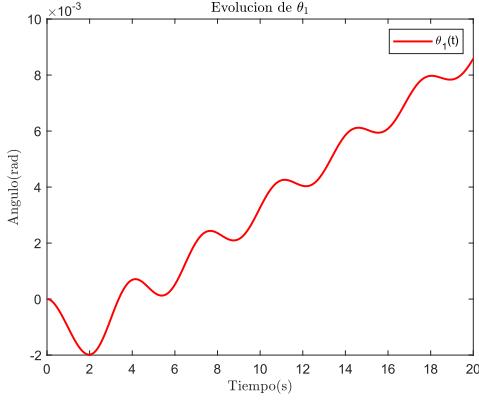


Figura 5.4: Coordenada generalizada θ_1 en simulación de movimiento pendular.

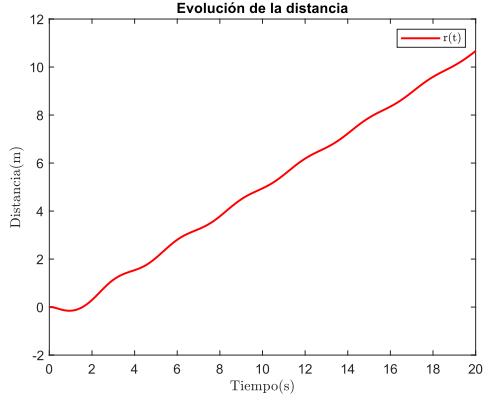


Figura 5.5: Coordenadas generalizadas lineales (r) en simulación de movimiento pendular.

estudiadas en esta sección, sería periódico y, ni el carro avanzaría ni la grúa giraría en sentido positivo. El movimiento hacia adelante y hacia atrás, y el giro positivo y negativo, se compensarían de forma periódica. Sin embargo, el hecho de que los elementos de la torre grúa no sean simétricos provoca que el carro avance y que la grúa gire en sentido positivo.

Finalmente, hay que destacar que el movimiento de la grúa debido al movimiento del péndulo es muy pequeño. La gran masa e inercia de la pluma, hace que en 20 segundos de simulación esta haya girado únicamente 0,008 radianes ($0,45^\circ$). Si a esta simulación se añade el efecto del rozamiento que existe en el giro de la torre, este movimiento prácticamente no se produciría.

5.2 Movimiento de carro

Para comprobar que las entradas se incluyen correctamente, se va a introducir una fuerza constante $F_{in} = 1kN$ al carro de la torre. Las condiciones iniciales serán nulas $[\dot{\theta}_1, \dot{r}, \dot{\beta}_2, \dot{\beta}_1, \theta_1, r, \beta_2, \beta_1] = [0, 0, 0, 0, 0, 0, 0, 0]$, por lo tanto, con la grúa desde el reposo, el carro comenzará a acelerar hacia delante.

En las Figuras 5.6 y 5.7 se observa que la fuerza escogida es lo suficientemente grande para desplazar el carro pero lo suficientemente pequeña para que la carga no pendule demasiado. De esta manera, una validación simple de esta simulación es tener en cuenta que el movimiento a estudiar es un movimiento rectilíneo uniformemente acelerado (M.R.U.A). La masa del M.R.U.A es el conjunto de carro y péndulo, los cuales suman 3.76 t. Por otro lado, la aceleración que sufre este subsistema es de $a = F/m = 1000N/3,76t = 0,27m/s^2$. Con estos parámetros la posición del carro en el tiempo viene dada por la Ecuación

$$r(t) = \frac{1}{2}at^2 = 0,133t^2. \quad (5.1)$$

La Figura 5.6 muestra la simulación realizada a través del método RK4 y con la Ecuación 5.1, como se observa, ambas soluciones son igual de correctas, validando de esta forma las fuerzas de entrada en la librería dinámica.

Con esta simulación, se puede apreciar una de las limitaciones de la librería. Si se observan las secuencias de la Figura 5.8, en los fotogramas finales, el carro sale de la pluma debido a que el enlace geométrico se sigue cumpliendo. Una alternativa realista a este movimiento es que, al llegar al final de la pluma, la torre grúa incorporado un tope que impidiera el avance del carro. Este tipo de restricciones que toman forma de inecuaciones como muestra

$$r(t) \leq r_{topo}, \quad (5.2)$$

son no holónomas y, como se muestra con este simple ejemplo, en ocasiones es crucial incluir este tipo de restricciones para representar el sistema de manera correcta.

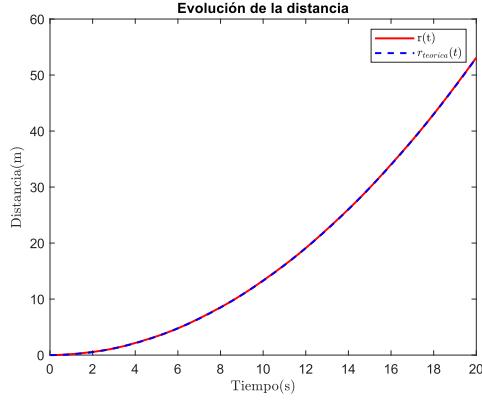


Figura 5.6: Coordenadas generalizadas lineales (r) en simulación de aceleración de la pluma.

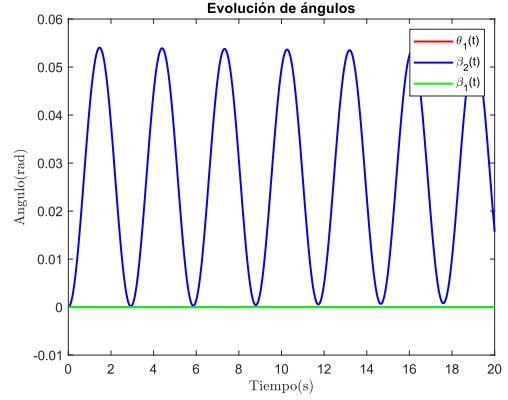


Figura 5.7: Coordenadas generalizadas angulares ($\theta_1, \beta_2, \beta_1$) en simulación de aceleración de carro.

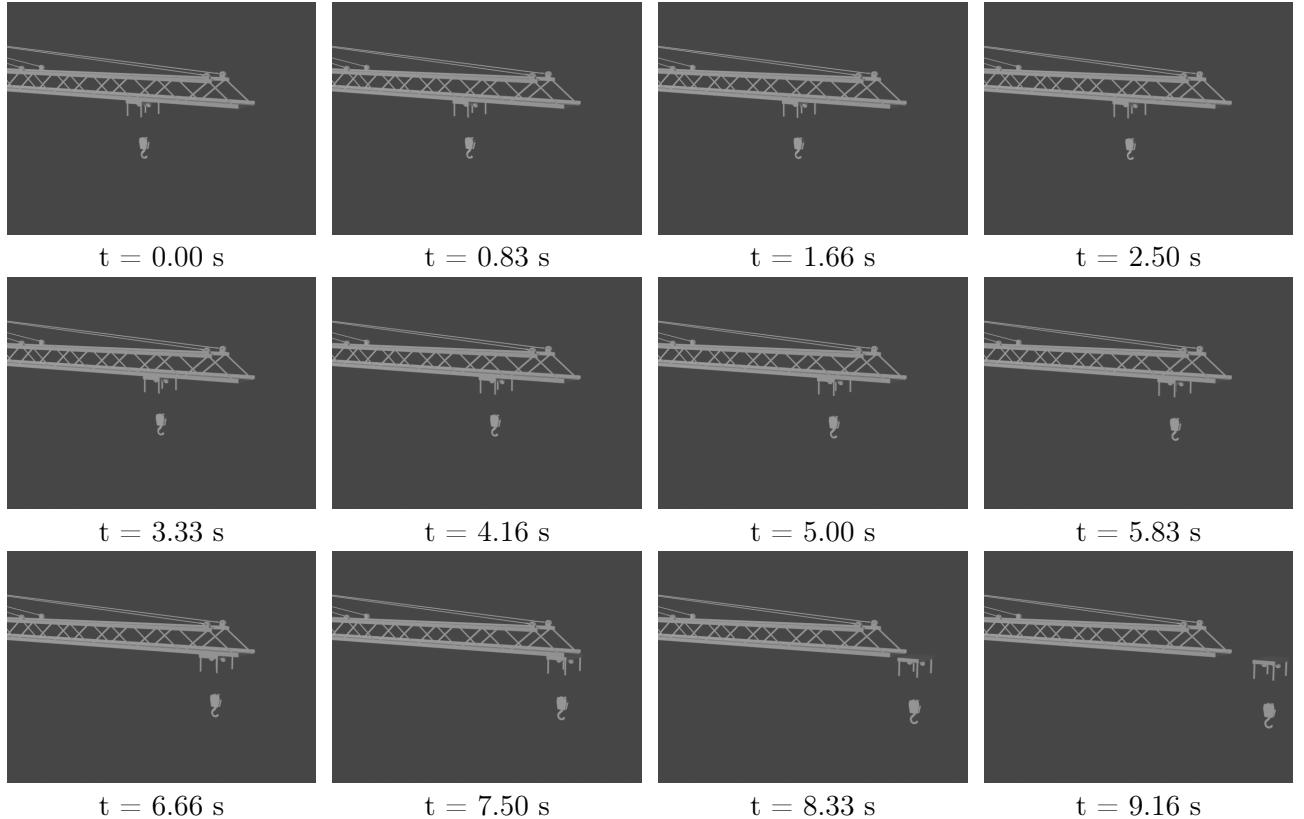


Figura 5.8: Secuencia de aceleración positiva carro.

5.3 Giro de la pluma

En esta tercera simulación se introduce un momento de entrada M_{in} en la pluma de la torre con el objetivo de observar el movimiento inducido en el carro y la carga. Este momento de entrada generará un movimiento circular uniformemente acelerado (M.C.U.A), el cual, produce una aceleración en la pluma que se puede descomponer en:

- Aceleración tangencial
- Aceleración normal

Estas dos componentes afectan al carro y a la carga generando una fuerza de inercia que hace que se desplacen en sentido contrario a dichas aceleraciones.

El caso más simple en este contexto es el carro. Debido a que este únicamente puede desplazarse a lo largo del eje de la pluma, la aceleración normal generará una fuerza de inercia (fuerza centrífuga) que hace que el carro se desplace hacia delante. En la Figura 5.10, se observa como el carro comienza retrocediendo debido a la tensión que genera la carga y, posteriormente, comienza a avanzar debido a la fuerza de inercia.

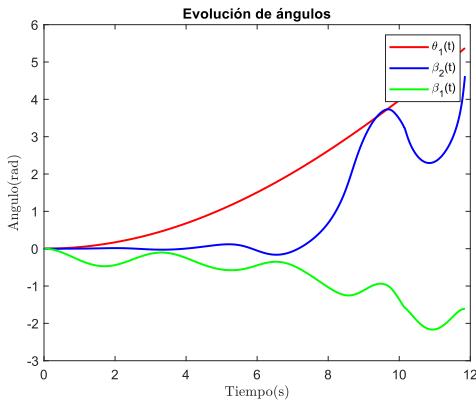


Figura 5.9: Coordenadas generalizadas angulares ($\theta_1, \beta_2, \beta_1$) en simulación de aceleración de la pluma.

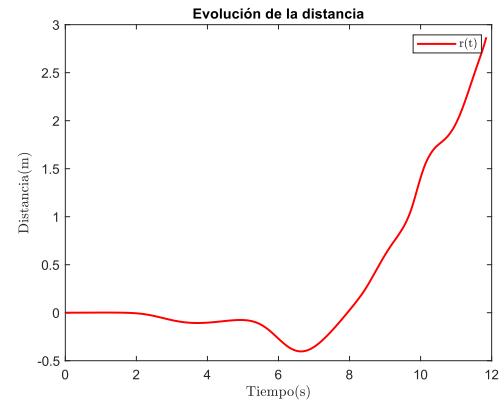


Figura 5.10: Coordenadas generalizadas lineales (r) en simulación de aceleración de la pluma.

El movimiento de la carga viene afectado por ambas aceleraciones, la normal y la tangencial. En la Figura 5.9, se puede observar el efecto de aceleración tangencial debido a que el ángulo β_1 disminuye. Este fenómeno también se ve en la secuencia presentada en la Figura 5.11, donde, a partir de los 5 segundos de simulación, la fuerza de inercia es suficiente para que la carga se eleve en sentido contrario al movimiento de la pluma.

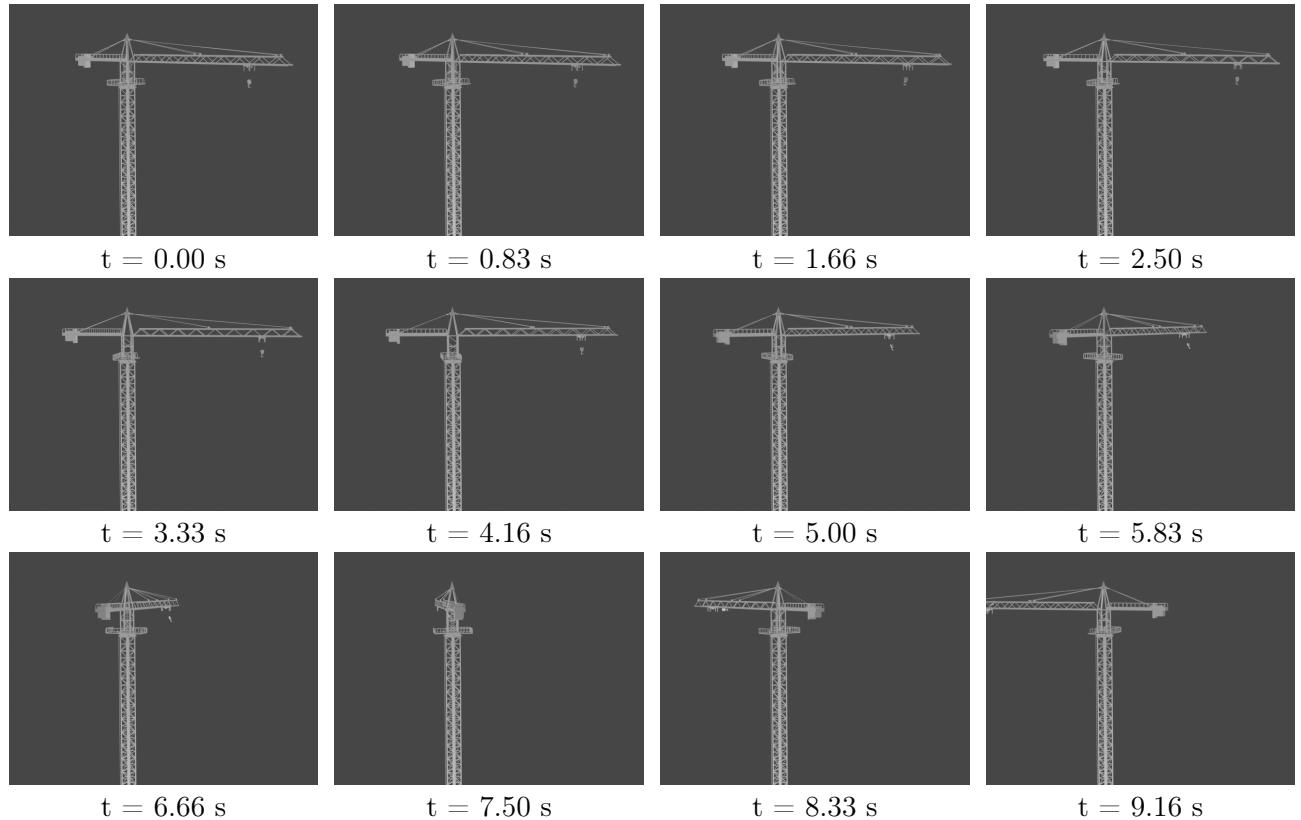


Figura 5.11: Secuencia de giro pluma.

5.4 Coste computacional

Las tres simulaciones de las Secciones 5.1, 5.2 y 5.3 se han realizado con ayuda de un microprocesador de escritorio Intel Core i5-1135G7 de 2.40GHz. Debido a que este no es un dispositivo especializado para realizar operaciones matriciales a altas frecuencias, los tiempos de simulación se pueden disminuir drásticamente con equipamiento adecuado. Si bien la actualización de componentes puede llevar a mejoras de procesado, un paso previo es tener en cuenta que la librería puede optimizarse para mejorar el rendimiento. De hecho, durante la programación de la librería *Dynamic_Library* la eficiencia no ha sido uno de los objetivos a la hora de crear la herramienta.

La optimización del tiempo de cómputo se basa en la mejora de los algoritmos. Estos algoritmos pueden realizar diversas funciones, desde el cálculo de ecuaciones del movimiento hasta la búsqueda de propiedades de una clase. Sin embargo, cada cambio en el código debe verificarse para asegurar que efectivamente mejora el rendimiento. Para ello, es esencial contar con información confiable sobre el desempeño del algoritmo e intentar reducir los tiempos de cómputo.

Por este motivo, se ha llevado un registro del tiempo de cómputo de cada una de las simulaciones. En la Figura 5.12 se muestra el tiempo de simulación en función del tiempo real de cómputo. Se puede observar que, para las tres simulaciones, el tiempo de resolución numérica es muy similar, siendo de aproximadamente 4.5 minutos de computación por cada segundo de simulación. Si se realizan futuras investigaciones sobre la mejora de los algoritmos, este dato será la base con la que poder comparar y asegurar la mejora de rendimiento.

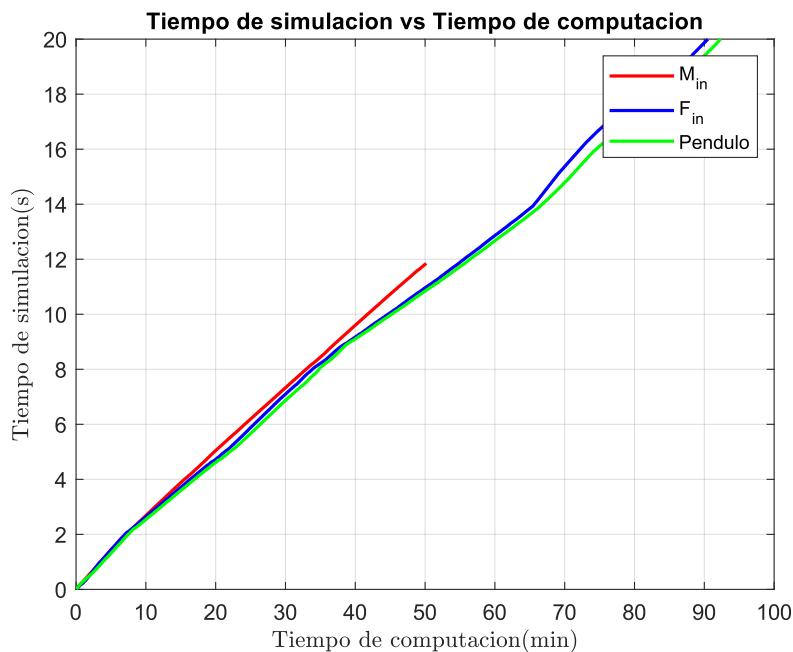


Figura 5.12: Comparativa en tiempo de cómputo de las simulaciones: movimiento pendular, F_{in} y M_{in}

6 Conclusiones

Teniendo en mente los objetivos que se han presentado al principio de esta memoria, se puede concluir que se han cumplido todos ellos.

La obtención de parámetros numéricos a través de la discretización (Sección 3) permite estudiar los sistemas de forma más realista. En este proyecto se ha conseguido calcular los parámetros mecánicos de una torre grúa con un modelo 3D como referencia. Esto supone una mejora respecto al modelo propuesto por [1] debido a que los sólidos rígidos estudiados ya no son geometrías simples, sino discretizaciones de un modelo realista. Además, se han aumentado los grados de libertad del modelo previo de 3 a 4 permitiendo un mayor movimiento de la carga.

La metodología de discretización de modelos 3D permite un rápido modelado de sistemas físicos, haciendo que el diseño de controladores se realice de forma más eficiente debido a que la flexibilidad para ajustarse a los cambios es mayor. Con este método, existe la posibilidad de comenzar a modelar en proyectos en los que no se conocen todas las características del diseño mecánico final.

Finalmente, las simulaciones de la Sección 4.5, ponen de manifiesto que no es necesario un cálculo simbólico de las EDM para obtener buenos resultados. De hecho, se vislumbra que la combinación de métodos simbólicos y numéricos puede tener un mejor desempeño. Este complemento da solución a uno de los inconvenientes presentados en el trabajo final de máster [8] en el cual obtener un sistema de ODEs que representen las EDM en simbólico tiene un alto coste computacional.

7 Lineas futuras

Control de errores numéricos

La resolución de ecuaciones diferenciales por métodos numéricos es una rama amplia de las matemáticas. En este trabajo, no se ha estudiado con la rigurosidad necesaria la precisión que ofrece el algoritmo RK4 utilizado, de hecho, si se alarga el tiempo de simulación, las soluciones comienzan a divergir. Una línea futura de investigación debería centrarse en evaluar la precisión de las soluciones obtenidas. Se sugiere el estudio de invariantes en los sistemas físicos. En sistemas mecánicos, existen magnitudes como la energía, el momento angular y el momento lineal que se conservan. Estas magnitudes pueden calcularse y monitorizarse a lo largo de la simulación para asegurar que se mantienen constantes, lo que ayudaría a validar la precisión del método numérico empleado.

Implementación de controladores

Los controladores que se pueden implementar en sistemas mecánicos pueden ser de diferentes tipos, pero la gran parte de los que permiten un buen control son los que se diseñan en cadena cerrada, es decir con realimentación. Incluir un sistema en un bucle con realimentación hace que se introduzcan nuevas variables dentro del modelo completo, así, las entradas ya no se estudian como algo externo al sistema (2.3), sino que pasan a ser parte del sistema mismo. Además, debido a que una gran parte de la teoría de control se basa en modelos lineales, la mayor parte de los controladores que se diseñan se estudian para modelos linealizados en torno a puntos de operación. Validar el correcto desempeño del controlador en el sistema no lineal se convierte entonces en una tarea imprescindible y, la simulación del conjunto modelo no lineal + controlador pasa a un plano importante en el diseño.

Coherencia en el uso de lenguajes de programación

A lo largo de la memoria se ha hecho uso de Matlab y Python como lenguajes de programación. También, la API de Blender ha servido para realizar las animaciones de las simulaciones realizadas. El paso entre estos lenguajes de programación y software induce costes computacionales que, aplicados a un gran proyecto, pueden llegar a ser considerables. Muchas veces existen herramientas necesarias que únicamente están programadas en un lenguaje de programación determinado o incluso software que realiza cierta función mejor que otro pero, en una amplia mayoría de casos, estos costes se pueden reducir empleando únicamente un lenguaje. La escalabilidad de este tipo de software depende en gran medida de la compatibilidad que todas las herramientas tienen entre ellas, por lo tanto, se propone el uso de Python para la programación de todas las librerías propuestas para su posterior unión.

Bibliografía

- [1] I. Erro, “Modelado y linealización de una grúa rotativa,” 2023.
- [2] J. H. Jiahui Ye, “Analytical analysis and oscillation control of payload twisting dynamics in a tower crane carrying a slender payload.” <https://www.sciencedirect.com/science/article/pii/S0888327021001588>, 2021.
- [3] J. Baert, “An accurate method for Voxelizing Polygon Meshes.” <https://www.forceflow.be/2012/10/26/an-accurate-method-for-voxelizing-polygon-meshes/>, 2012.
- [4] C. Prasantha, “Tower Crane.” <https://sketchfab.com/3d-models/tower-crane-49851dc7a51b43bda6aea06856c26a85>, 2022.
- [5] Liebherr, “Tower Cranes and Mobile Construction Cranes.” <https://www.liebherr.com/external/products/products-assets/ec6eb8f6-c4d3-4296-bc43-46d629b53a79-2/liebherr-550ec-h-40-litronic-datasheet.pdf>.
- [6] C. Pederkoff, “stl-to-voxel.” <https://github.com/cpederkoff/stl-to-voxel>, 2024.
- [7] S. Brunton, “Coding a Fourth-Order Runge-Kutta Integrator in Python and Matlab.” <https://www.youtube.com/watch?v=vNoFdtcPFdk>, 2022.
- [8] V. Coselev, “Método de cálculo de ecuaciones del movimiento para sistemas mecánicos holónomos. Librería en Matlab y aplicación a modelo de torre grúa.,” 2024.
- [9] Wikipedia, “List of Runge–Kutta methods.” https://en.wikipedia.org/wiki/List_of_Runge%E2%80%93Kutta_methods#cite_note-kutta-3.
- [10] E. T. Stoneking, “Implementation of Kane’s Method for a Spacecraft Composed of Multiple Rigid Bodies.” <https://ntrs.nasa.gov/api/citations/20160000805/downloads/20160000805.pdf>.
- [11] M. Deflorian and M. Rungger, “Generalization of an input-to-state stability preserving Runge–Kutta method for nonlinear control systems.” <http://dx.doi.org/10.1016/j.cam.2013.05.017>, 2014.
- [12] M. C. Sola, “STL Volume Model Calculator.” <https://github.com/mcanet/STL-Volume-Model-Calculator>, 2024.
- [13] V. Coselev, “Dynamic Library.” https://github.com/vcoselev/Dynamic_Library, 2024.
- [14] Wikipedia, “Blender.” [https://en.wikipedia.org/wiki/Blender_\(software\)](https://en.wikipedia.org/wiki/Blender_(software)).
- [15] Wikipedia, “Matriz de transformación.” https://es.wikipedia.org/wiki/Matriz_de_transformaci%C3%B3n, 2017.
- [16] Stackoverflow, “Import stl script blender.” <https://stackoverflow.com/questions/25083566/import-stl-script-blender>, 2016.

8 Anexo

8.1 Clase “Body” Matlab

```
1 classdef body < handle
2
3     properties(Access = public)
4         Name
5         Mass
6         Points
7         Density
8         Volume
9         x
10        y
11        z
12        CG
13        n
14        var_X
15        var_Y
16        var_Z
17        m_i
18        Ix
19        Iy
20        Iz
21        Ixy
22        Ixz
23        Iyz
24        I
25
26 end
27 methods(Access = public)
28     function obj = body(Name, Points, volume, density)
29         obj.Name = Name;
30         obj.Mass = volume*density;
31         obj.Points = Points;
32         obj.Density = density;
33         obj.Volume = volume;
34         obj.x = obj.Points(:,1);
35         obj.y = obj.Points(:,2);
36         obj.z = obj.Points(:,3);
37         obj.n = length(obj.x);
38         obj.CG = [sum(obj.x);sum(obj.y);sum(obj.z)]./obj.n;
39         obj.var_X = obj.CG(1)-obj.x;
40         obj.var_Y = obj.CG(2)-obj.y;
41         obj.var_Z = obj.CG(3)-obj.z;
42         obj.m_i = obj.Mass/obj.n;
43
44         obj.Ix = sum(obj.m_i*(obj.var_Y.^2+obj.var_Z.^2));
45         obj.Iy = sum(obj.m_i*(obj.var_X.^2+obj.var_Z.^2));
46         obj.Iz = sum(obj.m_i*(obj.var_Y.^2+obj.var_X.^2));
47
48         obj.Ixy = sum(obj.m_i*(obj.var_X.*obj.var_Y));
49         obj.Ixz = sum(obj.m_i*(obj.var_X.*obj.var_Z));
50         obj.Iyz = sum(obj.m_i*(obj.var_Y.*obj.var_Z));
51
52         obj.I =[obj.Ix, obj.Ixy, obj.Ixz;
```

```

52         obj.Ixy, obj.Iy, obj.Iyz;
53         obj.Ixz, obj.Iyz, obj.Iz];
54     end
55
56     function pcshow_body(obj)
57         pcshow([obj.var_X,obj.var_Y,obj.var_Z]);
58         title("Body"+ obj.Name);
59         xlabel("X");
60         ylabel("Y");
61         zlabel("Z");
62     end
63
64     function switch_axis(obj,x_axis, y_axis, z_axis)
65         switch(x_axis)
66             case "x"
67                 x_pos = 1;
68             case "y"
69                 x_pos = 2;
70             case "z"
71                 x_pos = 3;
72         end
73
74         switch(y_axis)
75             case "x"
76                 y_pos = 1;
77             case "y"
78                 y_pos = 2;
79             case "z"
80                 y_pos = 3;
81         end
82
83         switch(z_axis)
84             case "x"
85                 z_pos = 1;
86             case "y"
87                 z_pos = 2;
88             case "z"
89                 z_pos = 3;
90         end
91         obj.Points = [obj.Points(:,x_pos),obj.Points(:,y_pos),obj.Points(:,z_pos)]
92     ];
93         obj.x = obj.Points(:,1);
94         obj.y = obj.Points(:,2);
95         obj.z = obj.Points(:,3);
96         obj.n = length(obj.x);
97         obj.CG = [sum(obj.x);sum(obj.y);sum(obj.z)]./obj.n;
98         obj.var_X = obj.CG(1)-obj.x;
99         obj.var_Y = obj.CG(2)-obj.y;
100        obj.var_Z = obj.CG(3)-obj.z;
101        obj.m_i = obj.Mass/obj.n;
102
103        obj.Ix = sum(obj.m_i*(obj.var_Y.^2+obj.var_Z.^2));
104        obj.Iy = sum(obj.m_i*(obj.var_X.^2+obj.var_Z.^2));
105        obj.Iz = sum(obj.m_i*(obj.var_Y.^2+obj.var_X.^2));
106
107        obj.Ixy = sum(obj.m_i*(obj.var_X.*obj.var_Y));
108        obj.Ixz = sum(obj.m_i*(obj.var_X.*obj.var_Z));
109        obj.Iyz = sum(obj.m_i*(obj.var_Y.*obj.var_Z));
110
111        obj.I =[obj.Ix, obj.Ixy, obj.Ixz;
112                  obj.Ixy, obj.Iy, obj.Iyz;
113                  obj.Ixz, obj.Iyz, obj.Iz];
114    end
115 end

```

116 end

Código 8.1: Clase “Body” en Matlab.

8.2 Script Matlab: Cálculo de tensores de inercia y masas

```

1 clear
2
3 Steel_Density = 7850; %Kg/m3
4 Concrete_Density = 2400 %Kg/m3
5
6 Base_Points = readmatrix("txt\Base.txt");
7 Base_Volume = readmatrix("txt\Base_Volume.txt");
8
9 Jib_Concrete_Points = readmatrix("txt\Jib_Concrete.txt");
10 Jib_Concrete_Volume = readmatrix("txt\Jib_Concrete_Volume.txt");
11
12 Jib_Steel_Points = readmatrix("txt\Jib_Steel.txt");
13 Jib_Steel_Volume = readmatrix("txt\Jib_Steel_Volume.txt");
14
15 Trolley_Points = readmatrix("txt\Trolley.txt");
16 Trolley_Volume = readmatrix("txt\Trolley_Volume.txt");
17
18 Load_Points = readmatrix("txt\Load.txt");
19 Load_Volume = readmatrix("txt\Load_Volume.txt");
20
21 pcshow([Base_Points; Jib_Steel_Points; Jib_Concrete_Points; Trolley_Points;
22     Load_Points])
23 title("All");
24 xlabel("X");
25 ylabel("Y");
26 zlabel("Z");
27
28 %% Base
29 Base_Body = body("Base",Base_Points,Base_Volume,Steel_Density);
30 %Base_Body.switch_axis("y","x","z");
31 Base_Body.Points;
32 Base_Body.pcshow_body;
33
34
35 CG_Base_Body = sum(Base_Body.Points)/Base_Body.n;
36 Base_Body.I
37
38 %% Jib Concrete
39
40
41 Jib_Concrete_Body = body("Jib_Concrete",Jib_Concrete_Points,Jib_Concrete_Volume,
42     Concrete_Density);
43 %Jib_Concrete_Body.switch_axis("y","x","z");
44 Jib_Concrete_Body.pcshow_body;
45 CG_Jib_Concrete_Body = sum(Jib_Concrete_Body.Points)/Jib_Concrete_Body.n;
46 Jib_Concrete_Body.I
47
48 %% Jib Steel
49
50 Jib_Steel_Body = body("Jib_Steel",Jib_Steel_Points,Jib_Steel_Volume,Steel_Density);
51 %Jib_Steel_Body.switch_axis("y","x","z");
52 Jib_Steel_Body.pcshow_body;
53 Jib_Steel_Body.I
54 CG_Jib_Steel_Body = sum(Jib_Steel_Body.Points)/Jib_Steel_Body.n;
55 CG_Jib_Body = (CG_Jib_Concrete_Body+CG_Jib_Steel_Body)/2

```

```

56 CG_Jib_Body -CG_Base_Body
57 %% Load
58
59
60 Load_Body = body("Load",Load_Points,Load_Volume,Steel_Density);
61 %Load_Body.switch_axis("y","x","z");
62 Load_Body.pcshow_body;
63 Load_Body.I
64 CG_Load_Body = sum(Load_Body.Points)/Load_Body.n;
65 CG_Load_Body -CG_Base_Body
66 %% Trolley
67
68
69 Trolley_Body = body("Trolley",Trolley_Points,Trolley_Volume,Steel_Density);
70 %Trolley_Body.switch_axis("y","x","z");
71 Trolley_Body.pcshow_body;
72 Trolley_Body.CG
73 Trolley_Body.I
74 Trolley_Body.Mass
75
76 CG_Trolley_Body = sum(Trolley_Body.Points)/Trolley_Body.n;
77
78 %% Figure
79
80 x0=10;
81 y0=10;
82 width=1080;
83 height=720;
84 f = figure(1)
85 set(f,'position',[x0,y0,width,height])
86 pcshow([Base_Body.Points],[0x41 0x69 0xe1],'BackgroundColor','white')
87 hold on
88 pcshow([Jib_Steel_Body.Points],[0x10 0x34 0xa6],'BackgroundColor','white')
89 pcshow([Jib_Concrete_Body.Points],[0xc8 0xc6 0xc4],'BackgroundColor','white')
90 pcshow([Trolley_Body.Points],[0x5d 0x5e 0x61],'BackgroundColor','white')
91 pcshow([Load_Body.Points],[0xd9 0x21 0x21],'BackgroundColor','white')
92 pcshow([CG_Base_Body; (CG_Jib_Concrete_Body+CG_Jib_Steel_Body)/2; CG_Trolley_Body;
93 CG_Load_Body],'green','MarkerSize',200,'BackgroundColor','white')
94 title("Tower Crane 3D Model",'interpreter','latex');
95 xlabel("X");
96 ylabel("Y");
97 zlabel("Z");
98 legend("Base","Jib(Steel)","Jib(Concrete)","Load",'interpreter','latex','Location',
99 'southeast')

```

Código 8.2: Script para el cálculo de los tensores de inercia en CG de sólidos.

8.3 Script Matlab: Sistema dinámico de torre grúa y simulación

```

1 clear
2 %syms g
3 g = 9.81;
4 Crane_System = Dynamic_Library.System("Crane");
5 %Creamos el origen de nuestro sistema.
6 Crane_System.Create_New_Point("N","Canonical",sym([0;0;0]));
7 Crane_System.Create_New_Base("B_0",'axis_labels',[str2sym("x_0");str2sym("y_0");
8 str2sym("z_0")]);
9 Crane_System.Create_New_Coordinate_System("C_0","N","B_0");
10 Crane_System.Create_New_Generalized_Coordinate("theta_1",str2sym("theta_1(t)"));
11
12 OG_Base = sym([0.3513; 1.38; 24.9501]);

```

```

13 %Crane_System.Create_New_Point("N_1","C_0",sym([0; 0; str2sym("h"))));
14 Crane_System.Create_New_Point("G_Base","C_0",OG_Base);
15 Crane_System.Create_New_Coordinate_System("G_Base_CS","G_Base","B_0");
16 G_Base_Join_Point_Base_Jib = sym([0; 0; 18.9995]);
17 Crane_System.Create_New_Point("Join_Point_Base_Jib","G_Base_CS",
18     G_Base_Join_Point_Base_Jib);
19 Crane_System.Create_New_Base("Jib", ...
20     'axis_labels',[str2sym("J_1");str2sym("J_2");str2sym("J_3")], ...
21     ...
22     'father_base','B_0',...
23     'angle',str2sym("theta_1(t)'), ...
24     'axis_rotation','3');
25 Crane_System.Create_New_Coordinate_System("Referencial_Rotation_Axis_Jib",
26     "Join_Point_Base_Jib","Jib");
27
28 %Creamos el referencial del carrito
29
30 Crane_System.Create_New_Generalized_Coordinate("r",str2sym("r(t)"));
31 Join_Point_Base_Jib_G_Trolley = sym([29.1647+str2sym("r(t)");0.0446;-0.3451]);
32 Crane_System.Create_New_Point("A_ast","Referencial_Rotation_Axis_Jib",
33     Join_Point_Base_Jib_G_Trolley);
34 Crane_System.Create_New_Coordinate_System("Referencial_Trolley","A_ast","Jib");
35
36 Crane_System.Create_New_Generalized_Coordinate("beta_2",str2sym("beta_2(t)"));
37 Crane_System.Create_New_Base("Bar_beta_2", ...
38     'axis_labels',[str2sym("P_1_2");str2sym("P_2_2");str2sym("P_3_2")], ...
39     ...
40     'father_base','Jib',...
41     'angle',str2sym("beta_2(t)'), ...
42     'axis_rotation','2');
43
44 Crane_System.Create_New_Generalized_Coordinate("beta_1",str2sym("beta_1(t)"));
45 Crane_System.Create_New_Base("Bar_beta_1", ...
46     'axis_labels',[str2sym("P_1_1");str2sym("P_2_1");str2sym("P_3_1")], ...
47     ...
48     'father_base',"Bar_beta_2", ...
49     'angle',str2sym("beta_1(t)'), ...
50     'axis_rotation','1');
51
52 % Crane_System.Create_New_Generalized_Coordinate("Gamma",str2sym("Gamma(t)"));
53 % Crane_System.Create_New_Base("Bar", ...
54 %     'axis_labels',[str2sym("P_1");str2sym("P_2");str2sym("P_3")], ...
55 %     ...
56 %     'father_base',"Bar_beta_1", ...
57 %     'angle',str2sym("Gamma(t)'), ...
58 %     'axis_rotation','3');
59
60 % Crane_System.Create_New_Coordinate_System("Referencial_Trolley_Bar","A_ast","Bar");
61 Crane_System.Create_New_Coordinate_System("Referencial_Trolley_Bar","A_ast",
62     "Bar_beta_1");
63 %Crane_System.Create_New_Generalized_Coordinate("ly",str2sym("ly(t)"));
64 %Crane_System.Create_New_Point("P_ast","Referencial_Trolley_Bar",sym([0;0;-str2sym("ly(t)")]));
65 ly = 1;
66 G_Trolley_G_Load = sym([0;0;ly-3.9560]);
67 Crane_System.Create_New_Point("P_ast","Referencial_Trolley_Bar",G_Trolley_G_Load);
68 %Creamos los solidos rigidos del sistema.
69
70 %Crane_System.Create_New_Point("G_Jib","Referencial_N_Jib",sym([str2sym("x(CG_Jib")
71     ;0;0]));
72 Join_Point_Base_Jib_G_Jib = sym([-2.2937;0.0657;0]);
73 Crane_System.Create_New_Point("G_Jib","Referencial_Rotation_Axis_Jib",
74     Join_Point_Base_Jib_G_Jib)
75 I_Jib_Concrete = 1.0e+04 *[1.7719    0    0;
76                           0    2.1950    0;
77                           0    0    1];

```

```

67          0      0      1.5159];
68 Mass_Jib_Concrete = 2.1872e+04;
69
70 I_Jib_Steel = [0.1513      0      0;
71           0      3.1988      0;
72           0      0.1824    3.0953]*1.0e+07;
73
74 Mass_Jib_Steel = 1.7551e+05;
75
76 I_Jib = I_Jib_Concrete+I_Jib_Steel;
77
78 Mass_Jib = Mass_Jib_Concrete + Mass_Jib_Steel;
79
80 % I_Jib = sym([]);
81 % for i = 1:3
82 %     for j = 1:3
83 %         if i <= j
84 %             str = "I_Jib_"+string(i)+string(j);
85 %             I_Jib(i,j)=str2sym(str);
86 %             Crane_System.Create_New_Parameter(str,str2sym(str));
87 %
88 %         else
89 %             str = "I_Jib_"+string(j)+string(i);
90 %             I_Jib(i,j)=str2sym(str);
91 %         end
92 %     end
93 % end
94 % Mass_Jib = str2sym("m_Jib");
95 Crane_System.Create_New_Rigid_Body("Jib", ...
96             'G_Point','G_Jib',...
97             'Base','Jib',...
98             'Mass',Mass_Jib, ...
99             'Intertial_Tensor',I_Jib);
100
101 Crane_System.Create_New_Action("Gravity_Jib",...
102             'Point','G_Jib', ...
103             'Base','B_0', ...
104             'Rigid_Body','Jib', ...
105             'Vector',sym([0;0;0;0;0;-Mass_Jib*g]));
106 Vec_Angular_Velocity_Jib = Crane_System.Angular_Velocity("B_0","Jib","B_0");
107 cin_vis = 200E-4; % m^2/s
108 density = 850; % Kg/m^3
109 mu = cin_vis * density;
110 r = 1.35;
111 h= 0.4;
112 e = 0.05*r;
113 c = mu/e;
114 A = 2*pi*r*h;
115 c = 100000
116 Coulomb_Friction_Moment = -h*pi*r^2*(r+1)*c*A*Vec_Angular_Velocity_Jib;
117 Crane_System.Create_New_Action("Coulomb_Friction_Moment",...
118             'Point',"Join_Point_Base_Jib", ...
119             'Base','B_0', ...
120             'Rigid_Body','Jib', ...
121             'Vector',[Coulomb_Friction_Moment;0;0;0]);
122 Crane_System.Create_New_Input("Moment_In_Jib",...
123             'Point',"Join_Point_Base_Jib", ...
124             'Base','B_0', ...
125             'Rigid_Body','Jib', ...
126             'Vector',sym([0;0;str2sym("tau_in_1(t)")+str2sym(" ...
127             tau_in_2(t)");0;0;0]), ...
128             'External_Variables',[str2sym("tau_in_1(t)");str2sym(" ...
129             tau_in_2(t)")]);

```

```

130 0      1.5859    0;
131 0      2.0106];
132
133 Mass_Trolley = 2.6562e+03;
134
135 % I_Trolley = sym([]);
136 % for i = 1:3
137 %     for j = 1:3
138 %         if i <= j
139 %             str = "I_Trolley_"+string(i)+string(j);
140 %             I_Trolley(i,j)=str2sym(str);
141 %             Crane_System.Create_New_Parameter(str,str2sym(str));
142 %
143 %         else
144 %             str = "I_Trolley_"+string(j)+string(i);
145 %             I_Trolley(i,j)=str2sym(str);
146 %         end
147 %     end
148 % end
149 % Mass_Trolley = str2sym("m_Trolley");
150 Crane_System.Create_New_Rigid_Body("Trolley", ...
151     'G_Point', "A_ast", ...
152     'Base', "Jib", ...
153     'Mass', Mass_Trolley, ...
154     'Intertial_Tensor', I_Trolley);
155
156 Crane_System.Create_New_Action("Gravity_Trolley", ...
157     'Point', "A_ast", ...
158     'Base', "B_0", ...
159     'Rigid_Body', "Trolley", ...
160     'Vector', sym([0;0;0;0;0;-Mass_Trolley*g]));
161
162 Crane_System.Create_New_Input("Force_In_Trolley", ...
163     'Point', "A_ast", ...
164     'Base', "Jib", ...
165     'Rigid_Body', "Trolley", ...
166     'Vector', sym([0;0;0;str2sym("F_in_1(t)")+str2sym(" ...
167     F_in_2(t)")+str2sym("r(t)");str2sym("F_in_3(t)");0;0]), ...
168     'External_Variables', [str2sym("F_in_1(t)");str2sym(" ...
169     F_in_2(t)");str2sym("F_in_3(t)")]);
170
171 %Crane_System.Create_New_Point("Centroid_Surface_Trolley","Canonical",sym ...
172 ([29.486520767211914; 1.4035983085632324; 43.916645936279295]))
173 Crane_System.Create_New_Point("Centroid_Surface_Trolley", ...
174     'Referencial_Rotation_Axis_Jib',sym([29.1352+str2sym("r(t)");0.0236;-0.0330]))
175 a = 1.38;
176 b = 2.38;
177 A = a*b;
178 c = 100000;
179 G_Trolley_To_GCentroid_Trolley = Crane_System.Get_Two_Points_Vector("A_ast", ...
180     "Centroid_Surface_Trolley");
181 Centroid_Surface_Trolley_Obj = Crane_System.Get_Point_Info("System_Points",'point', ...
182     "Centroid_Surface_Trolley");
183 V_Centroid_Surface_Trolley = Centroid_Surface_Trolley_Obj{2}.Get_Info(" ...
184     Point_Velocity_Coordinates_From_Canonical");
185 F_Vis_Trolley = -c*A*V_Centroid_Surface_Trolley;
186 M_Vis_Trolley = simplify(cross(G_Trolley_To_GCentroid_Trolley,F_Vis_Trolley));
187 Crane_System.Create_New_Action("Friction_Trolley", ...
188     'Point', "Centroid_Surface_Trolley", ...
189     'Base', "Jib", ...
190     'Rigid_Body', "Trolley", ...
191     'Vector',[M_Vis_Trolley;F_Vis_Trolley]);
192
193 I_Load = [145.5235    0      0;
194 0      168.1075   5.8392;
195 0      5.8392   43.3397];

```

```

188 Mass_Load = 1.1109e+03;
189 % I_Load = sym([]);
190 % for i = 1:3
191 %     for j = 1:3
192 %         if i <= j
193 %             str = "I_Load_"+string(i)+string(j);
194 %             I_Load(i,j)=str2sym(str);
195 %             Crane_System.Create_New_Parameter(str,str2sym(str));
196 %
197 %         else
198 %             str = "I_Load_"+string(j)+string(i);
199 %             I_Load(i,j)=str2sym(str);
200 %         end
201 %     end
202 % end
203 % Mass_Load = str2sym("m_Load");
204 Crane_System.Create_New_Rigid_Body("Load", ...
205 %                                         'G_Point','P_ast',...
206 %                                         'Base','Bar_beta_1',...
207 %                                         'Mass',Mass_Load, ...
208 %                                         'Intertial_Tensor',[0,0,0;0 0 0;0 0 0]);
209
210 % Crane_System.Create_New_Rigid_Body("Load", ...
211 %                                         'G_Point','P_ast',...
212 %                                         'Base','Bar',...
213 %                                         'Mass',Mass_Load, ...
214 %                                         'Intertial_Tensor',I_Load);
215
216 Crane_System.Create_New_Action("Gravity_Load",...
217 %                                 'Point','P_ast', ...
218 %                                 'Base','B_0', ...
219 %                                 'Rigid_Body','Load', ...
220 %                                 'Vector',sym([0;0;0;0;0;-Mass_Load*g]));
221 addpath fun_handles\
222 Crane_System.unify_system
223
224 Crane_System.Get_input_tk_xk(0,[0;0;0;0;pi/4;1;-pi/4;-pi/4],[1;2;3;4;5])
225
226 A_uv_ode = @Crane_System.Get_A_tk_xk;
227 d_A_uv_ode = @Crane_System.Get_d_A_tk_xk;
228 tau_uv_ode = @Crane_System.Get_tau_tk_xk;
229 input_uv_ode = @Crane_System.Get_input_tk_xk;
230
231 T_Jib_uv_ode = Crane_System.Transformation_Matrix_Handles{1,3};
232 T_Trolley_uv_ode = Crane_System.Transformation_Matrix_Handles{2,3};
233 T_Load_uv_ode = Crane_System.Transformation_Matrix_Handles{3,3};
234 vars = [Crane_System.v;Crane_System.u];
235 v_ode = odeFunction(Crane_System.v,vars,"File","v_function");
236 %% Compute trajectory
237 frames_per_second = 60;
238 dt = 1/frames_per_second;
239 sim_time = 8;
240 tspan = [0:dt:sim_time];
241 %x0 = [0;0;0;0;pi/4;0;-pi/4;-pi/4];
242 x0 = [0;0;0;0;0;0;0];
243 X=[];
244 X(:,1)=x0;
245 xin = x0;
246 external_xin = [];
247 sz = size(tspan);
248 %M_in_x_in = [zeros(sz)+0;zeros(sz)+0];
249 F_in_x_in = [zeros(sz)+3000;zeros(sz)+0;zeros(sz)+0];
250 M_in_x_in = [[zeros(1,241)+1000000;zeros(1,241)+2000000],[zeros(1,240)-3000000;zeros(1,240)-4000000]];
251 %F_in_x_in = [[zeros(1,121)+10000;zeros(1,121)+0;zeros(1,121)+0],[zeros(1,120)-10000];

```

```

zeros(1,120)+0; zeros(1,120)+0]]
```

252
253 **external_xin** = [M_in_x_in;F_in_x_in];
254 **for** i=1:tspan(**end**)/dt
255 time = i*dt;
256 xout = rk4singlestep_Ab(A_uv_ode, ...
257 d_A_uv_ode, ...
258 tau_uv_ode, ...
259 [input_uv_ode], ...
260 v_ode, ...
261 dt, ...
262 time, ...
263 xin, ...
264 external_xin(:,i), ...
265 vars);
266 X = [X xout];
267 xin = xout;
268 T_Jib = T_Jib_uv_ode(time,xout);
269 writematrix(T_Jib,'Jib.csv','Delimiter','comma','WriteMode','append');
270 T_Trolley = T_Trolley_uv_ode(time,xout);
271 writematrix(T_Trolley,'Trolley.csv','Delimiter','comma','WriteMode','append');
272 T_Load = T_Load_uv_ode(time,xout);
273 writematrix(T_Load,'Load.csv','Delimiter','comma','WriteMode','append');
274 **%if** mod(i,10) == 0
275 disp(i);
276 disp(xout.);
277 disp(string(datetime));
278 **%end**
279 **end**
280
281 **%%**
282 plot(tspan,X(5:**end**,:),'LineWidth',1)
283 legend(string(vars(5:**end**)))

Código 8.3: Script de configuración del sistema mecánico de torre grúa y simulación.

8.4 Función Matlab: Método RK4 con entradas

```

1 function xout = rk4singlestep_Ab(A, d_A, tau, in, v, dt, tk, xk, external_xin,  

vars)  

2 A_1 = A(tk,xk);  

3 d_A_1 = d_A(tk,xk);  

4 tau_1 = tau(tk,xk);  

5 input_1 = in(tk,xk,external_xin);  

6 v_1 = v(tk,xk);  

7 b_1 = -d_A_1*v_1+tau_1+input_1;  

8 f1 = [A_1\b_1; v_1];  

9  

10  

11 A_2 = A(tk+dt/2,xk+(dt/2)*f1);  

12 d_A_2 = d_A(tk+dt/2,xk+(dt/2)*f1);  

13 tau_2 = tau(tk+dt/2,xk+(dt/2)*f1);  

14 input_2 = in(tk+dt/2,xk+(dt/2)*f1,external_xin);  

15 v_2 = v(tk+dt/2,xk+(dt/2)*f1);  

16 b_2 = -d_A_2*v_2+tau_2+input_2;  

17 f2 = [A_2\b_2; v_2];  

18  

19 A_3 = A(tk+dt/2,xk+(dt/2)*f2);  

20 d_A_3 = d_A(tk+dt/2,xk+(dt/2)*f2);  

21 tau_3 = tau(tk+dt/2,xk+(dt/2)*f2);  

22 input_4 = in(tk+dt/2,xk+(dt/2)*f2, external_xin);  

23 v_3 = v(tk+dt/2,xk+(dt/2)*f2);  

24 b_3 = -d_A_3*v_3+tau_3+input_4;
```

```

25   f3 = [A_3\b_3 ; v_3];
26
27   A_4 = A(tk+dt, xk+dt*f3);
28   d_A_4 = d_A(tk+dt, xk+dt*f3);
29   tau_4 = tau(tk+dt, xk+dt*f3);
30   input_4 = in(tk+dt, xk+dt*f3, external_xin);
31   v_4 = v(tk+dt, xk+dt*f3);
32   b_4 = -d_A_4*v_4+tau_4+input_4;
33   f4 = [A_4\b_4; v_4];
34
35   xout = xk +(dt/6)*(f1+2*f2+2*f3+f4);
36
37 end

```

Código 8.4: Función para método RK4 con entradas.

8.5 Script Python: Animación de la torre grúa en Blender

```

1 import bpy
2 import math
3 from mathutils import Matrix
4 import numpy as np
5 from numpy import genfromtxt
6 ### IMPORT OF 3D Models
7 #Import stl files and change origin of each body to CG.
8 stl_names = ["Base","Jib_Concrete","Jib_Steel","Trolley","Load"]
9 GC_Models = [[0.3513, 1.3800, 24.9501],
10               [-12.0440, 1.3862, 43.9635],
11               [8.1592, 1.5052, 43.9357],
12               [29.5160, 1.4246, 43.6045],
13               [29.4569, 1.3992, 39.6491]]
14 for model_name,CG in zip(stl_names,GC_Models):
15     bpy.ops.import_mesh.stl(filepath=bpy.path.abspath("//stl//"+model_name+".stl"))
16     bpy.context.scene.cursor.location = CG
17     Body_Object = bpy.data.objects[model_name]
18     Body_Object.select_set(True)
19     bpy.ops.object.origin_set(type='ORIGIN_CURSOR')
20     Body_Object.select_set(False)
21
22     bpy.context.scene.cursor.location = [0,0,0]
23
24     Base_Obj = bpy.data.objects["Base"]
25     Trolley_Obj = bpy.data.objects["Trolley"]
26     Load_Obj = bpy.data.objects["Load"]
27
28     #Join the Jib Bodies
29     j_1 = bpy.data.objects["Jib_Concrete"] # or whatever object you want
30     j_2 = bpy.data.objects["Jib_Steel"]
31     selected_objects = [
32         j_1,
33         j_2
34     ]
35     j_1.select_set(True)
36     j_2.select_set(True)
37     with bpy.context.temp_override(active_object=j_1, selected_objects=
38     selected_objects):
39         bpy.ops.object.join()
40     Jib_Obj = Jib_Steel_Obj = bpy.data.objects["Jib_Concrete"]
41     #Change Name to Jib and origin to CG.
42     Jib_Obj.name = 'Jib'
43     CG_Jib = [-1.9424, 1.4457, 43.9496]
44     Jib_Obj.select_set(True)

```

```

44 bpy.context.scene.cursor.location = CG_Jib
45 bpy.ops.object.origin_set(type='ORIGIN_CURSOR')
46 Jib_Obj.select_set(False)
47
48 bpy.context.scene.cursor.location = [0,0,0]
49
50 ### SIMULATION
51
52
53 T_Jib_sim = genfromtxt(bpy.path.abspath("//csv//Jib.csv"), delimiter=',')
54 T_Trolley_sim = genfromtxt(bpy.path.abspath("//csv//Trolley.csv"), delimiter=',')
55 T_Load_sim = genfromtxt(bpy.path.abspath("//csv//Load.csv"), delimiter=',')
56
57 T_Jib_sim_Tensor = np.zeros((4,4,int(len(T_Jib_sim)/4)))
58 T_Trolley_sim_Tensor = np.zeros((4,4,int(len(T_Trolley_sim)/4)))
59 T_Load_sim_Tensor = np.zeros((4,4,int(len(T_Load_sim)/4)))
60
61 for i in range(int(len(T_Jib_sim)/4)):
62     T_Jib_sim_Tensor[:, :, i] = T_Jib_sim[4*i:4*i+4, 0:4]
63 for i in range(int(len(T_Trolley_sim)/4)):
64     T_Trolley_sim_Tensor[:, :, i] = T_Trolley_sim[4*i:4*i+4, 0:4]
65 for i in range(int(len(T_Load_sim)/4)):
66     T_Load_sim_Tensor[:, :, i] = T_Load_sim[4*i:4*i+4, 0:4]
67
68
69 for nT in enumerate(np.rollaxis(T_Jib_sim_Tensor, 2)):
70     n_frame = nT[0]+1
71     Jib_Obj.matrix_world = Matrix(nT[1])
72     bpy.context.view_layer.update()
73     Jib_Obj.keyframe_insert("location", frame=n_frame)
74     Jib_Obj.keyframe_insert("rotation_euler", frame=n_frame)
75
76
77 for nT in enumerate(np.rollaxis(T_Trolley_sim_Tensor, 2)):
78     n_frame = nT[0]+1
79     Trolley_Obj.matrix_world = Matrix(nT[1])
80     bpy.context.view_layer.update()
81     Trolley_Obj.keyframe_insert("location", frame=n_frame)
82     Trolley_Obj.keyframe_insert("rotation_euler", frame=n_frame)
83
84 for nT in enumerate(np.rollaxis(T_Load_sim_Tensor, 2)):
85     n_frame = nT[0]+1
86     Load_Obj.matrix_world = Matrix(nT[1])
87     bpy.context.view_layer.update()
88     Load_Obj.keyframe_insert("location", frame=n_frame)
89     Load_Obj.keyframe_insert("rotation_euler", frame=n_frame)
90

```

Código 8.5: Programación completa para la importación y animación de la torre grúa.