TRABAJO FIN DE GRADO

Grado en Ingeniería Mecánica

UNIVERSIDAD DE ALMERIA

ESCUELA SUPERIOR DE INGENIERÍA

Diseño de material interactivo para los estudiantes de “Teoría de Mecanismos”

Curso: 2020/2021

Modalidad TFG: Técnico-Experimental

Alumno/a:

Jesús Sánchez García

Director/es:

Antonio Giménez Fernández

José Luis Torres Moreno

Agradecimientos

Quiero dedicar unas líneas para agradecer a todos los que han estado a mi alrededor durante estos años apoyándome en este camino.

Agradecer a mi familia, en especial a mis padres todo el apoyo y dedicación que me han dado estos años. Sin ellos tengo muy claro que esto hubiera sido imposible.

También quiero agradecer a los compañeros que me han rodeado estos años de carrera, ellos han conseguido que este duro camino se haya hecho mucho más ameno.

También, por supuesto, quiero agradecer a mis tutores Antonio y José Luis por la ayuda que me han proporcionado para realizar este trabajo y dar este paso tan importante.

Muchas gracias a todos.

Índice

Resumen

El proyecto llevado a cabo consiste en el desarrollo de un material de apoyo interactivo para mejorar la forma de explicar y entender la asignatura de Teoría de Mecanismos. Se basa en la creación de un código en el programa Jupyter Notebook (programa de la plataforma Anaconda y que utiliza lenguaje Python para su implementación posterior en el libro en formato PDF) con el cual será posible mostrar el comportamiento de los distintos mecanismos de una manera más interesante para el alumno ya que en lugar de una imagen podrán ser ellos mismos los que creen el mecanismo.

Palabras clave: Teoría de Mecanismos, mecanismo, Jupyter Notebook y Python.

Abstract

The project carried out consists of the development of an interactive support material to improve the way of explaining and understanding the subject of Mechanisms Theory. It is based on the creation of a code in the Jupyter Notebook program (program of the Anaconda platform and that uses Python language for its later implementation in the book in PDF format) with which it will be possible to show the behavior of the different mechanisms in a more interesting way for the student since instead of an image they will be able to create the mechanism themselves.

Keywords: Teoría de Mecanismos, mecanismo, Jupyter Notebook and Python.

Índice de figuras

1 INTRODUCCIÓN

1.1 Introducción y motivación del TFG

El conocimiento del movimiento, las fuerzas, el trabajo, y la energía de cualquier punto de las piezas que se mueven de una máquina, es un aspecto fundamental del trabajo de un Ingeniero Mecánico, que se dedique al diseño de máquinas.

Un buen conocimiento de la Teoría de Cinemática y Dinámica de Mecanismos puede ayudar al estudiante de titulaciones de Ingeniería en el ámbito Industrial en la enseñanza de las asignaturas.

Actualmente, no hay muchos libros de texto que vinculen el conocimiento del libro de texto clásico de mecánica vectorial para ingenieros con los artículos clásicos sobre diseño de máquinas y los métodos utilizados en su enseñanza.

En los cursos de mecanismos impartidos en el programa de grado, se estudian métodos tradicionales de investigación de dinámica y cinemática de mecanismos basados en la aplicación directa de los principios y teoremas de la mecánica clásica. Aunque estos métodos pueden resolver pequeños problemas cinemáticos y dinámicos, la complejidad de las ecuaciones involucradas limita su campo de aplicación. Una reflexión sobre los métodos computacionales de mecanismos vistos hasta ahora muestra que las herramientas disponibles son trigonometría básica, algunas consideraciones geométricas, leyes de Newton, cinemática de sólidos rígidos y resolución analítica de de ecuaciones diferenciales.

Con estos medios es posible abordar la mayor parte de los problemas que aparecen en el estudio clásico de la teoría de máquinas y mecanismos. Aunque el planteamiento de los problemas por métodos analíticos es directo, la resolución puede volverse muy compleja. También los métodos gráficos de análisis cinemático tienen serias limitaciones.

Generalmente, los métodos analíticos son los preferidos cuando los problemas son abordables, por su comodidad y por la facilidad con la que se interpretan los resultados.

Sin embargo, hoy en día, debido a la existencia de programas de ordenador con capacidad para resolver problemas de análisis cinemático y dinámico puede crear la tentación de centrarse en su utilización y prescindir de los planteamientos teóricos. Esta forma de pensar es incorrecta, ya que es necesario el conocimiento de las bases teóricas para proporcionar a los futuros ingenieros la capacidad de interpretar los resultados y de intuir posibles errores. Es por ello que, desde el punto de vista pedagógico, los métodos gráficos y analíticos siguen siendo muy útiles para la compresión de los problemas.

1.2 Objetivo

Con este proyecto se desea la creación de herramientas interactivas, que se puedan usar desde cualquier plataforma digital, donde los estudiantes puedan estudiar y reforzar los conocimientos adquiridos en las sesiones teóricas y prácticas de la asignatura "Teoría de Mecanismos".

Se diseñarán los mecanismos más didácticos (4 barras, biela-manivela corredera, biela-manivela con guía móvil, mecanismo de Retorno Rápido, mecanismo de Stephenson), y otros más complejos donde los alumnos podrán calcular los valores dinámicos de cualquier punto: posición, velocidad, aceleración, fuerzas y reacciones.

El objetivo de este proyecto es la innovación a la hora de explicar una asignatura tan visual como es Teoría de Máquinas y Mecanismos. Con la realización de este TFG el alumno podrá interactuar con el libro y descubrir de una manera más simple y a la vez más atractiva cómo funcionan estos elementos.

Para llevarlo a cabo es necesario implementar los mecanismos en Jupyter con todo el proceso que eso conlleva (cálculo de posiciones, velocidades, aceleraciones, matrices de restricción, Jacobiano, etc.) para luego realizar el gráfico y posteriormente la simulación.

En este trabajo se han relacionado diferentes materias de estudio dentro del grado como son: Cinemática de Máquinas, Métodos Numéricos y manejo de herramientas informáticas.

1.3 Contexto

En la Universidad de Almería se imparten cuatro grados pertenecientes al ámbito Industrial. Estos grados son Ingeniería Eléctrica, Ingeniería Química Industrial, Ingeniería Electrónica Industrial e Ingeniería Mecánica.

Estos grados, al ser todos del ámbito industrial, se imparten los dos primeros cursos de forma común. Es decir, todos los alumnos que cursen alguna de estas carreras tendrán las mismas asignaturas los dos primeros años.

El primer año de carrera contiene, en su mayoría, asignaturas de carácter básico (Física I y II, Matemáticas I y II, Química, etc). Estas asignaturas introducen al alumno en el campo de la ingeniería, pero de una forma muy generalizada lo que lleva al alumno a aprender conceptos teóricos complejos, pero aparentemente sin ninguna función práctica.

En el segundo curso de los grados mencionados es donde se empiezan a impartir las asignaturas de carácter obligatorio que muestran a los alumnos las aplicaciones reales de lo aprendido en el curso anterior. Sin embargo, el objetivo de mostrar a los alumnos algunas asignaturas que tienen un campo de estudio tan amplio en pocos meses a veces resulta complejo, tanto para el docente como para el alumno.

La asignatura de Teoría de Mecanismos es la primera asignatura del plan de estudios donde se adquieren los conocimientos básicos de cinemática y dinámica de máquinas. Tras cursar la asignatura el alumno debe ser capaz de analizar el comportamiento cinemático y dinámico de un mecanismo además del conocimiento sobre diseño y clasificación de engranajes.

En el año 2019 la (en su momento) alumna Beatriz Diaz Lozano realizó su trabajo fin de grado sobre el desarrollo de material de prácticas interactivo para la asignatura Teoría de Mecanismos usando Jupyter Notebook.

Este trabajo pretende ser una continuación del mencionado anteriormente, desarrollando nuevos mecanismos con el mismo enfoque y objetivo.

1.4 Planificación Temporal

Actividad	Duración (horas)
Estudio de análisis cinemático por métodos numéricos	20
Realización de los mecanismos propuestos fase teórica	75
Estudio del lenguaje Python y manual Jupyter Notebook	35
Instalación de la plataforma Anaconda	4
Desarrollo de los mecanismos propuestos en lenguaje Python	150
Elaboración del anteproyecto	25
Realización de la memoria del trabajo	50
Total	359

1.5 Competencias utilizadas en el TFG

Competencias generales:

- CB1: Poseer y comprender conocimientos procedentes de la vanguardia del campo de estudio puesto que se han desarrollado y resuelto problemas pertenecientes al campo de Teoría de Máquinas y de lenguaje informático.

- CB2: Aplicar los conocimientos al trabajo y poseer las competencias que suelen demostrarse por medio de la elaboración y defensa de argumentos y la resolución de problemas dentro del área de estudio, ya que se han resuelto los problemas propuesto con un objetivo práctico y de utilidad.

- CB3: Capacidad de reunir e interpretar datos relevantes para emitir juicios que incluyan una reflexión sobre temas relevantes de índole social, científica o ética. Esta capacidad se ha aplicado a la hora de interpretar los distintos resultados de los problemas cinemáticos propuestos.

- CB5: Desarrollo de habilidades de aprendizaje necesarias para emprender estudios posteriores con un alto grado de autonomía, ya que se han desarrollado herramientas nuevas en base a los campos de estudios del grado sin una base sólida en dicho campo.

Competencias transversales:

- UAL2: Utilizar las Técnicas de Información y Comunicación (TICs) como una herramienta para para el aprendizaje, la investigación y el trabajo cooperativo puesto que las herramientas desarrolladas han sido en el campo de la programación.

- UAL3: Capacidad para identificar, analizar, y definir los elementos significativos que constituyen un problema para resolverlo con rigor pues ha habido problemas en la mayoría de los intentos de resolución de los problemas cinemáticos y se han conseguido solventar con solidez.

Competencias específicas:

- CT3: Conocimiento en materias básicas y tecnológicas, que les capacite para el aprendizaje de nuevos métodos y teorías, y les dote de versatilidad para adaptarse a nuevas situaciones ya que el proyecto elaborado se basa en conocimientos de la carrera, pero desarrollando nuevas herramientas que aún no habían sido creadas.

- CB1: Capacidad para la resolución de los problemas matemáticos que puedan plantearse en la ingeniería pues han sido resueltos numerosos problemas cinemáticos por métodos numéricos.

- CB3: Conocimientos básicos sobre el uso y programación de los ordenadores, sistemas operativos, bases de datos y programas informáticos con aplicación en ingeniería puesto que se ha empleado el lenguaje Python para la elaboración de este trabajo.

- CRI7: Conocimiento de los principios de teoría de máquinas y mecanismos.

1.6 Estructura de la memoria del TFG

Esta memoria está dividida en siete capítulos u apartados:

- Introducción: Se comienza dando una perspectiva del contexto, el objetivo, la motivación y una breve descripción del trabajo de fin de grado elaborado.

- Revisión bibliográfica: Este el comienzo antes de empezar el propio proyecto. Se investiga sobre la forma de crear el trabajo, las herramientas más útiles y necesarias para lograr el objetivo propuesto.

- Material y métodos: El tercer apartado se centra en la explicación teórica del método implementado, los conceptos básicos sobre el análisis cinemático y la forma generalizada de resolver los problemas de posición, velocidad y aceleración.

- Resolución de los mecanismos: En el cuarto capítulo se aborda la resolución de los mecanismos propuestos en este trabajo, se plantean y desarrollan todas las ecuaciones y sistemas de forma teórica y posteriormente se implementan en código para proporcionar los resultados.

- Resultados: Aquí se muestran resumidamente los resultados numéricos y gráficos obtenidos de los problemas cinemáticos de los distintos mecanismos.

- Conclusiones: Breve reflexión sobre lo que ha significado este trabajo y el desarrollo del mismo además de su implicación real.

- Bibliografía: Referencias en las que se ha basado este trabajo.

2 REVISIÓN BIBLIOGRÁFICA

2.1 Anaconda, Jupyter Notebook y el lenguaje Python

2.1.1 ¿Qué es Anaconda?

Anaconda es una distribución de los lenguajes de programación Python y R para computación científica (ciencia de datos, aplicaciones de Machine Learning, procesamiento de datos a gran escala, análisis predictivo, etc.). (Rondón 2022)

Anaconda se agrupa en 4 sectores, Anaconda Navigator, Anaconda Project, Las librerías de Ciencia de datos y Conda.

Esta plataforma cuenta con una gran cantidad de características entre las que podemos destacar las siguientes:

- Libre y de código abierto.
- Multiplataforma (puede utilizarse en Linux, macOS y Windows).
- Permite instalar y administrar paquetes, dependencias y entornos para la ciencia de datos con Python de una manera muy sencilla.
- Cuenta con herramientas como Dask, numpy, pandas y Numba para analizar Datos.
- Permite visualizar datos con Matplotlib.
- Anaconda Navigator es una interfaz gráfica de usuario (GUI) bastante sencilla.
- Puede gestionar de manera avanzada paquetes relacionados a la Ciencia de datos con Python desde la terminal.
- Está equipado de herramientas que permiten crear y compartir documentos que contienen código con compilación en vivo, ecuaciones, descripciones y anotaciones.
- Los proyectos son portables, lo que permite compartir proyectos con otros y ejecutar proyectos en diferentes plataformas.

(Toro 2018)

2.1.1.1 Instalar Anaconda

Instalar Anaconda Distribution es bastante sencillo, solo es necesario ir a la sección de descarga de Anaconda Distribution y descargar la versión que se requiere (Python 3.6 o Python 2.7).

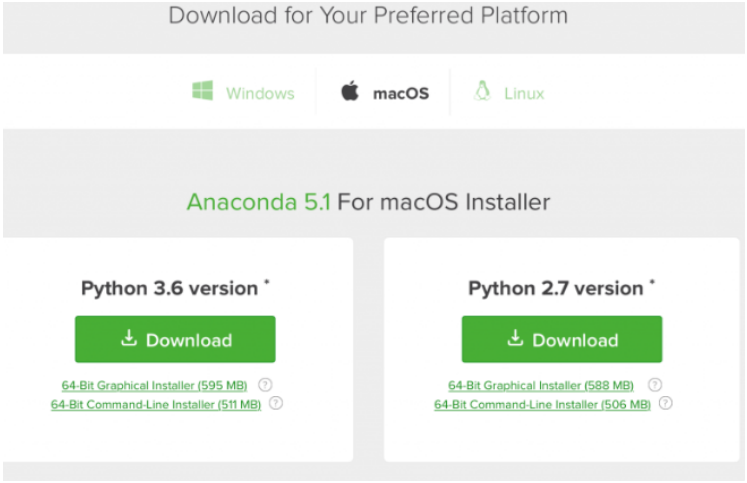


Figura 2.1. Elección versión de Anaconda

Una vez descargada se abre una terminal, en el directorio correspondiente ejecutamos el bash de instalación con la versión correspondiente.

Es un proceso muy intuitivo donde solo es necesario aceptar los términos de la licencia e instalar el programa.

Cuando la instalación finaliza se mostrará el siguiente entorno:

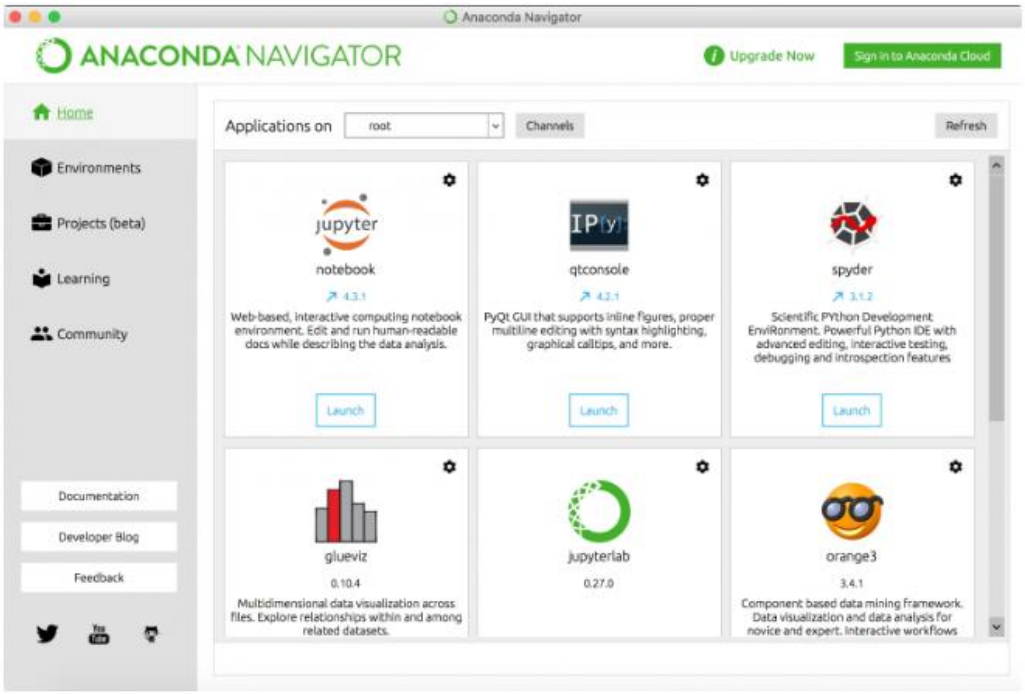


Figura 2.2. Entorno Anaconda Navigator

Con esto habrá concluido la instalación de Anaconda y se procederá a abrir la aplicación Jupyter Notebook.

2.1.2 Jupyter Notebook

Jupyter Notebooks es una aplicación web, también de código abierto que permite crear y compartir documentos con código en vivo, ecuaciones, visualizaciones y texto explicativo.

El entorno de las hojas de trabajo de Jupyter nos permite realizar un seguimiento a los errores y mantener el código limpio.

2.1.2.1 Acceder al entorno

En el apartado anterior se procedió a instalar anaconda, abrimos dicha plataforma desde el menú de nuestro ordenador y encontramos la aplicación Jupyter Notebook.

Se abre una ventana de comandos que lanza automáticamente la interfaz que está dividida en tres pestañas: "Files", Running" y "clusters". En la pestaña "Files" es donde podremos crear nuevos notebooks o abrir uno existente.

Figura 2.3. Interfaz Jupyter Notebook

Para crear un notebook no tenemos más que seleccionar la opción "New" que aparece en la esquina superior derecha. Una vez creado, se le asigna un nombre haciendo "click" sobre "Untitled".

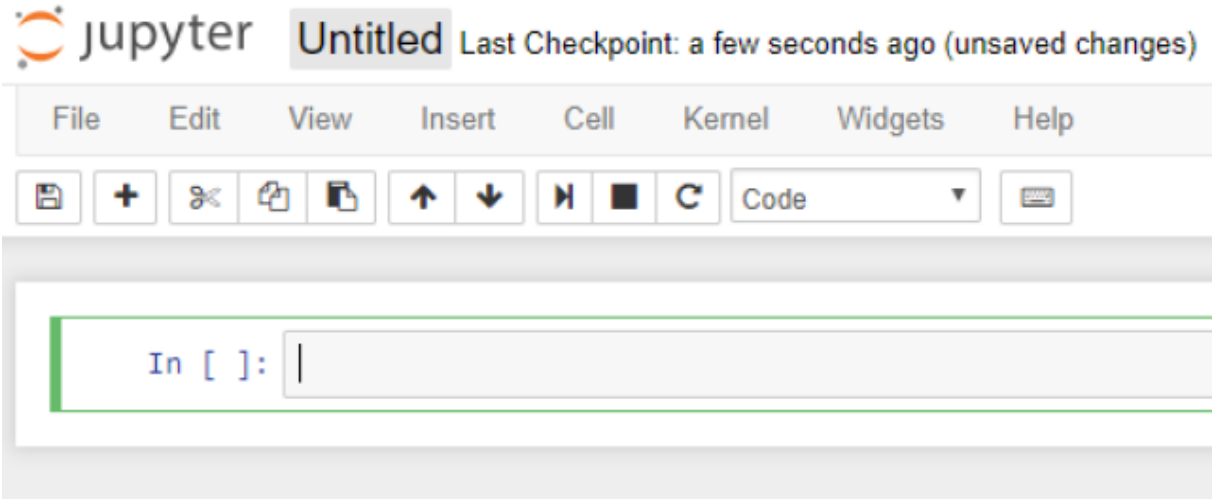


Figura 2.4. Notebook de Python

Se abre haciendo click sobre el nombre. El notebook consiste en una serie de celdas dentro de las cuales se puede escribir directamente el código.

Para ejecutar el código se elige la opción "Run Cells" dentro del menú. Otra de las ventajas que proporciona Jupyter Notebook es que da la posibilidad de crear "checkpoints" o puntos de referencia.

Cuando creas un checkpoint lo que haces en realidad es guardar el estatus del notebook en ese preciso instante de forma que puedas volver a ese punto concreto y deshacer los cambios que se hayan hecho después. Esto resulta de interés cuando se están realizando pruebas y algo no sale bien. Se puede volver sin problemas al punto dónde todo era correcto sin necesidad de empezar otra vez desde el principio.

Para crear un checkpoint, tan sólo hay que seleccionar la opción "Save and Checkpoint" desde el menú "File".

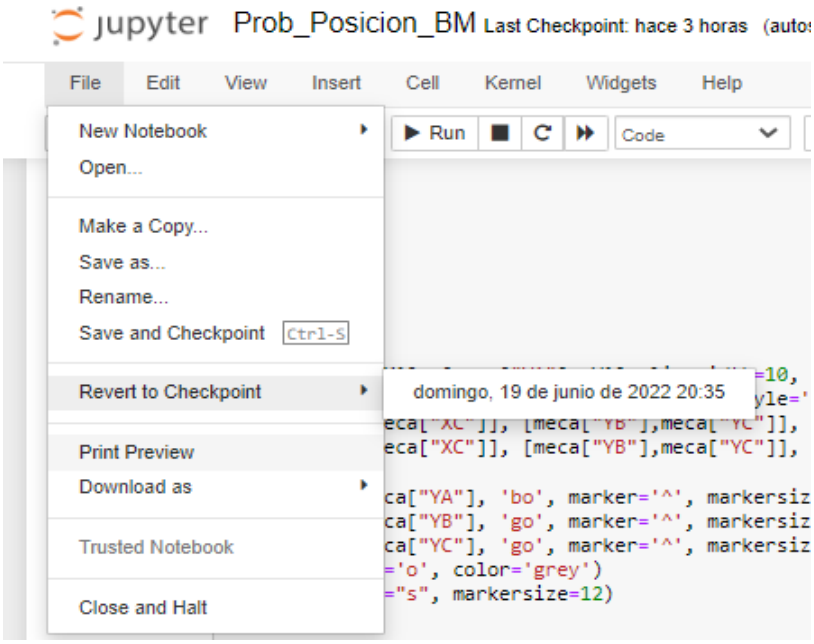


Figura 2.5. Menú del Notebook

Por último para exportar el notebook solo hay que seleccionar la opción "Download as" y elegir el formato que mas interese, podemos elegir descargarlo en notebook (.ipynb), Python (.py), html, pdf, etc. (Santos 2018)

2.1.3 Lenguaje Python

Python es un lenguaje de programación interpretado cuya filosofía hace hincapié en la legibilidad de su código. A diferencia de otros lenguajes como Java o .NET, se trata de un lenguaje interpretado, es decir, que no es necesario compilarlo para ejecutar las aplicaciones escritas en Python, sino que se ejecutan directamente por el ordenador utilizando un programa denominado intérprete.

Otra de sus características es la de ser un lenguaje multiparadigma, esto quiere decir que soporta distintos tipos de lenguaje de programación, como programación orientada a objetos, programación imperativa y, en menor medida, programación funcional.

Python es un lenguaje de programación multiplataforma, es decir, que los programas creados en este lenguaje pueden funcionar en cualquier sistema operativo, bien sea Windows, Apple o Linux, algo que permite desarrollar aplicaciones en cualquier sistema operativo con una facilidad asombrosa. Una gran cantidad de tecnologías se llevan muy bien con Python debido a su sencillez y a su gran potencia para el tratamiento de datos.

Debido a estas características, el uso de Python está muy extendido en muchos campos que resultan de interés:

- Análisis de datos y big data: El uso de Python está muy extendido en el análisis de datos y el big data. Su simplicidad y su gran número de bibliotecas de procesamiento de datos hacen que Python sea ideal a la hora de analizar y gestionar una gran cantidad de datos en tiempo real.
- Ciencia de datos: La razón por la que este lenguaje es ampliamente utilizado en data science es la misma que en el anterior; la sencillez y la potencia para trabajar con un gran número de datos, unidos al gran número de bibliotecas existentes, hacen que Python sea ideal para este tipo de tareas.

- Inteligencia artificial: Gran parte de los avances en inteligencia artificial (IA) se debe a Python. La capacidad de plasmar ideas complejas en pocas líneas, unidas al gran número de frameworks existentes, han hecho que Python sea uno de los lenguajes de programación que están impulsando a la IA.

- Juegos y gráficos 3D: Python también posee una gran capacidad para manejar gráficos 3D gracias la gran cantidad de marcos de trabajo y herramientas existentes. Numerosos juegos populares son escritos en este lenguaje ya que el motor gráfico, las animaciones y sus distintas funcionalidades pueden ser perfectamente desarrolladas con Python.

(Universidades 2021)

2.1.3.1 Numpy

Para el desarrollo de los distintos mecanismos el lenguaje Python dispone de librerías. Una librería es un conjunto de implementaciones funcionales, codificadas en un lenguaje de programación, que ofrece una interfaz bien definida para la funcionalidad que se requiere.

Las librerías son la forma que tiene Python de almacenar instrucciones o variables en un archivo para, posteriormente, utilizarlas en un script sin necesidad de definirlas cada vez que se requiera.

Una de estas librería es *Numpy*, una librería de Python especializada en el cálculo numérico y el análisis de datos. *Numpy* proporciona potentes estructuras de datos, incorpora una nueva clase de objetos llamados *arrays* que permite representar colecciones de datos de un mismo tipo en varias dimensiones, y funciones muy eficientes para su manipulación.

Un *array* es una estructura de datos de un mismo tipo organizada en forma de tabla o cuadrícula de distintas dimensiones (vectores y matrices).

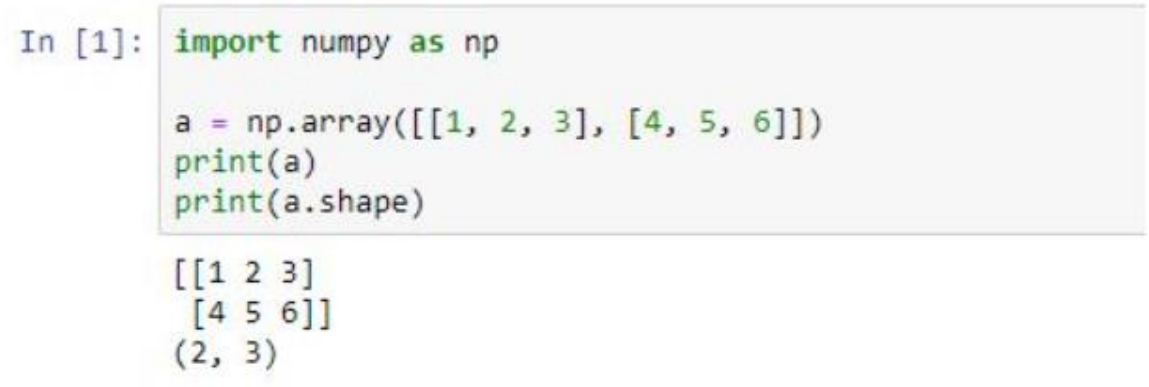


Figura 2.6. Ejemplo de matriz utilizando librería Numpy

2.1.3.2 Matplotlib

Matplotlib es una librería para generar gráficas a partir de datos contenidos en listas, vectores, en el lenguaje de programación Python y en su extensión matemática *NumPy*.

Los mecanismos se componen de eslabones, pares cinemáticos, puntos fijos y móviles y múltiples elementos que son necesarios identificar para la correcta comprensión de esta asignatura.

Para cumplir este objetivo se ha necesitado de la ayuda de la librería Matplotlib, esta librería posee una web donde detalla todo el código relacionado con la creación de gráficas y animaciones en Python.

En la librería de Matplotlib se pueden elaborar desde gráficas básicas hasta animaciones en 3D y, además, cuenta con tutoriales y ejemplos para desarrollar los distintos gráficos.

Figura 2.7. Ejemplos de gráfico con Matplotlib

Para el diseño que interesa en este trabajo nos guiamos de las siguientes herramientas que nos proporciona esta librería:

- Marker reference: Este subapartado de la librería nos muestra como cambiar el diseño de los puntos. De esta manera podemos cambiar la forma y el color de los puntos del mecanismo y así poder diferencia entre puntos fijos y móviles. Dependiendo de nuestros intereses podemos diseñarlo de las siguientes maneras.

```
import matplotlib.pyplot as plt
from matplotlib.lines import Line2D


text_style = dict(horizontalalignment='right', verticalalignment='center',
                  fontsize=12, fontfamily='monospace')
marker_style = dict(linestyle=':', color='0.8', markersize=10,
                    markerfacecolor="tab:blue", markeredgecolor="tab:blue")


def format_axes(ax):
    ax.margins(0.2)
    ax.set_axis_off()
    ax.invert_yaxis()


def split_list(a_list):
    i_half = len(a_list) // 2
    return a_list[:i_half], a_list[i_half:]
```

Figura 2. 8. Diseño de puntos con marker reference

```
fig, axs = plt.subplots(ncols=2)
fig.suptitle('Filled markers', fontsize=14)
for ax, markers in zip(axs, split_list(Line2D.filled_markers)):
    for y, marker in enumerate(markers):
        ax.text(-0.5, y, repr(marker), **text_style)
        ax.plot([y] * 3, marker=marker, **marker_style)
    format_axes(ax)

plt.show()
```

Figura 2.8. Ejemplo de código para crear puntos

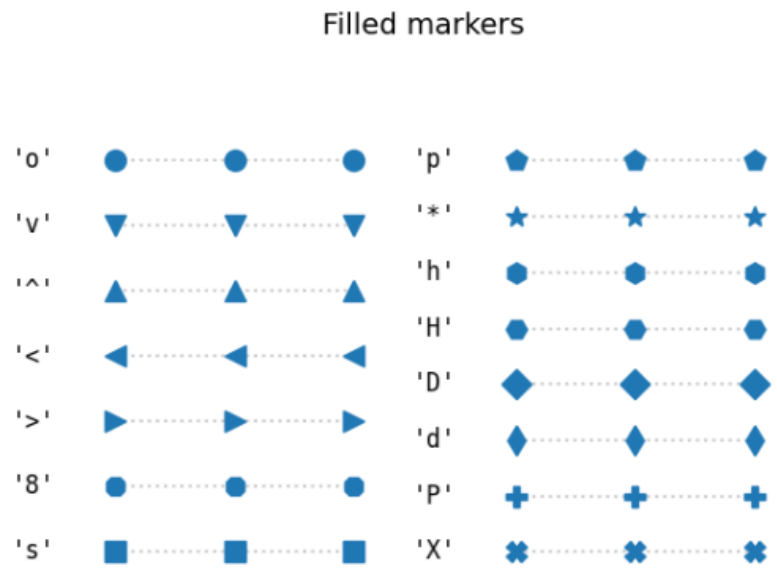


Figura 2.9. Puntos creados con marker reference

- Linestyles: Para cambiar el diseño de las líneas tenemos la herramienta linestyle donde nos muestra los distintos tipos de líneas y el código para implementarlas.

```
def plot_linestyles(ax, linestyles, title):
    X, Y = np.linspace(0, 100, 10), np.zeros(10)
    yticklabels = []

    for i, (name, linestyle) in enumerate(linestyles):
        ax.plot(X, Y+i, linestyle=linestyle, linewidth=1.5, color='black')
        yticklabels.append(name)
```

Figura 2.10. Ejemplo de código para diseño de líneas

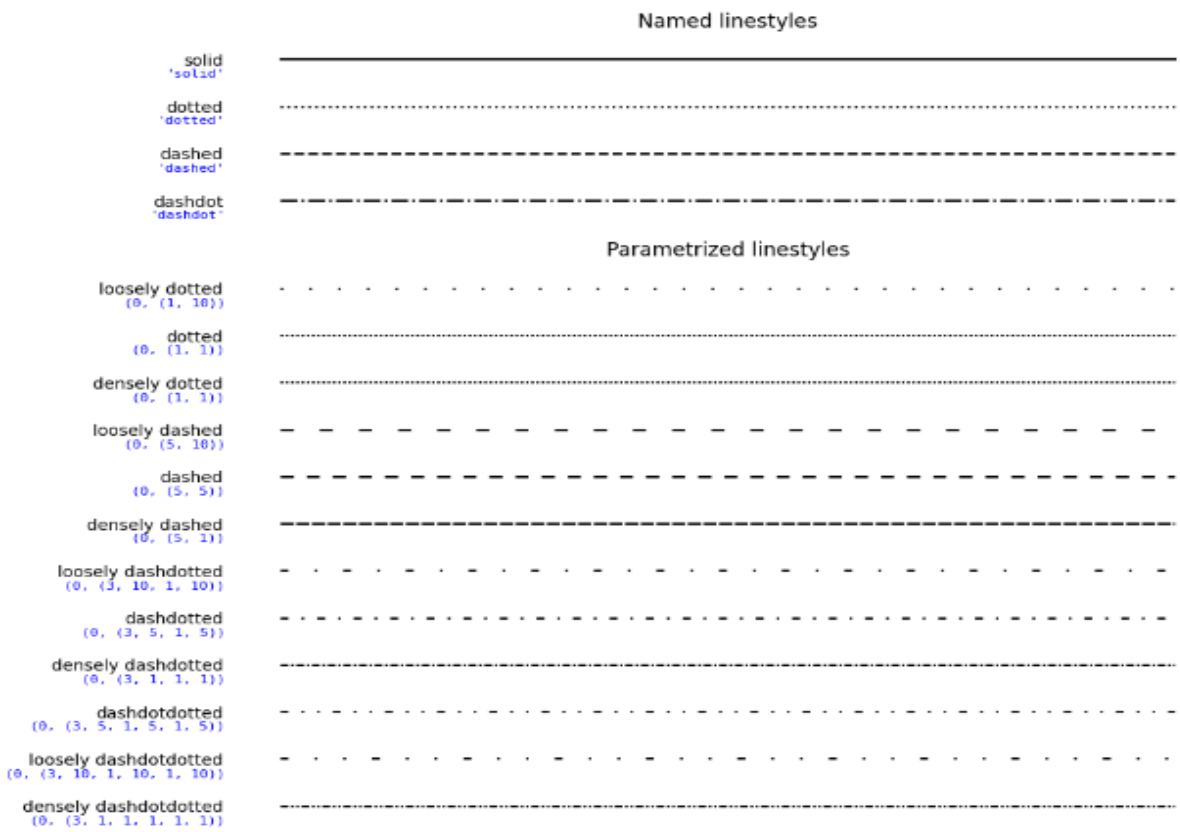


Figura 2.11. Tipos de líneas que se pueden crear con Linestyle

Además de estas, tenemos infinidad de herramientas que nos ayudan a modificar el diseño de las gráficas según nuestros requerimientos, todas ellas explicadas en la web de Matplotlib.

Otra biblioteca de gran utilidad dentro del paquete Matplotlib es animation. Con esta librería se pueden crear simulaciones y guardarlas como una animación HTML.

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation


def data_gen():
    for cnt in itertools.count():
        t = cnt / 10
        yield t, np.sin(2*np.pi*t) * np.exp(-t/10.)


def init():
    ax.set_ylim(-1.1, 1.1)
    ax.set_xlim(0, 10)
    del xdata[:]
    del ydata[:]
    line.set_data(xdata, ydata)
    return line,

fig, ax = plt.subplots()
line, = ax.plot([], [], lw=2)
ax.grid()
xdata, ydata = [], []


def run(data):
    # update the data
    t, y = data
    xdata.append(t)
    ydata.append(y)
    xmin, xmax = ax.get_xlim()

    if t >= xmax:
        ax.set_xlim(xmin, 2*xmax)
        ax.figure.canvas.draw()
    line.set_data(xdata, ydata)

    return line,

ani = animation.FuncAnimation(fig, run, data_gen, interval=10, init_func=init)
plt.show()
```

Figura 2.12. Ejemplo código de animación

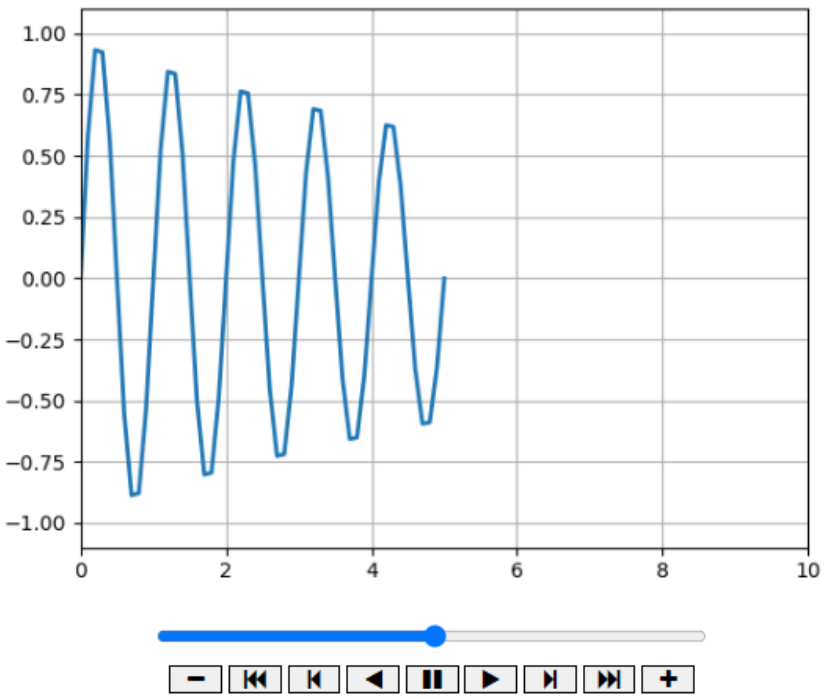


Figura 2.13. Ejemplo animación con matplotlib

3 MATERIALES Y MÉTODOS

3.1 Análisis cinemático por métodos numéricos

3.1.1 Razones para usar métodos numéricos

- Método analítico: Solo sirven para una única posición del mecanismo, es tedioso y más susceptible a errores y además existen cadenas cinemáticas no resolubles por métodos analíticos/gráficos. La ventaja que poseen respecto a los métodos numéricos es que la mayor intuición física en cada paso.
- Métodos numéricos: Programación en ordenador, resolución para todas las posiciones. Métodos genéricos, 100 % de los mecanismos. Riesgo de perder el sentido físico/mecánico.

3.1.2 Modelado

Se debe seleccionar un conjunto de coordenadas para describir un mecanismo. Estas deben describir de manera unívoca la posición, velocidad y aceleración del mecanismo en todo momento. (Rincón s.f.)

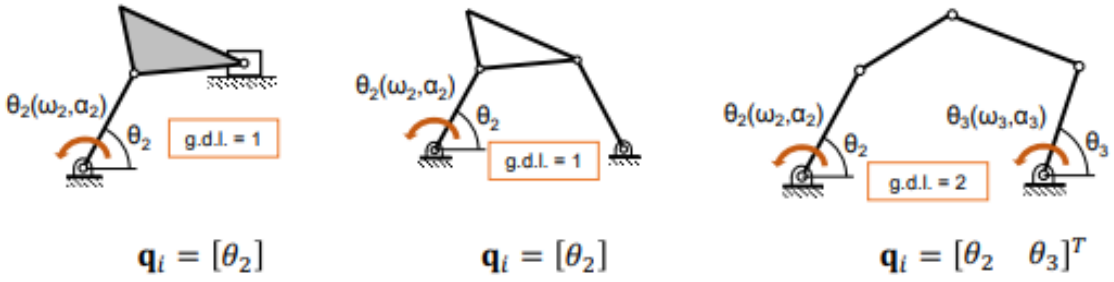


Figura 3.1. Ejemplos de mecanismos y sus coordenadas independientes

El número mínimo de coordenadas es igual a los g.d.l. (grados de libertad), estas coordenadas se conocen como coordenadas independientes. Sin embargo, no siempre definen al sistema de manera única ya que las coordenadas independiente s varían con el tiempo.

$$\Phi(t) = \begin{bmatrix} \theta_2(t) \\ \theta_3(t) \\ \theta_4(t) \end{bmatrix} = 0 \qquad (3.1)$$

Las coordenadas adicionales que se utilizan para definir completamente al sistema se conocen como coordenadas dependientes.

$$\left.\begin{matrix}\mathbf{q}_i = [\theta_2 \quad \theta_3 \quad \theta_4]^T \\ \mathbf{q}_d = [\theta_5 \quad \theta_6]^T\end{matrix}\right\} \rightarrow \mathbf{q} = \begin{Bmatrix}\mathbf{q}_i \\ \mathbf{q}_d\end{Bmatrix} \qquad (3.2)$$

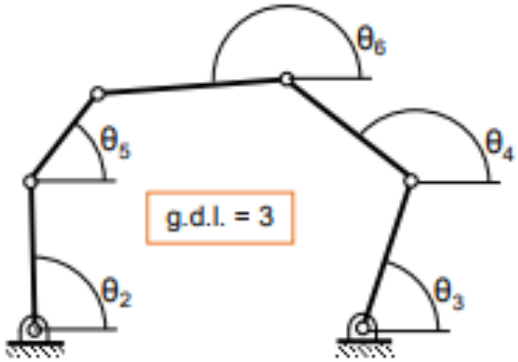


Figura 3.2. Ejemplo de mecanismo con coordenadas dependientes e independientes

El mismo mecanismo se puede definir de varias maneras posibles, dependiendo del objetivo de su análisis, dicho objetivo determinará la eficacia y simplicidad del problema. Los tipos de coordenadas que más se utilizan para definir a un mecanismo en métodos numéricos son:

· Coordenadas relativas.

· Coordenadas de punto referencia.

· Coordenadas naturales.

Coordenadas relativas: Las coordenadas relativas definen la posición de cada elemento respecto del anterior en la cadena cinemática, utilizando los parámetros correspondientes a los grados de libertad relativos permitidos en la cadena.

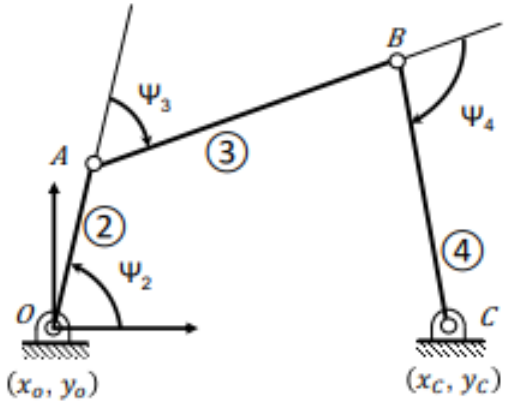


Figura 3. 3. Ejemplo mecanismo con coordenadas relativas

1 g. d. l. ⇒ 1 coordenada independiente ⇒ 1 restricción conductora

2 coordenadas dependientes ⇒ 2 restricciones cinemáticas

$$\mathbf{q}_i = [\psi_2]^T$$
$$\mathbf{q}_d = [\psi_3 \quad \psi_4]^T \tag{3.3}$$

$$\mathbf{\Phi}_d = \begin{Bmatrix} L_2\cos(\psi_2) + L_3\cos(\psi_2+\psi_3) + L_4\cos(\psi_2+\psi_3+\psi_4) - \mathbf{OC} \\ L_2\sin(\psi_2) + L_3\sin(\psi_2+\psi_3) + L_4\sin(\psi_2+\psi_3+\psi_4) \end{Bmatrix} = \mathbf{0} \tag{3.4}$$

Estas coordenadas tienen como ventaja el uso de pocas variables y, además, usadas en problemas de dinámica proporcionan información relevante (momentos, reacciones, etc.).

Coordenadas de punto de referencia: En cinemática plana la posición global de un elemento se define con tres coordenadas. Se determinan las componentes cartesianas de un punto de referencia (normalmente el centro de masas) y la orientación del cuerpo.

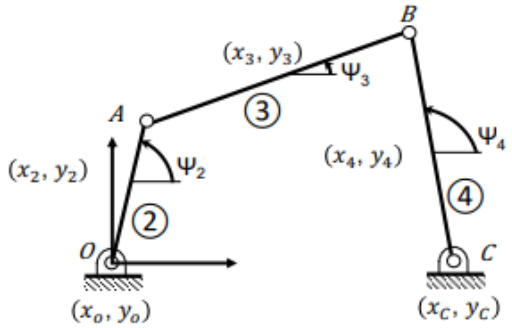


Figura 3. 4. Ejemplo mecanismo con coordenadas de punto referencia

1 g. d. l. ⇒ 1 coordenada independiente ⇒ 1 restricción conductora

8 coordenadas dependientes ⇒ 8 restricciones cinemáticas

$$\mathbf{q}_i = [\psi_2]^T$$
$$\mathbf{q}_d = [x_2 \quad y_2 \quad x_3 \quad y_3 \quad \psi_3 \quad x_4 \quad y_4 \quad \psi_4]^T \tag{3.5}$$

$$\mathbf{\Phi}_d = \begin{Bmatrix} (x_2 - x_0) - {}^{L_2}/_2\cos(\psi_2) \\ (y_2 - y_0) - {}^{L_2}/_2\sin(\psi_2) \\ (x_3 - x_2) - {}^{L_2}/_2\cos(\psi_2) - {}^{L_3}/_2\cos(\psi_3) \\ (y_3 - y_2) - {}^{L_2}/_2\sin(\psi_2) - {}^{L_3}/_2\sin(\psi_3) \\ (x_4 - x_3) - {}^{L_3}/_2\cos(\psi_3) - {}^{L_4}/_2\cos(\psi_4) \\ (y_4 - y_3) - {}^{L_3}/_2\sin(\psi_3) - {}^{L_4}/_2\sin(\psi_4) \\ (x_4 - x_C) - {}^{L_4}/_2\cos(\psi_4) \\ (y_4 - y_C) - {}^{L_4}/_2\sin(\psi_4) \end{Bmatrix} = \mathbf{0} \tag{3.6}$$

Como ventaja, el uso de estas coordenadas nos proporciona un método muy sistemático. Sin embargo, como se observa, requiere de muchas restricciones debido al uso de un gran número de coordenadas.

Coordenadas naturales: Este tipo de coordenadas fueron propuestas por el investigador español Javier García de Jalón en 1979, mientras trabajaba en la Universidad de Navarra. Son una evolución de las coordenadas de punto de referencia, en las que dichos puntos se trasladan a los pares cinemáticos. Así, cada elemento tiene al menos dos puntos de ref. Tanto la posición como la orientación del elemento se definen mediante coordenadas cartesianas, por lo que las expresiones angulares no son necesarias. (Fernandez, Moreno y Claraco 2017)

Las ecuaciones de restricción con coordenadas naturales son de dos tipos:

· De sólido rígido: Una ecuación por cada cuerpo.

· De par cinemático: Una ecuación por par.

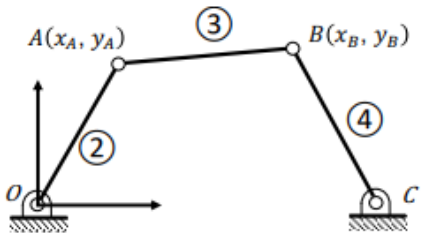


Figura 3. 5. Ejemplo mecanismo con coordenadas naturales

1 g. d. l. ⇒ 1 coordenada independiente ⇒ 1 restricción conductora

3 coordenadas dependientes ⇒ 3 restricciones cinemáticas

$$\mathbf{q}_i = [x_A]^T$$
$$\mathbf{q}_d = [x_B \quad y_B \quad y_C]^T \tag{3.7}$$

$$\mathbf{\Phi}_d = \begin{Bmatrix} (x_O - x_A)^2 + (y_O - y_A)^2 - L_2^2 \\ (x_A - x_B)^2 + (y_A - y_B)^2 - L_3^2 \\ (x_C - x_B)^2 + (y_C - y_B)^2 - L_4^2 \end{Bmatrix} = \mathbf{0} \tag{3.8}$$

Coordenadas mixtas: En la práctica es habitual utilizar una combinación de los tipos de coordenadas explicado anteriormente ya que nada nos impide mezclar coordenadas de distintos tipos, aún siendo redundantes, en un vector de coordenadas generalizadas q.

Lo más habitual es usar coordenadas naturales por sus múltiples ventajas y las coordenadas relativas por su sentido mecánico.

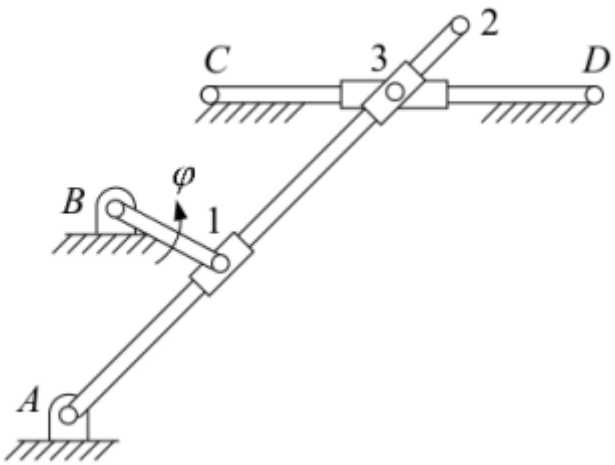


Figura 3. 6. Ejemplo mecanismo con coordenadas mixtas

En este caso usamos las coordenadas naturales para los sólidos rígidos y además añadimos la coordenada φ.

$$q = \begin{bmatrix} x1 \\ y1 \\ x2 \\ y2 \\ \mathrm{x3} \\ y3 \\ \rho \end{bmatrix} \qquad (3.9)$$

Al añadir 1 coordenada extra es necesario añadir 1 ecuación de restricción. Como la coordenada añadida es la del ángulo que forma la barra que une el punto B con el punto 1, vale cualquiera de estas dos ecuaciones:

$$(X_1 - X_B) - L_2 * cos\rho = 0 \qquad (3.10)$$

$$(Y_1 - Y_B) - L_2 * sin\rho = 0 \qquad (3.11)$$

3.1.3 Problema de posición

La posición de un mecanismo se determina resolviendo el sistema de ecuaciones de restricción, que es no lineal. El objetivo es obtener el resto de las coordenadas del vector *q* a partir de los g.d.l. conocidos, de manera compacta se puede expresar de la siguiente forma:

$$\Phi(q(t), t) = 0 \qquad (3.12)$$

Donde *t* es el tiempo, *q(t)* son las coordenadas generalizadas y Φ es la matriz de restricciones que agrupa a las ecuaciones de restricción temporales y cinemáticas.

Las incógnitas son las coordenadas generalizadas *q(t)*, que expresan la posición del mecanismo en el instante inicial. Sin embargo, al ser un problema no lineal, no se puede resolver directamente, por tanto, se necesita partir de un valor numérico para las coordenadas y utilizar un método iterativo para resolver sistemas de ecuaciones no lineales, como el de Newton-Raphson.

El método de aproximación iterativa Newton-Raphson se basa en definir una posición inicial para el mecanismo y a partir de dicha posición inicial realizar iteraciones hasta que la matriz de restricciones se aproxime a 0.

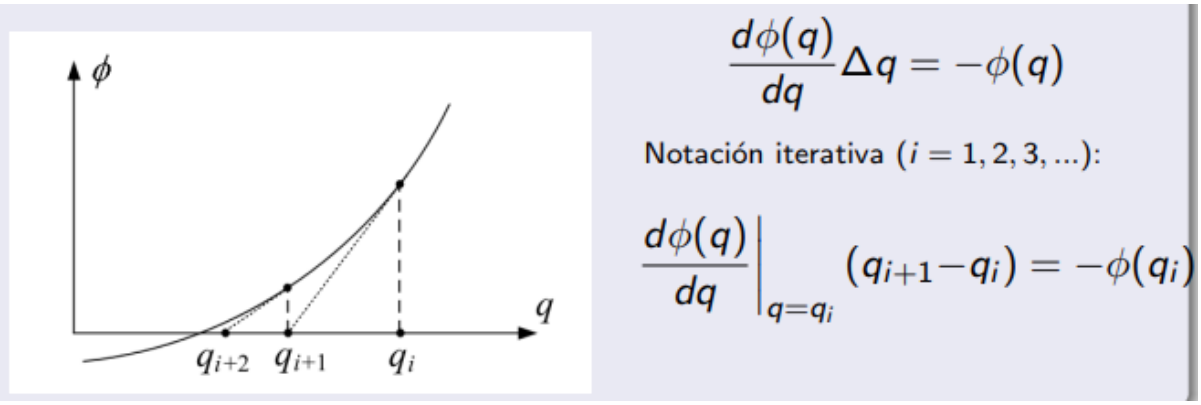


Figura 3. 7. Método Newton-Raphson problema de posición

$$\frac{d\phi(q)}{dq}\Delta q = -\phi(q)$$

$$\downarrow$$

$$\frac{\partial \mathbf{\Phi}(\mathbf{q}, t)}{\partial \mathbf{q}}(\Delta \mathbf{q}) = -\mathbf{\Phi}(\mathbf{q}, t)$$

Figura 3. 8. Fórmula problema de posición

A esta matriz de derivadas parciales se le conoce como Jacobiano y lo nombraremos como Φ_q.

$$\mathbf{\Phi_q} = \begin{bmatrix} \frac{\partial \phi_1}{\partial q_1} & \frac{\partial \phi_1}{\partial q_2} & \cdots & \frac{\partial \phi_1}{\partial q_n} \\ \frac{\partial \phi_2}{\partial q_1} & \frac{\partial \phi_2}{\partial q_2} & \cdots & \frac{\partial \phi_2}{\partial q_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial \phi_n}{\partial q_1} & \frac{\partial \phi_n}{\partial q_2} & \cdots & \frac{\partial \phi_n}{\partial q_n} \end{bmatrix} \quad (3.13)$$

La fórmula aplicada para la resolución del problema de posición queda de la siguiente manera:

$$\Phi(q + \Delta q, t) = \Phi(q, t) + \Phi_q(q, t) * \Delta q = 0 \quad (3.14)$$

Despejando esta ecuación queda de la siguiente manera:

$$\Phi_q * \Delta q = -\Phi \quad (3.15)$$

Como no es posible dividir el jacobiano respecto a su matriz de restricciones directamente lo que se debe hacer es multiplicar por la inversa del jacobiano a la izquierda ambos términos:

$$\Phi_q{}^{-1} * \Phi_q * \Delta q = \Phi_q{}^{-1} * -\Phi \qquad (3.16)$$

Cuando se multiplica una matriz por su inversa queda la matriz identidad por lo que simplificando obtenemos:

$$\Delta q = \Phi_q{}^{-1} * -\Phi \qquad (3.17)$$

Este valor de Δq se lo añadimos al antiguo valor de q para obtener un nuevo valor de q:

$$q_1 = q_0 + \Delta q \qquad (3.18)$$

Con este nuevo valor de q volvemos a operar, y así sucesivamente hasta que nuestras iteraciones nos lleven a que Φ se aproxime a 0.

Sin embargo, aún hay un problema que debemos resolver ya que el sistema de la ecuación (3.15) es indeterminado ya que:

$$\left[\ \mathbf{\Phi_q}\ \right]_{m\times n}\left[\mathbf{\Delta q}\right]_{n\times 1} = \left[-\mathbf{\Phi}(\mathbf{q},t)\right]_{m\times 1}$$

Como $m < n$, el rango de Φ_q será menor de n→ Sistema indeterminado.

Para solucionar este problema debemos añadir una nueva ecuación al sistema lineal por cada g.d.l. obligando a que una de las variables dependientes del vector *q* tenga un valor determinado.

Por ejemplo, si tenemos un mecanismo con 2 g.d.l.:

$$\left(\begin{array}{ccc} & \mathbf{\Phi_q} & \\ \hline 0\cdots & 1 & 0 \\ 0\cdots & 0 & 1 \end{array}\right)_{(m+g)\times n}\left[\mathbf{\Delta q}\right]_{n\times 1} = \left[\begin{array}{c} \text{-}\mathbf{\Phi}(\mathbf{q},t) \\ \hline 0 \\ 0 \end{array}\right]_{(m+g)\times 1}$$

De esta manera, añadimos unos valores fijos que permiten que el sistema se pueda resolver adecuadamente.

3.1.4 Problema de velocidad

Para obtener la ecuación de velocidad, se deriva la de posición aplicando la regla de la cadena, ya que la función depende de dos variables:

$$\dot{\Phi}(q(t),t) = \frac{d\Phi(q(t),t)}{dt} = \frac{\partial\Phi(q(t),t)}{\partial q} * \frac{\partial q}{\partial t} + \frac{\partial\Phi(q(t),t)}{\partial t} = \Phi_q * \dot{q} + \Phi_t = 0 \qquad (3.19)$$

Despejando queda:

$$\Phi_q * \dot{q} = -\Phi_t \qquad (3.20)$$

Donde Φ_q es la matriz Jacobiana explicada anteriormente y Φ_t la derivada de la matriz de restricciones respecto del tiempo.

$$\mathbf{\Phi}_t = \begin{bmatrix} \frac{\partial \phi_1}{\partial t} \\ \frac{\partial \phi_2}{\partial t} \\ \vdots \\ \frac{\partial \phi_n}{\partial t} \end{bmatrix} \tag{3.21}$$

Para resolver el problema de velocidad se debe despejar el vector $\dot{q}$, ya que es el vector que contiene las velocidades generalizadas. En este caso el problema de velocidad si es lineal.

3.1.5 Problema de aceleración

Para obtener la ecuación de aceleración, se deriva la de velocidad respecto del tiempo:

$$\ddot{\Phi}(q(t),t) = \frac{d\dot{\Phi}(q(t),t)}{dt} = \frac{\partial \dot{\Phi}(q(t),t)}{\partial q}\frac{\partial q}{\partial t} + \frac{\partial \dot{\Phi}(q(t),t)}{\partial t} = \Phi_q * \ddot{q} + \dot{\Phi}_q * \dot{q} + \dot{\Phi}_t = 0 \tag{3.22}$$

Dejando el sistema en función de las aceleraciones queda:

$$\Phi_q * \ddot{q} = -\dot{\Phi}_q * \dot{q} - \dot{\Phi}_t \tag{3.23}$$

3.2 Implementación en código

Para desarrollar un modelo matemático programable en un ordenador es necesario, en primer lugar, crear un modelo matemático simple y eficiente del mecanismo. Esto implica transformar los conceptos de elemento, par cinemático, velocidad, etc., en un conjunto de datos numéricos dispuestos en forma de matriz o vector: es el proceso de modelización (Alejo Avello, 2018).

En este trabajo fin de grado nos centraremos en la resolución de los problemas de posición, velocidad y aceleración de distintos mecanismos para su implementación en la parte docente de la asignatura Teoría de Mecanismos.

3.2.1 Análisis cinemático

El algoritmo de análisis cinemático resuelve los problemas de posición, velocidad y aceleración en un numero de posiciones de análisis definidas por el usuario. Este número de posiciones deben ser lo suficientemente grandes como para proporcionar información que permita evaluar correctamente el comportamiento del sistema sin suponer un elevado coste computacional.

La información proporcionada por el algoritmo de análisis cinemático resulta de gran utilidad para comprender el comportamiento del sistema, y se utiliza después del proceso de síntesis dimensional para verificar en qué medida el mecanismo solución satisface las expectativas del diseñador.

La secuencia de pasos que sigue el algoritmo de análisis cinemático es la siguiente:

1. Lectura de datos:
   - Numero de posiciones de análisis.
   - Nudos origen y final de barra.
   - Vector inicial introducido por el usuario
   - Longitudes iniciales de barra introducidas por el usuario
   - Parámetros cinemáticos del grado de libertad.
   - Restricciones geométricas: colinealidad, no inversión del triángulo, coordenada fija de punto.

2. Análisis cinemático. El algoritmo abre un bucle donde se resuelven, de manera secuencial, los problemas de posición, velocidad y aceleración, para cada una de las posiciones de análisis definidas por el usuario. La secuencia de pasos que se sigue en cada iteración es la siguiente:
   - Lectura del vector inicial.
   - Cálculo de las coordenadas de posiciones de los grados de libertad del sistema.
   - Resolución del problema de posición.
   - Se almacena el vector posición resultante como vector inicial del próximo punto de análisis.
   - Resolución del problema de velocidad.
   - Resolución del problema de aceleración.

3. Escritura de archivos de resultados.

4 RESOLUCIÓN DE LOS MECANISMOS

4.1 Biela-manivela con guía móvil

El mecanismo biela-manivela con guía móvil es una versión más simple del mecanismo de Retorno Rápido, en las imágenes posteriores vemos un ejemplo de como funciona este mecanismo, sin embargo, en las primeras la parte que nos interesa es la de la unión de la barra amarilla con la verde y su deslizadera, pues esa es la parte de biela-manivela con guía móvil.

Este mecanismo es muy utilizado en la industria y su uso comercial está principalmente ligado a las herramientas limadoras y cepilladoras.

El movimiento se obtiene con un motor de velocidad constante que mueve la manivela de entrada, en la salida se tiene el oscilador que permite transmitir una velocidad de trabajo que se desea constante y un retorno a mayor velocidad, el cual se logra en el menor tiempo posible para que la carrera de trabajo se inicie de nuevo.

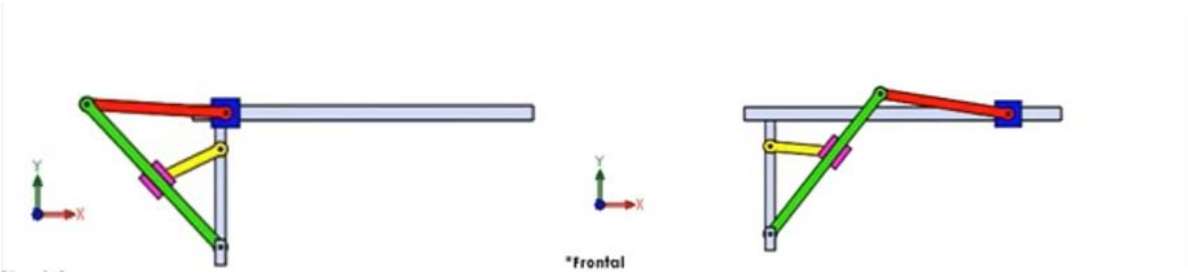


Figura 4. 1. Mecanismo Cepillo Manivela

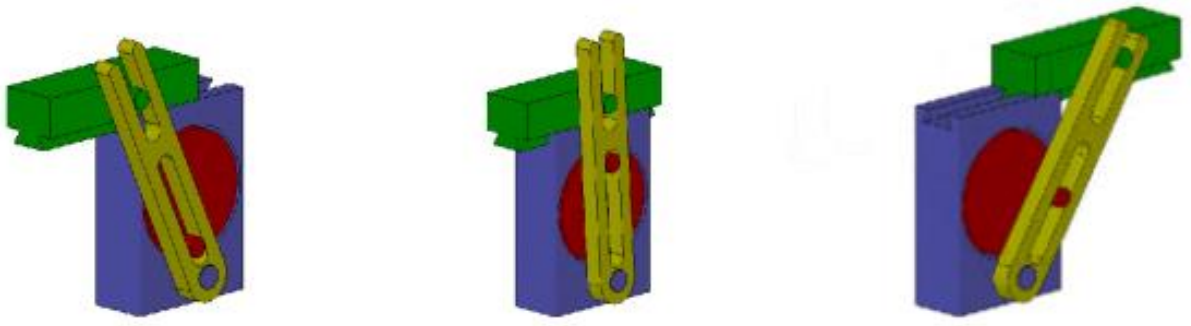


Figura 4. 2. Limadora Cepilladora

Mientras en el mecanismo biela-manivela corredera el eslabón fijo se conecta con una articulación a un eslabón y con una deslizadera a otro, en este mecanismo (biela-manivela con guía móvil), el eslabón fijo se conecta con dos articulaciones a sendos eslabones móviles del mecanismo.

4.1.1 Cálculo Teórico

4.1.1.1 Problema de posición

Paso 1. Modelado del mecanismo

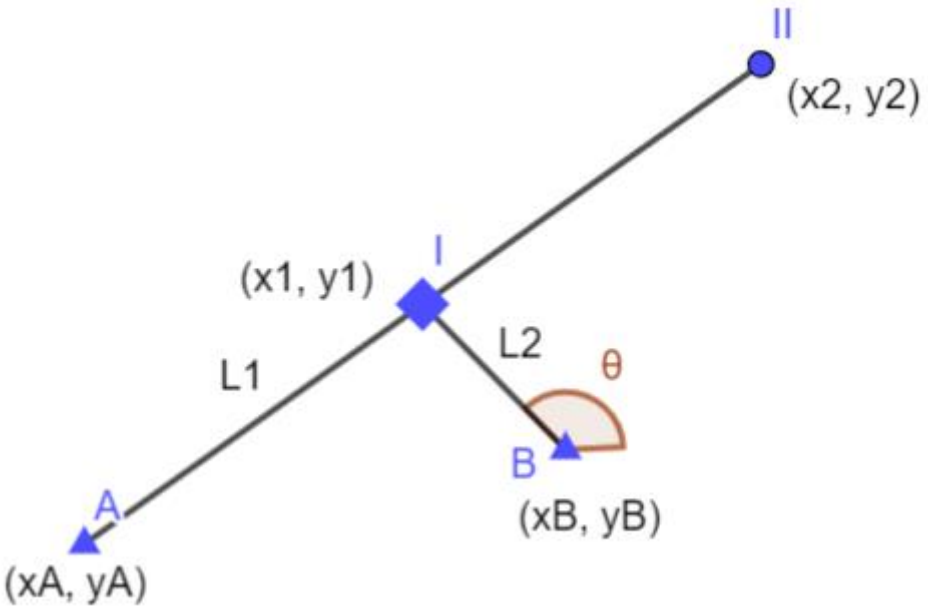


Figura 4. 3. Modelado mecanismo biela-manivela con guía móvil

Paso 2. Grados de libertad

Los grados de libertad se calculan a partir de la siguiente fórmula:

$$G = 3 * (N - 1) - 2P_I - P_{II} = 3 * (3 - 1) - 2 * 2 - 1 = 1 \qquad (4.0)$$

P_I→ Número de pares binarios de un grado de libertad

P_{II}→ Número de pares binarios de dos grados de libertad

Paso 3. Definición del vector q

Las coordenadas implementadas en este mecanismo son las que sabemos que varían con el tiempo, en este caso hemos usado coordenadas naturales y hemos tomado como variable independiente en ángulo que forma la barra L2 con el punto B y el punto 1 (x1, y1, x2, y2, θ).

$$q = \begin{bmatrix} x1 \\ y1 \\ x2 \\ y2 \\ \theta \end{bmatrix} \quad (4.1)$$

Para implementar este vector de coordenadas se procede a insertar los parámetros iniciales que definen al mecanismo, estos valores son fijos y no cambian con el tiempo. Los datos de prueba han sido los siguientes:

$$L1 = 3$$

$$L2 = 1$$

$$\theta = 0.5$$

$$X_B = 1$$

$$Y_B = 1$$

$$X_A = 0$$

$$Y_A = 0$$

Paso 4. Matriz de restricciones

Para definir la matriz de restricciones aplicamos las siguientes ecuaciones de restricción:

- Sólido A-2:

$$(X_2 - X_A)^2 + (Y_2 - Y_A)^2 - {L_1}^2 = 0 \quad (4.2)$$

- Sólido B-1:

$$(X_1 - X_B)^2 + (Y_1 - Y_B)^2 - {L_2}^2 = 0 \quad (4.3)$$

- Par prismático entre el punto 1 y la barra A-2:

$$X_1 * Y_2 - X_2 * Y_1 = 0 \quad (4.4)$$

- Coordenada relativa θ:

$$(X_1 - X_B) - L_2 * cos\theta = 0 \quad (4.5)$$
$$(Y_1 - Y_B) - L_2 * sin\theta = 0 \quad (4.5)$$

Con estas restricciones se forma la matriz de restricciones:

$$\Phi = \begin{bmatrix} (X_2 - X_A)^2 + (Y_2 - Y_A)^2 - {L_1}^2 \\ (X_1 - X_B)^2 + (Y_1 - Y_B)^2 - {L_2}^2 \\ X_1 * Y_2 - X_2 * Y_1 \\ (X_1 - X_B) - L_2 * cos\theta \end{bmatrix} \quad (4.6)$$

Como el punto (xA, yA) lo hemos considerado como nuestro origen de coordenadas podemos simplificar la primera ecuación de la matriz para que quede tal que:

$$\Phi = \begin{bmatrix} {X_2}^2 + {Y_2}^2 - {L_1}^2 \\ (X_1 - X_B)^2 + (Y_1 - Y_B)^2 - {L_2}^2 \\ X_1 * Y_2 - X_2 * Y_1 \\ (X_1 - X_B) - L_2 * cos\theta \end{bmatrix} \quad (4.7)$$

A esta matriz, además, debemos añadirle una condición para la última ecuación, ya que cuando un ángulo tiende a 0, su seno también lo hace, por lo que para esos casos utilizaríamos la restricción del coseno. En cambio, cuando el ángulo tiende más a 90 grados, es el coseno el que se aproxima a 0, por lo que en esos casos la restricción a utilizar sería la del seno. Nuestra matriz de restricciones con esta condición queda de la siguiente manera:

-Si $cos\theta < \frac{1}{\sqrt{2}}$:

$$\Phi = \begin{bmatrix} {X_2}^2 + {Y_2}^2 - {L_1}^2 \\ (X_1 - X_B)^2 + (Y_1 - Y_B)^2 - {L_2}^2 \\ X_1 * Y_2 - X_2 * Y_1 \\ (X_1 - X_B) - L_2 * cos\theta \end{bmatrix} \quad (4.8)$$

-Si $cos\theta > \frac{1}{\sqrt{2}}$:

$$\Phi = \begin{bmatrix} {X_2}^2 + {Y_2}^2 - {L_1}^2 \\ (X_1 - X_B)^2 + (Y_1 - Y_B)^2 - {L_2}^2 \\ X_1 * Y_2 - X_2 * Y_1 \\ (Y_1 - Y_B) - L_2 * sin\theta \end{bmatrix} \quad (4.9)$$

Paso 5. Matriz Jacobiana

Se procede a calcular la matriz Jacobiana derivando la matriz de restricciones Φ respecto del vector *q.* Este procedimiento se realiza derivando parcialmente cada ecuación de la matriz de restricciones respecto de cada punto del vector q. Realizando dichos cálculos obtenemos la siguiente matriz:

-Si $cos\theta < \frac{1}{\sqrt{2}}$:

$$\Phi_q = \begin{bmatrix} 0 & 0 & 2*X_2 & 2*Y_2 & 0 \\ 2*(X_1 - X_B) & 2*(Y_1 - Y_B) & 0 & 0 & 0 \\ Y_2 & -X_2 & -Y_1 & X_1 & 0 \\ 1 & 0 & 0 & 0 & L_2 * sin\theta \end{bmatrix} \quad (4.10)$$

-Si $cos\theta > \frac{1}{\sqrt{2}}$:

$$\Phi_q = \begin{bmatrix} 0 & 0 & 2*X_2 & 2*Y_2 & 0 \\ 2*(X_1 - X_B) & 2*(Y_1 - Y_B) & 0 & 0 & 0 \\ Y_2 & -X_2 & -Y_1 & X_1 & 0 \\ 1 & 0 & 0 & 0 & -L_2 * cos\theta \end{bmatrix} \quad (4.11)$$

Paso 6. Resolución del problema de posición

Como hemos explicado anteriormente, para resolver el problema de posición se utilizará el método de Newton-Raphson. A continuación, se mostrarán las fórmulas a aplicar:

$$\Phi(q + \Delta q, t) = \Phi(q, t) + \Phi_q(q, t) * \Delta q = 0 \quad (4.12)$$

Despejando esta ecuación queda de la siguiente manera:

$$\Phi_q * \Delta q = -\Phi \quad (4.13)$$

Como no es posible dividir el jacobiano respecto a su matriz de restricciones directamente lo que se debe hacer es multiplicar por la inversa del jacobiano a la izquierda ambos términos:

$${\Phi_q}^{-1} * \Phi_q * \Delta q = {\Phi_q}^{-1} * -\Phi \quad (4.14)$$

Cuando se multiplica una matriz por su inversa queda la matriz identidad por lo que simplificando obtenemos:

$$\Delta q = {\Phi_q}^{-1} * -\Phi \quad (4.15)$$

Este valor de Δq se lo añadimos al antiguo valor de q para obtener un nuevo valor de q:

$$q_1 = q_0 + \Delta q \quad (4.16)$$

Con este nuevo valor de q volvemos a operar, y así sucesivamente hasta que nuestras iteraciones nos lleven a que Φ se aproxime a 0.

Aparentemente, se trata de un sistema de cuatro ecuaciones con cinco incógnitas, pero en realidad θ es un dato de partida por lo que su valor debe permanecer fijo e igual a cero, lo que se traduce en:

$$\Phi_q * \Delta q = -\Phi \qquad (4.17)$$

$$\begin{bmatrix} 0 & 0 & 2*X_2 & 2*Y_2 & 0 \\ 2*(X_1-X_B) & 2*(Y_1-Y_B) & 0 & 0 & 0 \\ Y_2 & -X_2 & -Y_1 & X_1 & 0 \\ 1 & 0 & 0 & 0 & L_2*sin\theta \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x1_1 - x1_0 \\ y1_1 - y1_0 \\ x2_1 - x2_0 \\ y2_1 - y2_0 \\ \theta_1 - \theta_0 \end{bmatrix} =$$

$$= -\begin{bmatrix} {X_2}^2 + {Y_2}^2 - {L_1}^2 \\ (X_1-X_B)^2 + (Y_1-Y_B)^2 - {L_2}^2 \\ X_1*Y_2 - X_2*Y_1 \\ (X_1-X_B) - L_2*cos\theta \\ 0 \end{bmatrix} \qquad (4.18)$$

Siendo $x1_0$, $y1_0$, $x2_0$, $y2_0$, θ_0 la posición inicial dada a nuestro mecanismo.

4.1.1.2 Problema de velocidad

El objetivo de resolver este problema es obtener las velocidades de las variables del mecanismo una vez conocida su posición.

Como se ha visto anteriormente el sistema de ecuaciones que se debe resolver (3.20) es el siguiente:

$$\Phi_q * \dot{q} = -\Phi_t$$

Donde Φ_q es la matriz Jacobiana calculada en el problema de posición, $\dot{q}$ es el vector velocidad que queremos obtener y Φ_t la derivada parcial de las ecuaciones de restricción respecto del tiempo.

Al igual que en el problema de posición es necesario añadir un valor conocido (en este caso de velocidad) por cada grado de libertad del mecanismo para poder resolver el sistema, esto se consigue añadiendo una fila de 0 al Jacobiano excepto un 1 en la posición de nuestra variable conocida y el valor correspondiente que le se que quiera definir en la matriz Φ_t.

Ya que el tiempo no aparece explícitamente en el vector de restricciones, la derivada parcial Φ_t es cero en este caso, quedando:

$$\begin{bmatrix} 0 & 0 & 2*X_2 & 2*Y_2 & 0 \\ 2*(X_1-X_B) & 2*(Y_1-Y_B) & 0 & 0 & 0 \\ Y_2 & -X_2 & -Y_1 & X_1 & 0 \\ 1 & 0 & 0 & 0 & L_2*sin\theta \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} \dot{x1} \\ \dot{y1} \\ \dot{x2} \\ \dot{y2} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \qquad (4.19)$$

Para este problema se ha decidido que la velocidad angular $\dot{\theta} = 1$.

4.1.1.3 Problema de aceleración

El problema de aceleración (3.23) trata de obtener las aceleraciones de las variables del mecanismo a partir de la derivada de la velocidad respecto del tiempo.

$$\Phi_q * \ddot{q} = -\dot{\Phi}_q * \dot{q} - \dot{\Phi}_t$$

Donde $\ddot{q}$ es el vector de aceleraciones y el resto de componente de la ecuación ya han sido descritos anteriormente.

La matriz $\dot{\Phi}_t$ va a tener un valor nulo puesto que Φ_t solo tiene un valor y es una constante por tanto su derivada es 0.

La única matriz que queda por calcular es la derivada del Jacobiano $\dot{\Phi}_q$ ya que el resto ya se han obtenido en los problemas anteriores. Para calcular esta matriz volvemos a realizar derivadas parciales del Jacobiano, pero en este caso respecto de las velocidades $\dot{q}$.

-Si $cos\theta < \frac{1}{\sqrt{2}}$:

$$\dot{\Phi}_q = \begin{bmatrix} 0 & 0 & 2 * \dot{X}_2 & 2 * \dot{Y}_2 & 0 \\ 2 * \dot{X}_1 & 2 * \dot{Y}_1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \dot{\theta} * L_2 * cos\theta \end{bmatrix} \quad (4.20)$$

-Si $cos\theta > \frac{1}{\sqrt{2}}$:

$$\dot{\Phi}_q = \begin{bmatrix} 0 & 0 & 2 * \dot{X}_2 & 2 * \dot{Y}_2 & 0 \\ 2 * \dot{X}_1 & 2 * \dot{Y}_1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \dot{\theta} * L_2 * sen\theta \end{bmatrix} \quad (4.21)$$

Esta matriz la multiplicamos por el vector de velocidades, quedando la siguiente matriz de 4 filas y 1 columna, también se le añadirá una última fila donde irá el valor de la aceleración angular, dato que conocemos de antemano y que en este caso particular tendrá un valor de $\ddot{\theta} = 1$.

-Si $cos\theta < \frac{1}{\sqrt{2}}$:

$$\dot{\Phi}_q * \dot{q} = \begin{bmatrix} 2 * \dot{X_2}^2 + 2 * \dot{Y_2}^2 \\ 2 * \dot{X_1}^2 + 2 * \dot{Y_1}^2 \\ 0 \\ \dot{\theta}^2 * L_2 * cos\theta \\ 1 \end{bmatrix} \quad (4.22)$$

-Si $cos\theta > \frac{1}{\sqrt{2}}$:

$$\dot{\Phi}_q * \dot{q} = \begin{bmatrix} 2 * \dot{X_2}^2 + 2 * \dot{Y_2}^2 \\ 2 * \dot{X_1}^2 + 2 * \dot{Y_1}^2 \\ 0 \\ \dot{\theta}^2 * L_2 * sin\theta \\ 1 \end{bmatrix} \quad (4.23)$$

Como último paso despejamos el vector de aceleraciones $\ddot{q}$:

$$\begin{bmatrix} \ddot{x}1 \\ \ddot{y}1 \\ \ddot{x}2 \\ \ddot{y}2 \\ \ddot{\theta} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 2 * X_2 & 2 * Y_2 & 0 \\ 2 * (X_1 - X_B) & 2 * (Y_1 - Y_B) & 0 & 0 & 0 \\ Y_2 & -X_2 & -Y_1 & X_1 & 0 \\ 1 & 0 & 0 & 0 & L_2 * sin\theta \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}^{-1} * \begin{bmatrix} 2 * \dot{X_2}^2 + 2 * \dot{Y_2}^2 \\ 2 * \dot{X_1}^2 + 2 * \dot{Y_1}^2 \\ 0 \\ \dot{\theta}^2 * L_2 * sin\theta \\ 1 \end{bmatrix} \quad (4.24)$$

4.1.2 Implementación en Python

4.1.2.1 Problema de posición

Paso 1. Posición inicial

En primer lugar, abrimos nuestra aplicación Jupyter Notebook de la plataforma Anaconda y creamos un nuevo notebook.

Para implementar los distintos mecanismos en Python los primero es ejecutar todas las bibliotecas que se van a utilizar para resolver los problemas propuestos.

```
In [1]: import numpy as np
        import math as math
        import array as arr
        import pprint  # para depurar
        import matplotlib.pyplot as plt  #Para graficas
        import matplotlib.animation as animation
        import scipy.integrate as integrate
        import os
        from time import sleep
```

Como se ha visto anteriormente, para resolver los distintos problemas cinemáticos mediante métodos numéricos es necesario definir, primero, los parámetros fijos que definen al mecanismo y, posteriormente, definir una posición inicial en función de dichos parámetros.

Los parámetros fijos son aquellos que no varían con el tiempo, para resolver los distintos mecanismos es necesario definir las longitudes de las barras y los puntos fijos mediante los cuales se apoya el mecanismo.

Además, también es necesario darle un valor a una de las variables del vector *q* ya que sin ese valor conocido sería imposible resolver el problema de posición.

A continuación, utilizamos un dictionary vacío donde introducimos los valores fijos de nuestro mecanismo y el valor de la matriz dependiente que queremos añadir y, después, definimos su posición inicial. Para ello, utilizamos un numpy.array para crear nuestra matriz *q* de *m* filas y 1 columna.

```
print ('BIELA-MANIVELA CON GUIA MOVIL')
print('==================================')
#Lectura de datos por teclado
meca = {} #dictionary vacio, con propiedades de mecanismo

meca["L1"] = float (input ('Introduce longitud L1: '))
meca["L2"] = float (input ('Introduce longitud L2: '))
meca["theta"] = float (input ('Introduce angulo inicial theta:')) #En rad
meca["XB"] = float (input ('Introduce coordenada en x del punto B:'))
meca["YB"] = float (input ('Introduce coordenada en y del punto B:'))
meca["XA"] = 0
meca["YA"] = 0

# Defino posicion inicial:
q = np.array ([[meca["XB"]+meca["L2"]*math.cos(meca["theta"])], [meca["YB"]+meca["L2"]*math.sin(meca["theta"])], [5], [5],
print('q: ' + str(q))
```

De esta manera, partiremos de una posición inicial que dependerá de la longitud que introduzcamos de la barra L2 y del ángulo θ.

```
BIELA-MANIVELA CON GUIA MOVIL
==================================
Introduce longitud L1: 3
Introduce longitud L2: 1
Introduce angulo inicial theta:0.5
Introduce coordenada en x del punto B:1
Introduce coordenada en y del punto B:1
q: [[1.87758256]
 [1.47942554]
 [5.        ]
 [5.        ]
 [0.5       ]]
```

Paso 2. Matriz de restricciones

El siguiente paso será introducir las ecuaciones de la matriz de restricciones. Primero utilizamos *def* para nombrar a la función Phi. Esta función se inicializará a cero para introducirle las ecuaciones correspondientes y, además, se extraerán las coordenadas obtenidas del vector *q*.

No se debe olvidar la condición para la cual si $cos\theta < \frac{1}{\sqrt{2}}$ se utilizará la ecuación que implica el coseno y en caso contrario se aplicará la ecuación de seno.

```
def Phi (q,meca):
    #Inicializa a cero Phi
    Phi = np.zeros((5,1))

    #Extraer coordenadas
    X1 = q[0]
    Y1 = q[1]
    X2 = q[2]
    Y2 = q[3]
    theta = q[4]


    Phi[0] = X2**2 + Y2**2 - meca["L1"]**2
    Phi[1] = (X1-meca["XB"])**2 + (Y1-meca["YB"])**2 - meca["L2"]**2
    Phi[2] = X1*Y2 - Y1*X2
    if (abs(math.cos(theta)) < (math.sqrt(2)/2)):
        Phi[3] = (X1-meca["XB"])-meca["L2"]*math.cos(theta)
    else:
        Phi[3] = (Y1-meca["YB"])-meca["L2"]*math.sin(theta)

    return Phi
```

Paso 3. Matriz Jacobiana

Para el Jacobiano se aplicará el mismo método que para la matriz de restricciones, lo único que variará será la definición de las ecuaciones dependiendo de su posición en la matriz Jacobiana, para la programación en Python la primera fila es la fila [0] y para la primera columna igual, con lo cual, si queremos introducir una ecuación en la primera fila y en la primera columna, esta posición será [0,0].

Además, para poder realizar las operaciones correspondientes, añadiremos el valor 1 en la posición [4, 4] de nuestro Jacobiano que proporcionará el valor conocido al ángulo.

```
def jacob_Phiq(q,meca):
    #Inicializa a cero la matriz jacobiana
    Jacob = np.zeros((5,5))

    #Extraer coordenadas

    X1 = q[0]
    Y1 = q[1]
    X2 = q[2]
    Y2 = q[3]
    theta = q[4]

    #Montar matriz

    Jacob[0,0] = 2*X1
    Jacob[0,1] = 2*Y1
    Jacob[1,0] = -2*(X2-X1)
    Jacob[1,1] = -2*(Y2-Y1)
    Jacob[1,2] = 2*(X2-X1)
    Jacob[1,3] = 2*(Y2-Y1)
    Jacob[2,2] = -2*(meca["XB"]-X2)
    Jacob[2,3] = -2*(0-Y2)

    if (abs(math.cos(theta)) < (math.sqrt(2)/2) ):
        Jacob[3,4] = meca["L1"]*math.sin(theta)
        Jacob[3,0] = 1
    else:
        Jacob[3,4] = -meca["L1"]*math.cos(theta)
        Jacob[3,1] = 1


    Jacob[4,4] = 1

    return Jacob
```

Paso 4. Resolver problema de posición

Puesto que ya se ha obtenido la posición inicial del vector *q* y ya se han introducido la matriz de restricciones y la matriz Jacobiana, ya solo queda resolver el problema de posición.

Con lo visto anteriormente este no es un problema lineal, sino iterativo (hasta que Φ se aproxime a 0). Por tanto, será necesario imponer un límite de iteraciones ya que se puede dar el caso de que los datos introducidos no converjan para nuestro problema y eso nos llevaría a un proceso infinito donde el programa no dejaría de calcular.

También impondremos un error y una tolerancia para disminuir el número de iteraciones lo máximo posible sin que se vea afectado el resultado.

```
def resuelve_prob_posicion(q_init, meca):
    #Inicializacion de variables
    error = 1e10
    tolerancia = 1e-10

    #Inicializacion en cero de deltaQ, fi y q
    deltaQ = np.zeros ((5,1))
    q = q_init
    i=0

    # Iteraciones hasta conseguir que el error sea menor que la tolerancia

    while (error > tolerancia and i<=100):
        print("q=")
        pprint.pprint(q)


        #Extraer las coordenadas
        X1 = q[0]
        Y1 = q[1]
        X2 = q[2]
        Y2 = q[3]
        theta = q[4]

        fi=Phi(q,meca)
        print ("Phi" + "=")
        pprint.pprint(fi)
        J = jacob_Phiq(q,meca)
        print ("jacob" + "=")
        pprint.pprint(J)
        rango = np.linalg.matrix_rank(J, 1e-5)
        print("rango=" + str(rango) + "\n") #Si el rango es menor que el numero de filas no tiene solucion

        deltaQ = np.linalg.solve(J,-fi)
        q = q + deltaQ
        error = np.linalg.norm(deltaQ)  # El error es el modulo del vector
        i=i+1

        print("error iter" + str(i) + "=")
        pprint.pprint(error)
    print("num iters:" + str(i))
    if (error > tolerancia):
        raise Exception ('No se puede alcanzar la posición')
    return q

q=resuelve_prob_posicion(q,meca)
```

Con este código implementado empezará a realizar cálculos iterativos de la matriz de restricciones Φ, el Jacobiano Φ_q y el vector de coordenadas q. El programa realizará tantas iteraciones como sean necesarias hasta que el error esté por debajo de $1 * 10^{-10}$.

El resultado de los cálculos establecidos en el problema de posición nos muestra las iteraciones que se han debido realizar para la resolución de este. El número de iteraciones dependerá, primero, de la aproximación que se establece en la posición inicial a la posición real del mecanismo y, segundo, de la complejidad de las ecuaciones que se implementaron. Esta complejidad de las ecuaciones, a su vez, depende de los grados de libertad que posea el mecanismo y de las ecuaciones de restricción propuestas para su resolución.

```
q=
array([[1.87758256],
       [1.47942554],
       [2.35640298],
       [1.8567081 ],
       [0.5       ]])
Phi=
array([[1.29283251e-11],
       [0.00000000e+00],
       [0.00000000e+00],
       [0.00000000e+00],
       [0.00000000e+00]])
jacob=
array([[ 0.        ,  0.        ,  4.71280597,  3.7134162 ,  0.        ],
       [ 1.75516512,  0.95885108,  0.        ,  0.        ,  0.        ],
       [ 1.8567081 , -2.35640298, -1.47942554,  1.87758256,  0.        ],
       [ 0.        ,  1.        ,  0.        ,  0.        , -0.87758256],
       [ 0.        ,  0.        ,  0.        ,  0.        ,  1.        ]])
rango = 5
```

```
error iter6=
2.1547208461909 56e-12
num iters: 6
array([[1.87758256],
       [1.47942554],
       [2.35640298],
       [1.8567081 ],
       [0.5       ]])
```

Como se observa el código resuelve el problema hasta que en la matriz denominada "Phi" todos sus valores se aproximan a 0.

En el caso de este problema se ha requerido de 6 iteraciones para su resolución.

Paso 5. Representar el mecanismo

Para acabar con el problema de posición y comprobar que los datos obtenidos son correctos se realizará el dibujo del mecanismo. Esto nos proporcionará la visión que necesitamos para ver cómo funcionan las ecuaciones que implementamos y como se configura un mecanismo de estas características.

Desde el punto de vista docente este es uno de los pasos más importantes pues es la muestra de que las ecuaciones enseñadas en clase y los procesos realizados tienen un valor real y se pueden crear mecanismos en base a la teoría implantada en la asignatura.

Para dibujar el mecanismo primero extraemos la coordenadas obtenidas del vector *q* y definimos un cuadro de dibujo con los ejes de la misma dimensión con "plt.axis('equal`)".

Para dibujar tanto el mecanismo biela-manivela con guía móvil como el resto de los mecanismos, es necesario hacerlo por partes. Dibujamos cada barra por separado indicando la posición inicial y final de cada barra, en nuestro caso tenemos:

$Barra\ 1 \rightarrow ([XA, X2], [YA, Y2])$

$Barra\ 2 \rightarrow ([X1, X2], [Y1, Y2])$

$Barra\ 3 \rightarrow ([XB, X1], [YB, Y1])$

Definiendo las distintas barras ya se obtendría un dibujo del mecanismo biela-manivela con guía móvil.

```
def dibuja_mecanismo(q, meca):

    #Extraer los puntos moviles del mecanismo
    X1 = q[0]
    Y1 = q[1]
    X2 = q[2]
    Y2 = q[3]
    theta = q[4]

    plt.axis('equal')

    plt.plot ([meca["XA"], X2], [meca["YA"], Y2], )
    plt.plot ([X1, X2], [Y1, Y2])
    plt.plot ([meca["XB"], X1], [meca["YB"], Y1])

    plt.plot(meca["XA"], meca["YA"])
    plt.plot(meca["XB"], meca["YB"])
    plt.plot(X1, Y1)
    plt.plot(X2, Y2)
    plt.show()#block=False)
    return

dibuja_mecanismo(q,meca)
```

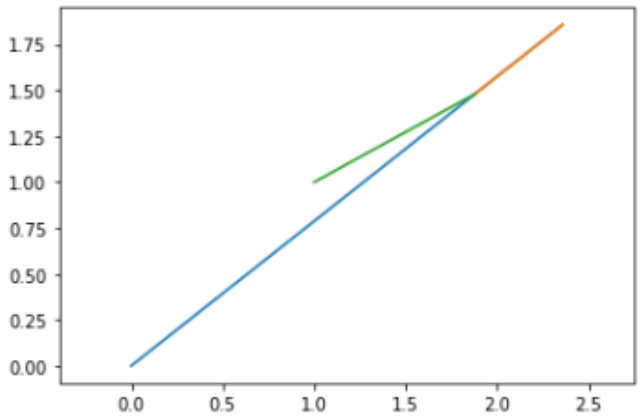


Figura 4. 4. Representación gráfica biela-manivela con guía móvil

Paso 6. Diseño

Un apartado al que también se le ha dado mucho énfasis en este trabajo es a conseguir un diseño lo más representativo posible de los distintos mecanismos puesto que lo que se quiere obtener es un mecanismo lo más realista posible para mejorar la compresión de la asignatura de Teoría de Mecanismos a los alumnos que la cursen.

En nuestro caso, necesitamos definir correctamente los puntos fijos, los puntos móviles, las distintas barras y como interaccionan entre sí. Para ellos nos guiamos de varios ejemplos de código que nos proporciona dicha web donde muestran tanto el código fuente como ejemplos prácticos.

Para el diseño del mecanismo biela-manivela con guía móvil, primero hemos graficado las barras para que la forma coincidiese con una barra lo más realista posible.

La forma se ha creado utilizando "linewidth" para definir la anchura de la barra, "solid_capstyle" para crear esa forma redondeada en los extremos y "color" para cambiarle el color.

En el caso de los puntos lo más importante era diferencia entre puntos móviles y puntos fijos, para ellos se ha añadido "marker" junto con la forma deseada. En este paso un triángulo para los puntos fijos, un rombo para mostrar la forma de la deslizadera que constituye el punto 1 y un círculo para mostrar el punto 2. Con "markersize" también se ha conseguido cambiar el tamaño de los puntos para que fuesen más representativos.

```
plt.plot ([meca["XA"], X2], [meca["YA"], Y2], linewidth=12, solid_capstyle='round', color='tab:red')
plt.plot ([meca["XA"], X2], [meca["YA"], Y2], linewidth=6, solid_capstyle='round', color='white')
plt.plot ([X1, X2], [Y1, Y2], color='white')
plt.plot ([meca["XB"], X1], [meca["YB"], Y1], linewidth=6, solid_capstyle='projecting',color='slategrey')

plt.plot(meca["XA"], meca["YA"], 'bo', marker='^', markersize=10)
plt.plot(meca["XB"], meca["YB"], 'go', marker='^', markersize=10)
plt.plot(X1, Y1, 'go', marker="D", markersize=10)
plt.plot(X2, Y2, marker='o', color='grey')
```

Después de modificar su diseño, así queda el mecanismo biela-manivela con guía móvil.

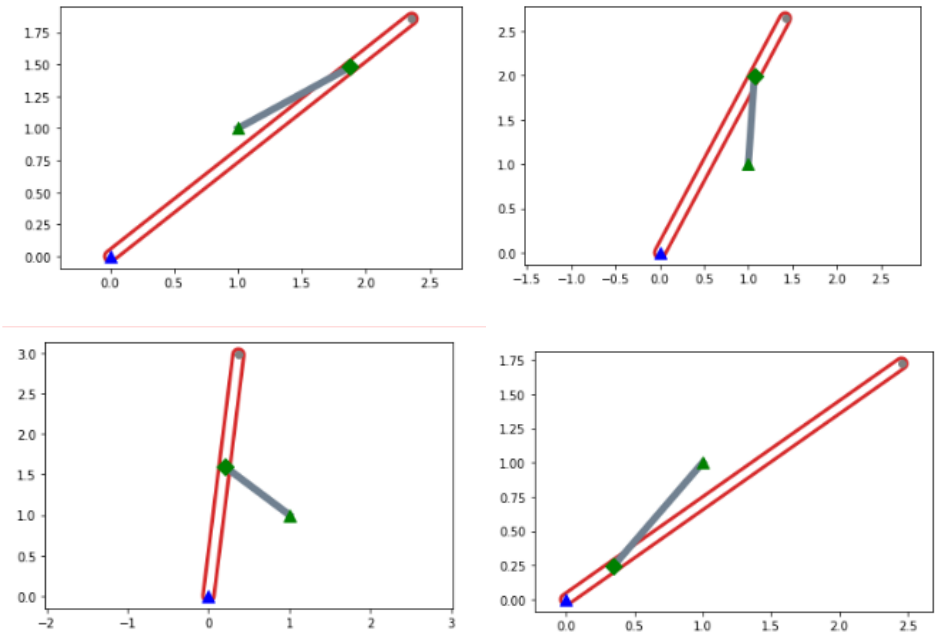


Figura 4. 5. Diseño del biela-manivela con guía móvil

4.1.2.2 Problema de velocidad

El problema de velocidad se resuelve de forma lineal, es decir, proporciona una solución exacta sin necesidad de iteraciones.

Para resolver el problema de velocidad necesitamos de la matriz jacobiana obtenida en el apartado anterior. Por lo tanto, abriremos un nuevo notebook y copiaremos el código mediante el cuál se elaboró dicha matriz.

A continuación, solo será necesario introducir la ecuación en lenguaje Python. Este proceso se consigue con "numpy.linalg.solve", pues utilizando este código se calcula la solución exacta x de una ecuación matricial determinada tal que $a * x = b$.

También cabe mencionar que se debe introducir el valor añadido de la velocidad angular ($\dot{\theta} = 1$)

```
def resuelve_prob_velocidad(q,qp,meca):
    qp = np.linalg.solve(jacob_Phiq(q,meca),qp)
    #print ("qp=")
    #pprint.pprint(qp)

    return qp

qp = np.zeros ((5,1))
qp[4]=1
qp = resuelve_prob_velocidad (q,qp, meca)
qp
```

De aquí obtenemos nuestro vector de velocidades $\dot{q}$:

```
array([[-0.47942554],
       [ 0.87758256],
       [-0.76588443],
       [ 0.97200651],
       [ 1.        ]])
```

Este es el resultado del vector velocidad para el mecanismo biela-manivela con guía móvil. No pasa nada si se obtienen valores negativos, esto es completamente normal puesto que un valor negativo indica que el sentido real de la velocidad en esa posición será contrario al supuesto y, al contrario, un valor positivo indica que el movimiento se produce en el mismo sentido.

4.1.2.3 Problema de aceleración

Para la aceleración ha sido necesario realizar el cálculo de $\dot{\Phi}_q * \dot{q}$ para que se quedase una matriz del modo $a * x = b$ y de esta manera utilizar el código con "numpy.linalg.solve", donde $a = \Phi_q$, $x = \ddot{q}$ y $b = \dot{\Phi}_q * \dot{q}$.

Introducimos el vector posición, el vector velocidad y creamos la matriz b para realizar el cálculo. También cabe mencionar que se ha introducido en la última fila el dato con la aceleración conocida $\ddot{\theta} = 1$.

```
def resuelve_prob_aceleracion (q,qp, qpp, meca):

    #Extraer las posiciones
    X1 = q[0]
    Y1 = q[1]
    X2 = q[2]
    Y2 = q[3]
    theta = q[4]
    #Extraer las velocidades
    X1q = qp[0]
    Y1q = qp[1]
    X2q = qp[2]
    Y2q = qp[3]
    thetaq = qp[4]
    b=qpp

    b[0] = 2*(X1q)**2 + 2*(Y1q)**2
    b[1] = -2*X1q*(X2q-X1q) - 2*Y1q*(Y2q-Y1q) + 2*X2q*(X2q-X1q) + 2*Y2q*(Y2q-Y1q)
    if (abs(math.cos(theta)) < (math.sqrt(2)/2) ):
        b[3] = thetaq**2 * (meca["L1"] * math.cos(theta))
    else:
        b[3] =  thetaq**2 * (meca["L1"] * math.sin(theta))

    qpp = np.linalg.solve(-jacob_Phiq(q,meca),b)


    return qpp


qpp=np.zeros((5,1))
qpp[4] = 1 #Aceleracion conocida
qpp=resuelve_prob_aceleracion(q,qp,qpp,meca)
qpp
```

```
array([[-1.3570081 ],
       [ 0.39815702],
       [-1.87613915],
       [-0.        ],
       [-1.        ]])
```

4.1.2.4 Gráfica de velocidades y aceleraciones

Una vez calculadas las velocidades y aceleraciones del mecanismo se pueden graficar para cada coordenada. Los gráficos representan las velocidades y aceleraciones para una vuelta completa del mecanismo, especificando una tabla del ángulo que va de 0 a 2 π revoluciones para 50 valores.

Se genera una matriz de ceros para cada coordenada que contiene 50 posiciones, y estos valores se irán sumando a medida que aumente el ángulo. El contador se inicializa a cero, luego se genera un bucle for que incrementa el contador uno por uno hasta que alcanza el valor máximo definido en la matriz especificada originalmente. Dentro del bucle la variable correspondiente al ángulo, se asigna al ángulo t, que irá aumentando a medida que se repita el proceso. Conociendo este valor se obtienen otros valores del problema.

Una vez resuelto, se genera una matriz de ceros para el vector de velocidad qp y el valor de velocidad se asigna al componente correspondiente a los datos de velocidad conocidos. Una vez recopiladas todas las componentes del vector qp, estos valores se almacenan en la matriz de 0 creada al principio cuya función es almacenar la velocidad de cada coordenada. Luego se incrementa el contador. Cuando se completa el bucle, se trazan cuatro gráficos utilizando funciones de la biblioteca Matplotlib.

```
def grafica_velocidad(q,meca):

    th = np.linspace(0,2*3.1416,50)

    VX1 = np.zeros((50,0))
    VY1 = np.zeros((50,0))
    VX2 = np.zeros((50,0))
    VY2 = np.zeros((50,0))
    i=0
    for t in th:

        q[4] = t

        q = resuelve_prob_posicion (q, meca)
        qp = np.zeros ((5,1))
        #Velocidad del gdl. En una vuelta completa del angulo se cumple angulo=2*Pi*t
        qp[4]=1
        qp = resuelve_prob_velocidad (q,qp, meca)


        VX1 = np.append(VX1, qp[0])
        VY1 = np.append(VY1, qp[1])
        VX2 = np.append(VX2, qp[2])
        VY2 = np.append(VY2, qp[3])
        i=i+1


    fig, axs = plt.subplots(ncols=2, figsize=(15, 15))
    plt.subplot(2,2,1)
    plt.plot(th,VX1)
    plt.xlabel ('Angulo')
    plt.ylabel ('Velocidad')
    plt.title ('Velocidad de X1')

    plt.subplot(2,2,2)
    plt.plot(th,VY1)
    plt.xlabel ('Angulo')
    plt.ylabel ('Velocidad')
    plt.title ('Velocidad de Y1')

    plt.subplot(2,2,3)
    plt.plot(th,VX2)
    plt.xlabel ('Angulo')
    plt.ylabel ('Velocidad')
    plt.title ('Velocidad de X2')

    plt.subplot(2,2,4)
    plt.plot(th,VY2)
    plt.xlabel ('Angulo')
    plt.ylabel ('Velocidad')
    plt.title ('Velocidad de Y2')

    plt.show()
    return

grafica_velocidad (q,meca)
```

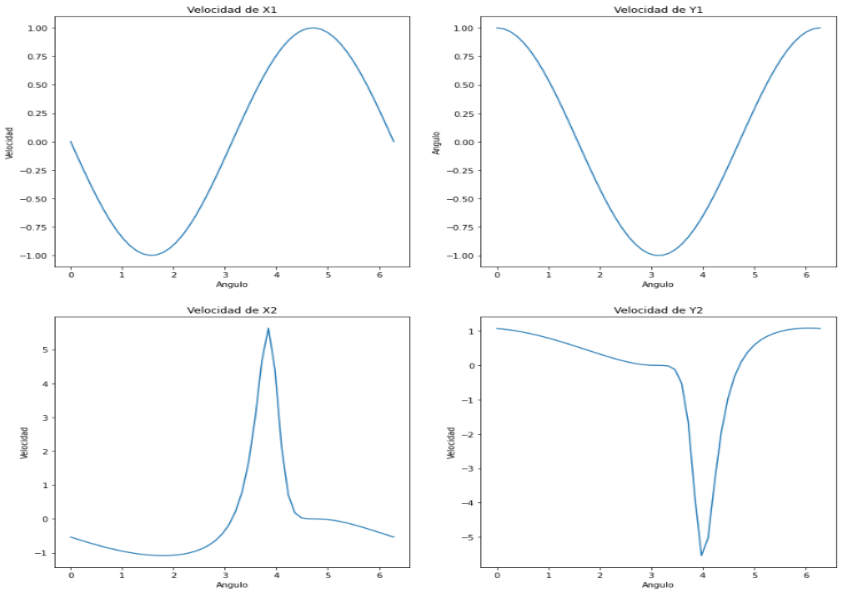


Figura 4. 6. Gráficas velocidades biela-manivela con guía móvil

```
def grafica_aceleracion(q,meca):

    th = np.linspace(0,2*3.1416,50)

    AX1 = np.zeros((50,0))
    AY1 = np.zeros((50,0))
    AX2 = np.zeros((50,0))
    AY2 = np.zeros((50,0))


    i=0
    for t in th:

        q[4] = t
        q = resuelve_prob_posicion (q,meca)

        qp = np.zeros((5,1))
        qp[4] = 1  #inicializar qp en 0 con qp[4] = 1 rad/s
        qp = resuelve_prob_velocidad(q, qp, meca)

        qpp = np.zeros((5,1))
        qpp[4] = 1  #inicializar qp en 0 con qpp[4] = 1 rad/s**2
        qpp = resuelve_prob_aceleracion(q,qp, qpp, meca)

        AX1 = np.append(AX1, qpp[0])
        AY1 = np.append(AY1, qpp[1])
        AX2 = np.append(AX2, qpp[2])
        AY2 = np.append(AY2, qpp[3])

        i=i+1


    fig, axs = plt.subplots(ncols=2, figsize=(15, 15))
    plt.subplot(2,2,1)
    plt.plot(th,AX1)
    plt.xlabel ('Tiempo')
    plt.ylabel ('Aceleración')
    plt.title ('Aceleración de X1')

    plt.subplot(2,2,2)
    plt.plot(th,AY1)
    plt.xlabel ('Tiempo')
    plt.ylabel ('Aceleración')
    plt.title ('Aceleración de Y1')

    plt.subplot(2,2,3)
    plt.plot(th,AX2)
    plt.xlabel ('Tiempo')
    plt.ylabel ('Aceleración')
    plt.title ('Aceleración de X2')

    plt.subplot(2,2,4)
    plt.plot(th,AY2)
    plt.xlabel ('Tiempo')
    plt.ylabel ('Aceleración')
    plt.title ('Aceleración de Y2')

    plt.show()
    return

grafica_aceleracion (q,meca)
```

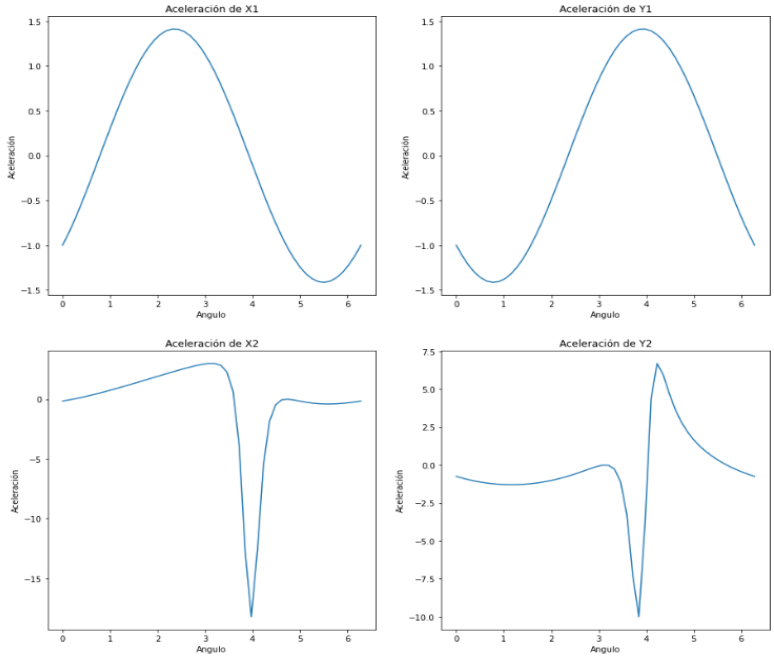


Figura 4. 7. Gráficas aceleraciones biela-manivela con guía móvil

Estos valores interpretan a la perfección la velocidad de cada punto del mecanismo pues como se ve en el punto 1 tenemos una velocidad constante ya que es el que realiza el movimiento giratorio mientras que en el punto 2 se observa picos de incremento que se ocasionan debido al movimiento realizado (ver animación), esto ocurrirá de la misma manera en el mecanismo de Retorno Rápido puesto que ambos realizan movimientos similares.

4.1.3 Animación

Para animar el mecanismo, primero debemos reescribir el código donde se resuelve el problema de posición, ya que es el código que define las diferentes coordenadas de las variables del mecanismo.

Las bibliotecas usadas en este caso son matplotlib.animation y la función HTML. Después de importar las bibliotecas, se establece los límites de los ejes y el valor calculado de q se guarda como last_q. Después, se define una función inicial en la que se inicializa la animación, y luego se crea la función de animación, que será la función que creará la animación en sí. En esta última función la variable last_q se define como una variable global. La función que realmente se usará es un bucle que vuelve a calcular el valor de la posición y lo representa en la misma pantalla. Por lo tanto, a q se le da el valor de last_q para que al principio q sea el último valor que calcular.

Entonces, dada la velocidad angular correspondiente a los grados de libertad del mecanismo se multiplica por el término correspondiente al ángulo en el vector q en el contador repetidor. Una vez calculado este valor, se resuelve el problema de posición para obtener el resto de los elementos que componen el vector q. Cuando se cambian los valores, se asigna el valor q a la variable last_q, quedando el nuevo valor. Luego se extraen las coordenadas y se establecen las coordenadas de los ejes x e y. Finalmente, se llama a la función FuncAnimation y se le pasan una serie de parámetros.

La primera será la forma creada anteriormente. La siguiente es la función de Animación, que se llama animate. A continuación, se muestra dónde iniciar la función, que en este caso sería init. Después de esta función, el número de imágenes a utilizar, es decir, el parámetro i se pasa a la función de animación. A continuación, aparece el parámetro Interval, que indica el intervalo de tiempo de los fotogramas en milisegundos. Finalmente, blit se establece = True, lo que indica que no se vuelve a calcular todo el fotograma, sino que solo se actualizan los fragmentos recién capturados en cada fotograma, lo que permite que la animación se ejecute durante más tiempo rápidamente. (Lozano 2019)

```
%matplotlib inline
import matplotlib.pyplot as plt

from matplotlib import animation, rc
from IPython.display import HTML


fig, ax = plt.subplots()

ax.set_xlim(( -5, 5))
ax.set_ylim((-5,5))
ax.set_aspect('equal')

line, = ax.plot([], [], lw=2, marker='.', markersize=4)

last_q = q

def init():
    line.set_data([], [])
    return (line,)

def animate(i,q,meca):
    global last_q
    q = last_q
    # i: contador de iteracion: hay que mapearla a un ángulo de la manivela
    omega=2*3.14159/100  # vel. angular
    q[4] = i*omega

    #llamar problema de pos:
    q = resuelve_prob_posicion(q, meca)
    last_q = q

    #Extraer las coordenadas
    X1 = q[0]
    Y1 = q[1]
    X2 = q[2]
    Y2 = q[3]
    theta = q[4]

    x=[meca["XB"], X1, meca["XA"], X1, meca["XA"], X2]
    y=[meca["YB"], Y1, meca["YA"], Y1, meca["YA"], Y2]


    line.set_data(x, y)
    return (line,)


anim = animation.FuncAnimation(fig, animate, init_func=init, fargs=(q,meca),
                               frames=100, interval=20,
                               blit=True)

HTML(anim.to_html5_video())
```

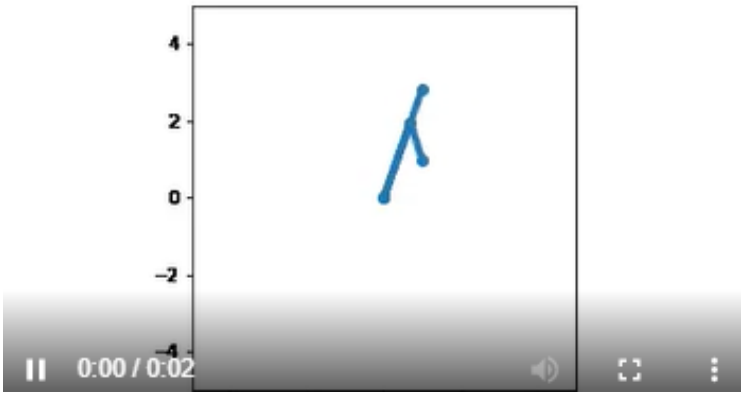


Figura 4. 8. Animación biela-manivela con guía móvil

4.2 Retorno Rápido

El mecanismo de Retorno Rápido, también conocido como mecanismo de Whitworth se trata de un mecanismo de yugo escocés giratorio que genera movimientos de carrera irregulares con un movimiento de avance lento y un movimiento de retorno rápido.

Es una versión más completa que el mecanismo biela-manivela con guía móvil ya que, mientras en el anterior el último punto de la biela esta libre y sin restricciones (de ahí su nombre), en este si debemos añadirle la restricción de que uno de los puntos de la biela se mantenga recto en una superficie.

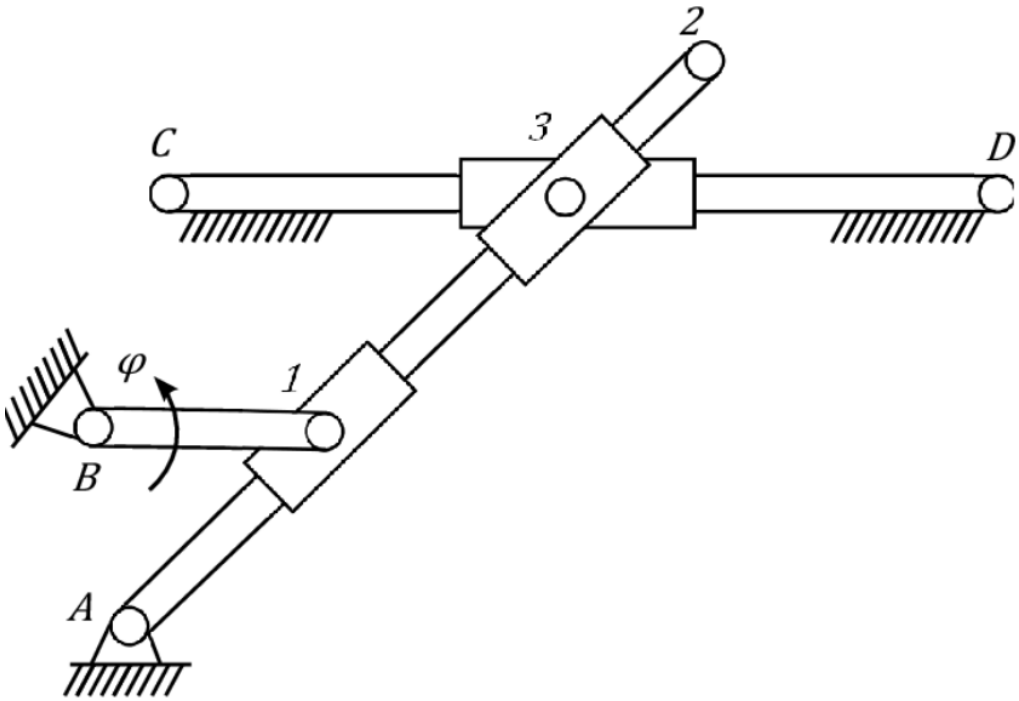


Figura 4. 9. Ejemplo mecanismo de retorno rápido

Las aplicaciones de este mecanismo son las mismas que el biela-manivela con guía móvil pero también cabe añadir que este mecanismo es muy utilizado en herramienta de maquinado para realizar cortes sobre una pieza.

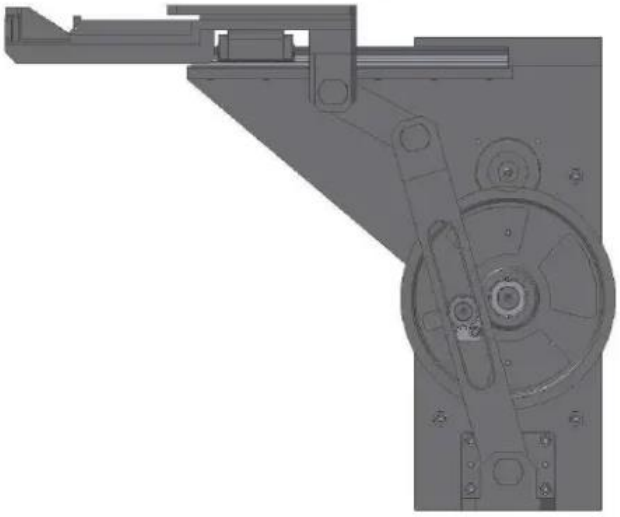


Figura 4. 10. Máquina cortadora

4.2.1 Cálculo Teórico

4.2.1.1 Problema de posición

Paso 1. Modelado del mecanismo

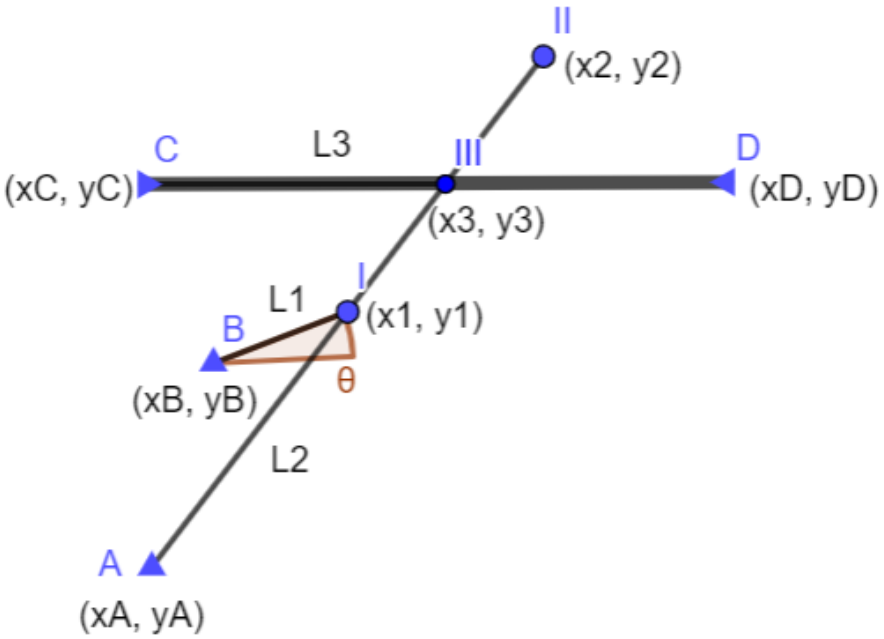


Figura 4. 11. Modelado Retorno Rápido

Paso 2. Grados de libertad

Utilizando la misma ecuación que antes (4.0) calculamos los grados de libertad:

$$G = 3 * (N - 1) - 2P_I - P_{II} = 3 * (7 - 1) - 2 * 8 - 1 = 1$$

Paso 3. Definición del vector q

Para este problema también se han utilizado coordenadas naturales y se ha tomado como variable independiente el ángulo que forma el punto B con el punto 1 (barra L1). De esta manera el vector de coordenadas queda:

$$q = \begin{bmatrix} x1 \\ y1 \\ x2 \\ y2 \\ x3 \\ y3 \\ \theta \end{bmatrix} \qquad (4.25)$$

Los valores fijos por los que ha sido definido el mecanismo de retorno rápido son los siguientes:

$$L1 = 0{,}6$$

$$L2 = 4$$

$$L3 = 6{,}5$$

$$\theta = 1$$

$$X_B = 1$$

$$Y_B = 1$$

$$X_A = 0$$

$$Y_A = 0$$

Paso 4. Matriz de restricciones

- Barras B-1 y A-2:

$$(X_1 - X_B)^2 + (Y_1 - Y_B)^2 - {L_1}^2 = 0 \qquad (4.26)$$

$$(X_2 - X_A)^2 + (Y_2 - Y_A)^2 - {L_2}^2 = 0 \qquad (4.27)$$

- Alineamiento A-1-2:

$$(X_1 - X_A) * (Y_2 - Y_A) - (Y_1 - Y_A) * (X_2 - X_A) = 0 \qquad (4.28)$$

- Alineamiento A-3-2:

$$(X_3 - X_A) * (Y_2 - Y_A) - (Y_3 - Y_A) * (X_2 - X_A) = 0 \qquad (4.29)$$

- Alineamiento C-3-D

$$(X_3 - X_C) * (Y_D - Y_C) - (Y_3 - Y_C) * (X_D - X_C) = 0 \qquad (4.30)$$

- Ángulo θ: Para la ecuación de restricción podemos usar la ecuación con el seno o con el coseno, ambas son perfectamente válidas, de hecho, dependiendo de la posición nos conviene más una u otra, es por eso que para la resolución de los problemas utilizaremos ambas.

$$(X_1 - X_B) - L_1 * cos\theta = 0 \qquad (4.31)$$
$$(Y_1 - Y_B) - L_1 * sin\theta = 0 \qquad (4.32)$$

Ahora se procede a formar la matriz de restricciones:

$$\Phi = \begin{bmatrix} (X_1 - X_B)^2 + (Y_1 - Y_B)^2 - {L_1}^2 \\ (X_2 - X_A)^2 + (Y_2 - Y_A)^2 - {L_2}^2 \\ (X_1 - X_A) * (Y_2 - Y_A) - (Y_1 - Y_A) * (X_2 - X_A) \\ (X_3 - X_A) * (Y_2 - Y_A) - (Y_3 - Y_A) * (X_2 - X_A) \\ (X_3 - X_C) * (Y_D - Y_C) - (Y_3 - Y_C) * (X_D - X_C) \\ (X_1 - X_B) - L_1 * cos\theta \end{bmatrix} \qquad (4.33)$$

Esta matriz se puede simplificar puesto que nuestro origen lo hemos estipulado en el punto fijo A(0,0), además, debemos añadirle la misma condición que para el mecanismo anterior, de esta manera la matriz queda:

-Si $cos\theta < 0{,}5$:

$$\Phi = \begin{bmatrix} (X_1 - X_B)^2 + (Y_1 - Y_B)^2 - {L_1}^2 \\ {X_2}^2 + {Y_2}^2 - {L_2}^2 \\ X_1 * Y_2 - Y_1 * X_2 \\ X_3 * Y_2 - Y_3 * X_2 \\ (X_3 - X_C) * (Y_D - Y_C) - (Y_3 - Y_C) * (X_D - X_C) \\ (X_1 - X_B) - L_1 * cos\theta \end{bmatrix} \qquad (4.34)$$

-Si $cos\theta > 0{,}5$:

$$\Phi = \begin{bmatrix} (X_1 - X_B)^2 + (Y_1 - Y_B)^2 - {L_1}^2 \\ {X_2}^2 + {Y_2}^2 - {L_2}^2 \\ X_1 * Y_2 - Y_1 * X_2 \\ X_3 * Y_2 - Y_3 * X_2 \\ (X_3 - X_C) * (Y_D - Y_C) - (Y_3 - Y_C) * (X_D - X_C) \\ (Y_1 - Y_B) - L_1 * sin\theta \end{bmatrix} \qquad (4.35)$$

Paso 5. Matriz Jacobiana

-Si $cos\theta < 0{,}5$:

$$\Phi_q = \begin{bmatrix} 2 * (X_1 - X_B) & 2 * (Y_1 - Y_B) & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 * X_2 & 2 * Y_2 & 0 & 0 & 0 \\ Y_2 & -X_2 & -Y_1 & X_1 & 0 & 0 & 0 \\ 0 & 0 & -Y_3 & X_3 & Y_2 & -X_2 & 0 \\ 0 & 0 & 0 & 0 & Y_D - Y_C & -(X_D - X_C) & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & L_1 * sin\theta \end{bmatrix} \qquad (4.36)$$

-Si $cos\theta > 0{,}5$:

$$\Phi_q = \begin{bmatrix} 2*(X_1 - X_B) & 2*(Y_1 - Y_B) & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2*X_2 & 2*Y_2 & 0 & 0 & 0 \\ Y_2 & -X_2 & -Y_1 & X_1 & 0 & 0 & 0 \\ 0 & 0 & -Y_3 & X_3 & Y_2 & -X_2 & 0 \\ 0 & 0 & 0 & 0 & Y_D - Y_C & -(X_D - X_C) & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & -L_1 * cos\theta \end{bmatrix} \quad (4.37)$$

Paso 6. Resolución del problema de posición

El método de resolución del problema de posición es exactamente igual que el explicado en el mecanismo anterior. De nuevo debemos añadir una fila a la matriz Jacobiana (4.36) para que el problema sea un sistema determinado y se pueda resolver, este dato será de nuevo el ángulo θ.

$$\begin{bmatrix} 2*(X_1 - X_B) & 2*(Y_1 - Y_B) & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2*X_2 & 2*Y_2 & 0 & 0 & 0 \\ Y_2 & -X_2 & -Y_1 & X_1 & 0 & 0 & 0 \\ 0 & 0 & -Y_3 & X_3 & Y_2 & -X_2 & 0 \\ 0 & 0 & 0 & 0 & Y_D - Y_C & -(X_D - X_C) & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & L_1 * sin\theta \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x1_1 - x1_0 \\ y1_1 - y1_0 \\ x2_1 - x2_0 \\ y2_1 - y2_0 \\ x3_1 - x3_0 \\ y3_1 - y3_0 \\ \theta_1 - \theta_0 \end{bmatrix} =$$

$$= -\begin{bmatrix} (X_1 - X_B)^2 + (Y_1 - Y_B)^2 - {L_1}^2 \\ {X_2}^2 + {Y_2}^2 - {L_2}^2 \\ X_1 * Y_2 - Y_1 * X_2 \\ X_3 * Y_2 - Y_3 * X_2 \\ (X_3 - X_C) * (Y_D - Y_C) - (Y_3 - Y_C) * (X_D - X_C) \\ (X_1 - X_B) - L_1 * cos\theta \end{bmatrix} \quad (4.38)$$

4.2.1.2 Problema de velocidad

$$\Phi_q * \dot{q} = -\Phi_t$$

Para el caso de mecanismo, como tiene 1 g.d.l. al igual que el anterior solo es necesario añadir un valor conocido, de nuevo ese valor será el de la velocidad angular $\dot{\theta} = 1$.

$$\begin{bmatrix} 2*(X_1 - X_B) & 2*(Y_1 - Y_B) & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2*X_2 & 2*Y_2 & 0 & 0 & 0 \\ Y_2 & -X_2 & -Y_1 & X_1 & 0 & 0 & 0 \\ 0 & 0 & -Y_3 & X_3 & Y_2 & -X_2 & 0 \\ 0 & 0 & 0 & 0 & Y_D - Y_C & -(X_D - X_C) & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & L_1 * sin\theta \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} \dot{x1} \\ \dot{y1} \\ \dot{x2} \\ \dot{y2} \\ \dot{x3} \\ \dot{y3} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \quad (4.39)$$

4.2.1.3 Problema de aceleración

$$\Phi_q * \ddot{q} = -\dot{\Phi}_q * \dot{q} - \dot{\Phi}_t$$

Procedemos al cálculo de la derivada del jacobiano:

-Si $cos\theta < 0{,}5$:

$$\dot{\Phi}_q = \begin{bmatrix} 2 * \dot{X_1} & 2 * \dot{Y_1} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 * \dot{X_2} & 2 * \dot{X_2} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \dot{\theta} * L_1 * cos\theta \end{bmatrix} \quad (4.40)$$

-Si $cos\theta > 0{,}5$:

$$\dot{\Phi}_q = \begin{bmatrix} 2 * \dot{X_1} & 2 * \dot{Y_1} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 * \dot{X_2} & 2 * \dot{X_2} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \dot{\theta} * L_1 * sin\theta \end{bmatrix} \quad (4.41)$$

Para simplificar el sistema multiplicaremos la derivada del jacobiano por el vector de velocidades y, como anteriormente, añadiremos el valor de la aceleración del ángulo $\ddot{\theta} = 1$.

-Si $cos\theta < 0{,}5$:

$$\dot{\Phi}_q * \dot{q} = \begin{bmatrix} 2 * \dot{X_1}^2 + 2 * \dot{Y_1}^2 \\ 2 * \dot{X_2}^2 + 2 * \dot{Y_2}^2 \\ 0 \\ 0 \\ 0 \\ \dot{\theta}^2 * L_1 * cos\theta \\ 1 \end{bmatrix} \quad (4.42)$$

-Si $cos\theta > 0{,}5$:

$$\dot{\Phi}_q * \dot{q} = \begin{bmatrix} 2 * \dot{X_1}^2 + 2 * \dot{Y_1}^2 \\ 2 * \dot{X_2}^2 + 2 * \dot{Y_2}^2 \\ 0 \\ 0 \\ 0 \\ \dot{\theta}^2 * L_1 * sin\theta \\ 1 \end{bmatrix} \qquad (4.43)$$

Por tanto, la ecuación matricial para resolver el problema de aceleraciones del mecanismo de Retorno Rápido quedará de la siguiente manera:

$$\begin{bmatrix} 2*(X_1 - X_B) & 2*(Y_1 - Y_B) & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2*X_2 & 2*Y_2 & 0 & 0 & 0 \\ Y_2 & -X_2 & -Y_1 & X_1 & 0 & 0 & 0 \\ 0 & 0 & -Y_3 & X_3 & Y_2 & -X_2 & 0 \\ 0 & 0 & 0 & 0 & Y_D - Y_C & -(X_D - X_C) & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & L_1 * sin\theta \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} \ddot{x1} \\ \ddot{y1} \\ \ddot{x2} \\ \ddot{y2} \\ \ddot{x3} \\ \ddot{y3} \\ \ddot{\theta} \end{bmatrix} =$$

$$= \begin{bmatrix} 2 * \dot{X_1}^2 + 2 * \dot{Y_1}^2 \\ 2 * \dot{X_2}^2 + 2 * \dot{Y_2}^2 \\ 0 \\ 0 \\ 0 \\ \dot{\theta}^2 * L_1 * cos\theta \\ 1 \end{bmatrix} \qquad (4.44)$$

4.2.2 Implementación en Python

4.2.2.1 Problema de posición

Paso 1. Posición inicial

De nuevo ejecutamos las librería que necesitamos para resolver los distintos problemas del mecanismo y posteriormente introducimos los valores iniciales y definimos la posición inicial.

```
print ('MECANISMO DE RETORNO RAPIDO')
print ('============================')
# Lectura de datos por teclado

meca = {} # dictionary vacio, con propiedades de mecanismo

meca["L1"] = float (input ('Introduce longitud L1:'))
meca["L2"] = float (input ('Introduce longitud L2:'))
meca["theta"] = float (input ('Introduce angulo inicial theta:')) #En rad
meca["XA"] = 0
meca["XB"] = 1
meca["XC"] = -2
meca["XD"] = 4.5
meca["YA"] = 0
meca["YB"] = 1
meca["YC"] = 2
meca["YD"] = meca["YC"]

# Defino posicion inicial:
q = np.array ([[meca["XB"]+meca["L1"]*math.cos(meca["theta"])], [meca["YB"]+meca["L1"]*math.sin(meca["theta"])], [5], [5], [4.7],
print('q: ' + str(q))
```

De esta manera se genera el vector de coordenadas inicial.

```
MECANISMO DE RETORNO RAPIDO
============================
Introduce longitud L1:0.6
Introduce longitud L2:4
Introduce angulo inicial theta:1.5
q: [[1.04244232]
 [1.59849699]
 [5.        ]
 [5.        ]
 [4.7       ]
 [2.        ]
 [1.5       ]]
```

Paso 2. Matriz de restricciones

Extraemos las coordenadas de nuestro vector e introducimos las ecuaciones para generar la matriz de restricciones.

```
#MATRIZ DE RESTRICCIONES

def Phi (q,meca):
    #Inicializa a cero Phi
    Phi = np.zeros((7,1))

    #Extraer coordenadas
    X1 = q[0]
    Y1 = q[1]
    X2 = q[2]
    Y2 = q[3]
    X3 = q[4]
    Y3 = q[5]
    theta = q[6]

    Phi[0] = (X1-meca["XB"])**2 + (Y1-meca["YB"])**2 - meca["L1"]**2
    Phi[1] = (X2)**2 + (Y2)**2 -meca["L2"]**2
    Phi[2] = (X1)*(Y2) - (Y1)*(X2)
    Phi[3] = (X3)*(Y2) - (Y3)*(X2)
    Phi[4] = 0

    if (abs(math.cos(theta)) < 0.5 ):
        Phi[5] = (X1-meca["XB"])-meca["L1"]*math.cos(theta)
    else:
        Phi[5] = (Y1-meca["YB"])-meca["L1"]*math.sin(theta)


    return Phi
```

Cabe recordar la implementación de la condición $cos\theta < 0{,}5$.

Paso 3. Matriz Jacobiana

De la misma manera que en el anterior mecanismo establecemos la derivada parcial de cada ecuación de la matriz de restricciones respecto de cada coordenada en su casilla correspondiente.

```
#JACOBIANO

def jacob_Phiq(q,meca):
    #Inicializa a cero la matriz jacobiana
    Jacob = np.zeros((7,7))

    #Extraer coordenadas

    X1 = q[0]
    Y1 = q[1]
    X2 = q[2]
    Y2 = q[3]
    X3 = q[4]
    Y3 = q[5]
    theta = q[6]

    #Montar matriz

    Jacob[0,0] = 2*(X1-meca["XB"])
    Jacob[0,1] = 2*(Y1-meca["YB"])
    Jacob[1,2] = 2*(X2)
    Jacob[1,3] = 2*(Y2)
    Jacob[2,0] = (Y2)
    Jacob[2,1] = -(X2)
    Jacob[2,2] = -(Y1)
    Jacob[2,3] = (X1)
    Jacob[3,2] = -(Y3)
    Jacob[3,3] = (X3)
    Jacob[3,4] = (Y2)
    Jacob[3,5] = -(X2)
    Jacob[4,4] = 0
    Jacob[4,5] = -(meca["XD"]-meca["XC"])

    if (abs(math.cos(theta)) < 0.5 ):
        Jacob[5,6] = meca["L1"]*math.sin(theta)
        Jacob[5,0] = 1
    else:
        Jacob[5,6] = -meca["L1"]*math.cos(theta)
        Jacob[5,1] = 1

    Jacob[6,6] = 1

    return Jacob
```

Paso 4. Resolver problema de posición

Procedemos a la resolución del mecanismo.

```
#RESUELVE PROBLEMA DE POSICION

def resuelve_prob_posicion(q_init, meca):
    #Inicializacion de variables
    error = 1e10
    tolerancia = 1e-10

    #Inicializacion en cero de deltaQ, fi y q
    deltaQ = np.zeros ((7,1))
    q = q_init
    i=0


    while (error > tolerancia and i<=100):
        print("q=")
        pprint.pprint(q)


        #Extraer las coordenadas
        X1 = q[0]
        Y1 = q[1]
        X2 = q[2]
        Y2 = q[3]
        X3 = q[4]
        Y3 = q[5]
        theta = q[6]


        fi=Phi(q,meca)
        print ("Phi" + "=")
        pprint.pprint(fi)
        J = jacob_Phiq(q,meca)
        print ("jacob" + "=")
        pprint.pprint(J)
        rango = np.linalg.matrix_rank(J, 1e-5)
        print("rango=" + str(rango) + "\n")
        deltaQ = np.linalg.solve(J,-fi)
        q = q + deltaQ
        error = np.linalg.norm(deltaQ)
        i=i+1

        print("error iter" + str(i) + "=")
        pprint.pprint(error)
    print("num iters:" + str(i))
    if (error > tolerancia):
        raise Exception ('No se puede alcanzar la posición')
    return q

q=resuelve_prob_posicion(q,meca)
```

```
q=
array([[1.04244232],
       [1.59849699],
       [2.18498919],
       [3.35049582],
       [1.30427811],
       [2.        ],
       [1.5       ]])
Phi=
array([[-1.11022302e-16],
       [ 0.00000000e+00],
       [ 0.00000000e+00],
       [ 3.55271368e-15],
       [ 0.00000000e+00],
       [-1.38777878e-17],
       [ 0.00000000e+00]])
jacob=
array([[ 0.08488464,  1.19699398,  0.        ,  0.        ,  0.        ,
         0.        ,  0.        ],
       [ 0.        ,  0.        ,  4.36997837,  6.70099164,  0.        ,
         0.        ,  0.        ],
       [ 3.35049582, -2.18498919, -1.59849699,  1.04244232,  0.        ,
         0.        ,  0.        ],
       [ 0.        ,  0.        , -2.        ,  1.30427811,  3.35049582,
        -2.18498919,  0.        ],
       [ 0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
        -6.5       ,  0.        ],
       [ 1.        ,  0.        ,  0.        ,  0.        ,  0.        ,
         0.        ,  0.59849699],
       [ 0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
         0.        ,  1.        ]])
rango=7

error iter6=
1.1246128343950386e-15
num iters:6
```

El número de iteraciones para resolver el problema de posición ha sido de 6 iteraciones.

Paso 5. Representación y diseño del mecanismo

Para la representación de este mecanismo se han utilizado varios "plt.plot" para representar tanto las barras como los puntos que conforman el mecanimo.

Para el diseño del mecanismo de Retorno Rápido, de nuevo nos hemos ayudado de las herramientas de código de la web Matplotlib.

Se han utilizado el mismo tipo de herramientas que las explicadas en el mecanismo biela-manivela con guía móvil.

```
def dibuja_mecanismo(q, meca):


    # Extraer los puntos moviles del mecanismo
    X1 = q[0]
    Y1 = q[1]
    X2 = q[2]
    Y2 = q[3]
    X3 = q[4]
    Y3 = q[5]
    theta = q[6]



    plt.axis('equal')


    plt.plot ([meca["XA"], X2], [meca["YA"], Y2], linewidth=10, solid_capstyle='round', color='silver')
    plt.plot ([X1, X2, X3], [Y1, Y2, Y3], linewidth=10, solid_capstyle='round', color='silver')
    plt.plot ([X1, meca["XB"]], [Y1, meca ["YB"]], linewidth=6, solid_capstyle='round')
    plt.plot ([meca["XC"], meca["XD"], X3], [meca["YC"], meca["YD"], Y3], linewidth=12, solid_capstyle='butt', color='tab:red')
    plt.plot ([meca["XC"], meca["XD"], X3], [meca["YC"], meca["YD"], Y3], linewidth=6, solid_capstyle='round', color='white')

    plt.plot(meca["XA"], meca["YA"], 'bo', marker='^', markersize=15)
    plt.plot(meca["XB"], meca["YB"], 'go', marker='^', markersize=15)
    plt.plot(meca["XC"], meca["YD"], 'go', marker='^', markersize=15)
    plt.plot(meca["XD"], meca["YD"], 'bo', marker='^', markersize=15)
    plt.plot(X3, Y3, marker="s", markersize=12)
    plt.plot(X2, Y2, 'bo')
    plt.plot(X1, Y1, marker='o', color='blue')

    plt.show()#block=False)
    return

dibuja_mecanismo(q,meca)
```

De esta manera el mecanismo diseñado es el siguiente.

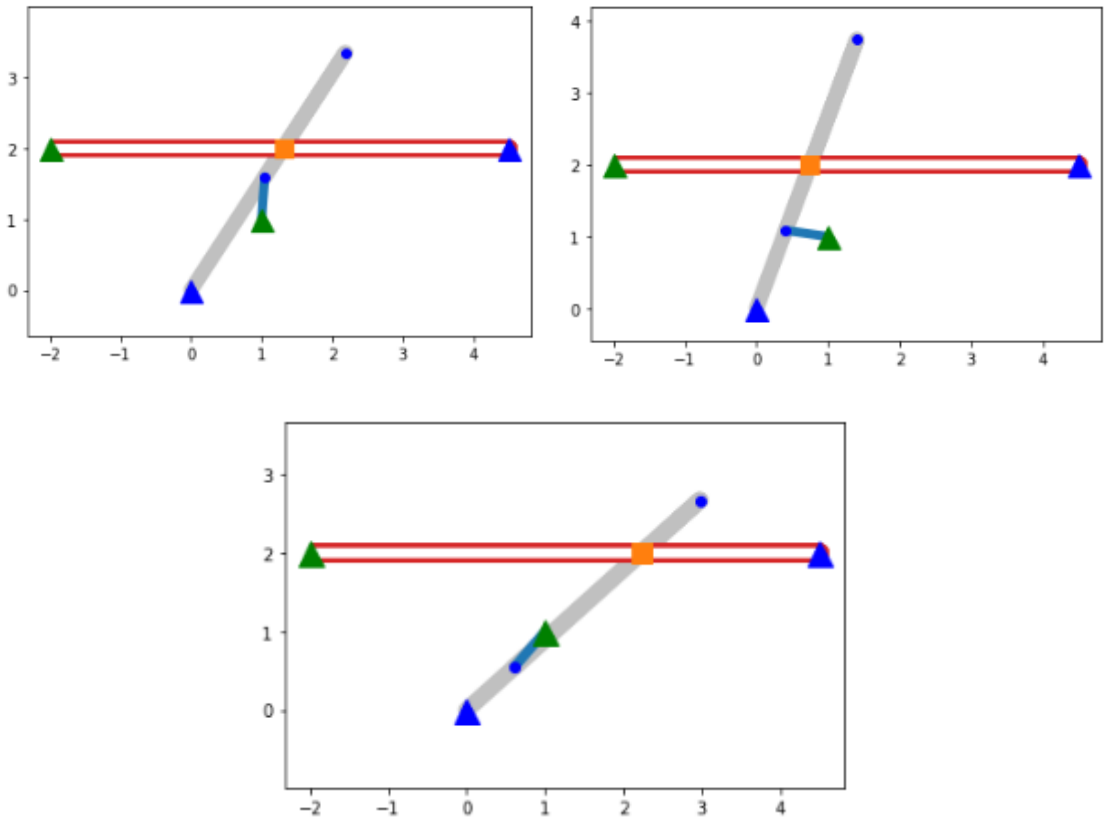


Figura 4. 12. Representación y diseño retorno rápido

4.2.2.2 Problema de velocidad

Nota: es necesario que a la entrada, qp sea todo ceros menos los valores de velocidad en los g.d.l.

```
# PASO 2 PROBLEMA DE VELOCIDAD

def resuelve_prob_velocidad(q,qp, meca):

    qp = np.linalg.solve(jacob_Phiq(q,meca),qp)

    return qp


qp = np.zeros ((7,1))
    #Velocidad del gdl.
qp[6]=1
qp=resuelve_prob_velocidad (q,qp, meca)

qp
```

```
array([[ 0.42073549],
       [ 0.27015115],
       [ 0.6285082 ],
       [-0.32287219],
       [ 0.39691499],
       [-0.        ],
       [ 1.        ]])
```

4.2.2.3 Problema de aceleración

```
#PASO 3 PROBLEMA DE ACELERACION


def resuelve_prob_aceleracion (q,qp,qpp,meca):

    #Extraer las posiciones
    X1 = q[0]
    Y1 = q[1]
    X2 = q[2]
    Y2 = q[3]
    X3 = q[4]
    Y3 = q[5]
    theta = q[6]
    #Extraer las velocidades
    X1q = qp[0]
    Y1q = qp[1]
    X2q = qp[2]
    Y2q = qp[3]
    X3q = qp[4]
    Y3q = qp[5]
    thetaq = qp[6]
    b=qpp

    b[0] = 2*(X1q)**2 + 2*(Y1q)**2
    b[1] = 2*(X2q)**2 + 2*(Y2q)**2
    b[2] = 0
    b[3] = 0
    b[4] = 0

    if (abs(math.cos(theta)) < (0.5) ):
        b[5] = thetaq**2 * (meca["L1"] * math.cos(theta))
    else:
        b[5] = thetaq**2 * (meca["L1"] * math.sin(theta))


    qpp = np.linalg.solve(jacob_Phiq(q,meca),-b)


    return qpp

qpp=np.zeros ((7,1))
qpp[6] = 1 #Aceleracion conocida
qpp = resuelve_prob_aceleracion(q,qp, qpp,meca)
qpp
```

```
array([[ 0.146704  ],
       [-0.5       ],
       [ 0.84887957],
       [-0.56081177],
       [ 0.5681001 ],
       [-0.        ],
       [-1.        ]])
```

4.2.2.4 Graficas de velocidades y aceleraciones

```
#PASO 4: GRÁFICAS DE VELOCIDADES
def grafica_velocidad(q,meca):

    th = np.linspace(0,2*3.1416,200)

    VX1 = np.zeros((50,0))
    VY1 = np.zeros((50,0))
    VX2 = np.zeros((50,0))
    VY2 = np.zeros((50,0))
    VX3 = np.zeros((50,0))
    VY3 = np.zeros((50,0))

    i=0
    for t in th:

        q[6] = t
        q = resuelve_prob_posicion (q,meca)

        qp = np.zeros((7,1))
        qp[6] = 1  #inicializar qp en 0 con qp[6] = 1 rad/s
        qp = resuelve_prob_velocidad(q, qp, meca)


        VX1 = np.append(VX1, qp[0])
        VY1 = np.append(VY1, qp[1])
        VX2 = np.append(VX2, qp[2])
        VY2 = np.append(VY2, qp[3])
        VX3 = np.append(VX3, qp[4])
        VY3 = np.append(VY3, qp[5])
        i=i+1


    fig, axs = plt.subplots(ncols=2, figsize=(20, 20))
    plt.subplot(3,2,1)
    plt.plot(th,VX1)
    plt.xlabel ('Angulo')
    plt.ylabel ('Velocidad')
    plt.title ('Velocidad de X1')

    plt.subplot(3,2,2)
    plt.plot(th,VY1)
    plt.xlabel ('Angulo')
    plt.ylabel ('Velocidad')
    plt.title ('Velocidad de Y1')

    plt.subplot(3,2,3)
    plt.plot(th,VX2)
    plt.xlabel ('Angulo')
    plt.ylabel ('Velocidad')
    plt.title ('Velocidad de X2')

    plt.subplot(3,2,4)
    plt.plot(th,VY2)
    plt.xlabel ('Angulo')
    plt.ylabel ('Velocidad')
    plt.title ('Velocidad de Y2')

    plt.subplot(3,2,5)
    plt.plot(th,VX3)
    plt.xlabel ('Angulo')
    plt.ylabel ('Velocidad')
    plt.title ('Velocidad de X3')

    plt.subplot(3,2,6)
    plt.plot(th,VY3)
    plt.xlabel ('Angulo')
    plt.ylabel ('Velocidad')
    plt.title ('Velocidad de Y3')

    plt.show()
    return

grafica_velocidad (q,meca)
```

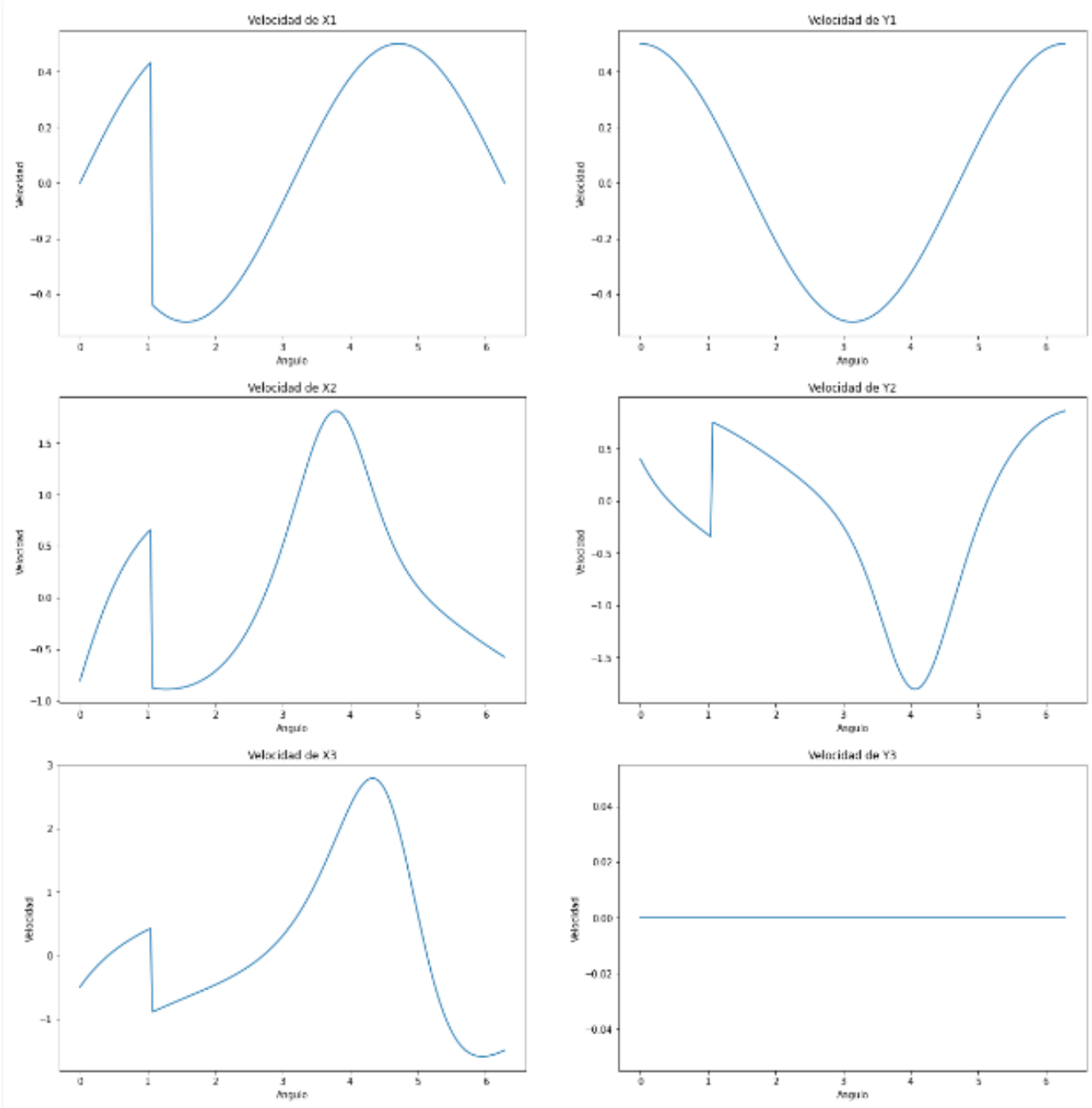


Figura 4. 13. Gráficas velocidades retorno rápido

Esta es la representación gráfica de las velocidades en los distintos puntos del mecanismo para una vuelta completa.

Los picos que se observan en varios puntos son debidos a un incremento y decremento de la velocidad a causa del movimiento característico de este mecanismo.

Cabe mencionar que en el punto Y3 su velocidad es 0 puesto que este punto solo tiene movimiento en el eje horizontal "X".

```
#PASO 5: GRÁFICAS ACELERACIONES

def grafica_aceleracion(q,meca):

    th = np.linspace(0,2*3.1416,50)

    AX1 = np.zeros((50,0))
    AY1 = np.zeros((50,0))
    AX2 = np.zeros((50,0))
    AY2 = np.zeros((50,0))
    AX3 = np.zeros((50,0))
    AY3 = np.zeros((50,0))


    i=0
    for t in th:

        q[6] = t
        q = resuelve_prob_posicion (q,meca)

        qp = np.zeros((7,1))
        qp[6] = 1  #inicializar qp en 0 con qp[6] = 1 rad/s
        qp = resuelve_prob_velocidad(q, qp, meca)

        qpp = np.zeros((7,1))
        qpp[6] = 1  #inicializar qp en 0 con qpp[4] = 1 rad/s**2
        qpp = resuelve_prob_aceleracion(q,qp, qpp, meca)

        AX1 = np.append(AX1, qpp[0])
        AY1 = np.append(AY1, qpp[1])
        AX2 = np.append(AX2, qpp[2])
        AY2 = np.append(AY2, qpp[3])
        AX3 = np.append(AX3, qpp[4])
        AY3 = np.append(AY3, qpp[5])
        i=i+1


    fig, axs = plt.subplots(ncols=2, figsize=(20, 20))
    plt.subplot(3,2,1)
    plt.plot(th,AX1)
    plt.xlabel ('Angulo')
    plt.ylabel ('Aceleración')
    plt.title ('Aceleración de X1')

    plt.subplot(3,2,2)
    plt.plot(th,AY1)
    plt.xlabel ('Angulo')
    plt.ylabel ('Aceleración')
    plt.title ('Aceleración de Y1')

    plt.subplot(3,2,3)
    plt.plot(th,AX2)
    plt.xlabel ('Angulo')
    plt.ylabel ('Aceleración')
    plt.title ('Aceleración de X2')

    plt.subplot(3,2,4)
    plt.plot(th,AY2)
    plt.xlabel ('Angulo')
    plt.ylabel ('Aceleración')
    plt.title ('Aceleración de Y2')

    plt.subplot(3,2,5)
    plt.plot(th,AX3)
    plt.xlabel ('Angulo')
    plt.ylabel ('Aceleración')
    plt.title ('Aceleración de X3')

    plt.subplot(3,2,6)
    plt.plot(th,AY3)
    plt.xlabel ('Angulo')
    plt.ylabel ('Aceleración')
    plt.title ('Aceleración de Y3')

    plt.show()
    return

grafica_aceleracion (q,meca)
```

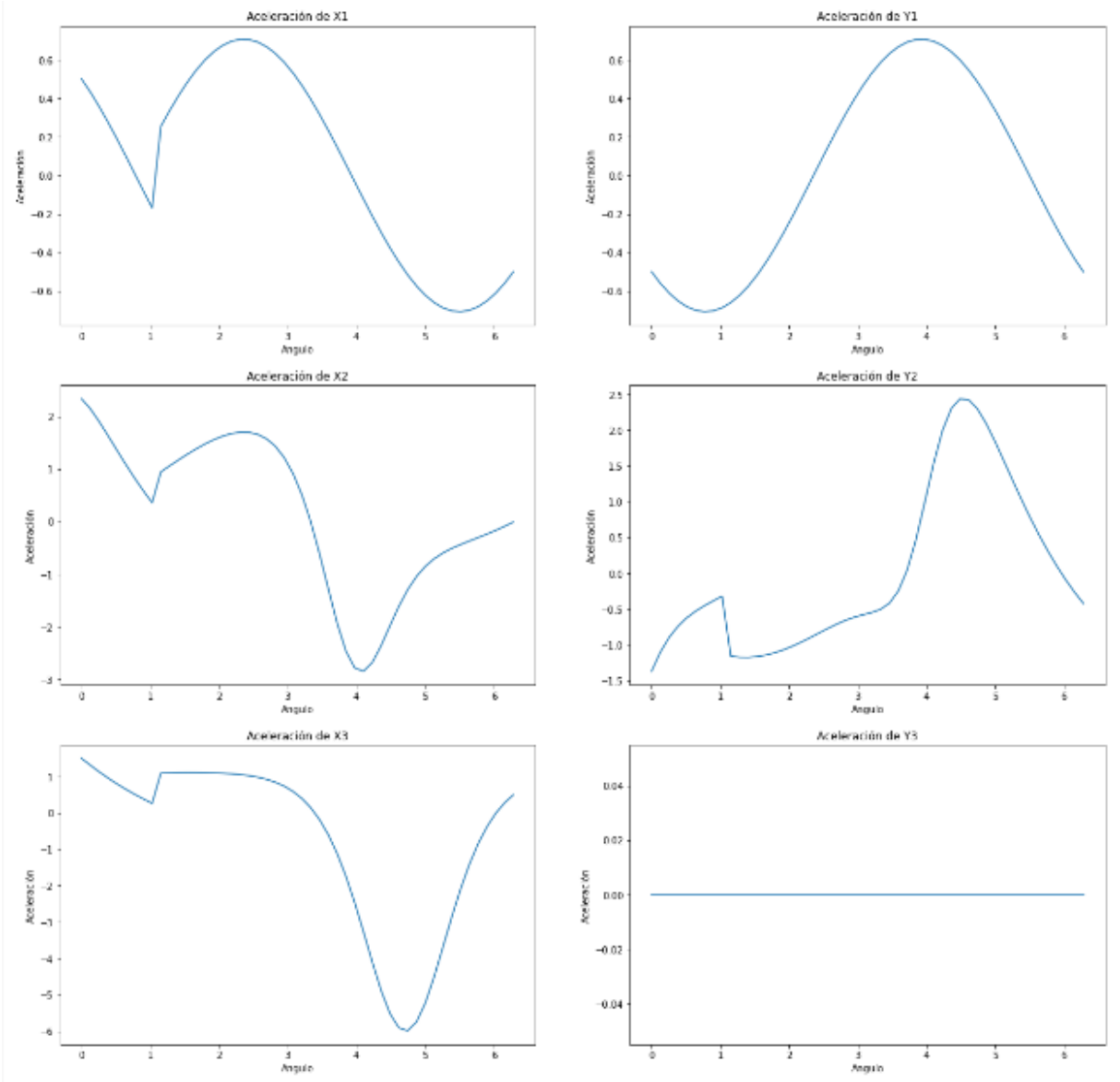


Figura 4. 14. Gráficas de aceleraciones retorno rápido

4.3 Mecanismo de Stephenson

El mecanismo de Stephenson, conocido con en este nombre en honor a su inventor George Stephenson (1781-1848), es un mecanismo más complejo y complicado de analizar, si se le compara con un mecanismo de cuatro barras.

Hay distintos tipos, estos mecanismos cuentan con 6 eslabones unidos por articulaciones, es decir, son mecanismos de 6 barras.

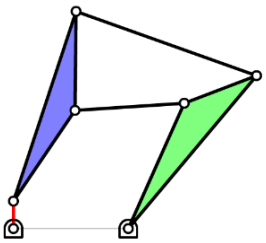


Figura 4. 15. Mecanismo de Stephenson I

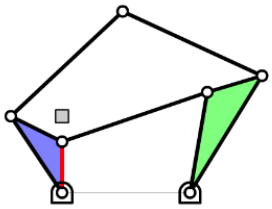


Figura 4. 16. Mecanismo de Stephenson II

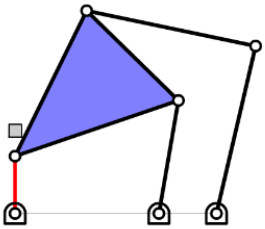
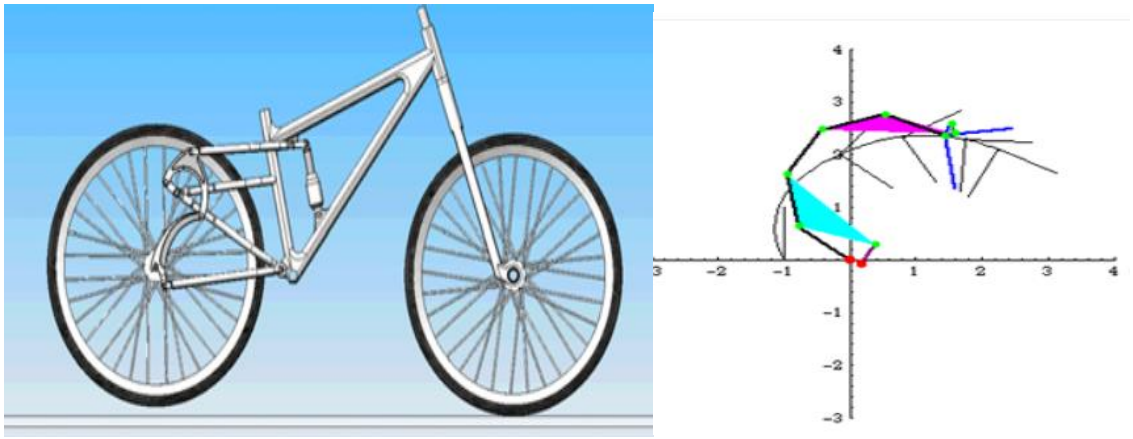


Figura 4. 17. Mecanismo de Stephenson II

La aplicación de este tipo de mecanismo es muy diversa, principalmente trata de simular distintos movimientos que podría tener una máquina en el uso industrial, pero también los encontramos en el uso cotidiano como, por ejemplo, el mecanismo de suspensión de las bicicletas.



4.3.1 Cálculo Teórico

4.3.1.1 Problema de posición

Paso 1. Modelado del mecanismo

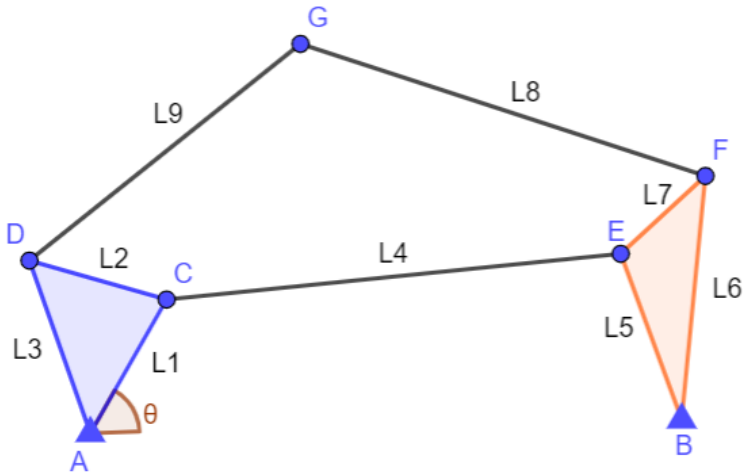


Figura 4. 18. Modelado mecanismo de Stephenson

Paso 2. Grados de libertad

$$G = 3 * (N - 1) - 2P_I - P_{II} = 3 * (6 - 1) - 2 * 7 = 1$$

Paso 3. Definición del vector q

$$q = \begin{bmatrix} xC \\ yC \\ xE \\ yE \\ xD \\ yD \\ xF \\ yF \\ xG \\ yG \\ \theta \end{bmatrix} \qquad (4.45)$$

Para el mecanismo de Stephenson se ha valorado definir todas las longitudes puesto que lo que interesa es el cálculo de los distintos problemas cinemáticos para el uso docente, por tanto, para evitar los errores debido al dimensionado se ha procedido a definirlas todas como valores fijos.

$$L1 = 3$$

$$L2 = 5$$

$$L3 = 4$$

$$L4 = 8$$

$$L5 = 6$$

$$L6 = 8$$

$$L7 = 3$$

$$L8 = 9$$

$$L9 = 8$$

$$X_B = 7$$

$$Y_B = 0$$

$$X_A = 0$$

$$Y_A = 0$$

Paso 4. Matriz de restricciones

Para definir la matriz de restricciones aplicamos las siguientes ecuaciones de restricción:

- Sólido A-C-D:

$$(X_C - X_A)^2 + (Y_C - Y_A)^2 - {L_1}^2 = 0 \quad (4.46)$$

$$(X_D - X_A)^2 + (Y_D - Y_A)^2 - {L_2}^2 = 0 \quad (4.47)$$

$$(X_C - X_D)^2 + (Y_C - Y_D)^2 - {L_3}^2 = 0 \quad (4.48)$$

- Sólido B-E-F:

$$(X_E - X_B)^2 + (Y_E - Y_B)^2 - {L_5}^2 = 0 \quad (4.49)$$

$$(X_B - X_F)^2 + (Y_B - Y_F)^2 - {L_6}^2 = 0 \quad (4.50)$$

$$(X_F - X_E)^2 + (Y_F - Y_E)^2 - {L_7}^2 = 0 \quad (4.51)$$

- Barra C-E:

$$(X_E - X_C)^2 + (Y_E - Y_C)^2 - {L_4}^2 = 0 \quad (4.52)$$

- Barra D-G:

$$(X_G - X_D)^2 + (Y_G - Y_D)^2 - {L_9}^2 = 0 \quad (4.53)$$

- Barra F-G:

$$(X_F - X_G)^2 + (Y_F - Y_G)^2 - {L_8}^2 = 0 \quad (4.54)$$

- Ángulo θ:

$$(X_C - X_A) - L_1 * cos\theta = 0 \quad (4.55)$$
$$(Y_C - Y_A) - L_1 * sin\theta = 0 \quad (4.56)$$

A partir de estas ecuaciones la matriz de restricciones queda de la siguiente manera:

$$\Phi = \begin{bmatrix} {X_C}^2 + {Y_C}^2 - {L_1}^2 \\ {X_D}^2 + {Y_D}^2 - {L_2}^2 \\ (X_C - X_D)^2 + (Y_C - Y_D)^2 - {L_3}^2 \\ (X_E - X_B)^2 + (Y_E - Y_B)^2 - {L_5}^2 \\ (X_B - X_F)^2 + (Y_B - Y_F)^2 - {L_6}^2 \\ (X_F - X_E)^2 + (Y_F - Y_E)^2 - {L_7}^2 \\ (X_E - X_C)^2 + (Y_E - Y_C)^2 - {L_4}^2 \\ (X_G - X_D)^2 + (Y_G - Y_D)^2 - {L_9}^2 \\ (X_F - X_G)^2 + (Y_F - Y_G)^2 - {L_8}^2 \\ X_C - L_1 * cos\theta \end{bmatrix} \quad (4.57)$$

Paso 5. Matriz Jacobiana

Al igual que en los problemas anteriores, la matriz Jacobiana se calcula derivando cada una de las filas de la matriz de restricción respecto de cada una de las coordenadas del vector q.

Debemos poner de nuevo la condición de seno y coseno dependiendo de la posición de nuestro mecanismo.

-Si $cos\theta < \frac{1}{\sqrt{2}}$:

$$\Phi_q = \begin{bmatrix} 2*X_C & 2*Y_C & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2*X_D & 2*Y_D & 0 & 0 & 0 & 0 & 0 \\ 2*(X_C-X_D) & 2*(Y_C-Y_D) & 0 & 0 & -2*(X_C-X_D) & -2*(Y_C-Y_D) & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2*(X_E-X_B) & 2*(Y_E-Y_B) & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -2*(X_B-X_F) & -2*(Y_B-Y_F) & 0 & 0 & 0 \\ 0 & 0 & -2*(X_F-X_E) & -2*(Y_F-Y_E) & 0 & 0 & 2*(X_F-X_E) & 2*(Y_F-Y_E) & 0 & 0 & 0 \\ -2*(X_E-X_C) & -2*(Y_E-Y_C) & 2*(X_E-X_C) & 2*(Y_E-Y_C) & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -2*(X_G-X_D) & -2*(Y_G-Y_D) & 0 & 0 & 2*(X_G-X_D) & 2*(Y_G-Y_D) & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2*(X_F-X_G) & 2*(Y_F-Y_G) & -2*(X_F-X_G) & -2*(Y_F-Y_G) & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & L_1*sin\theta \end{bmatrix}$$

(4.58)

-Si $cos\theta > \frac{1}{\sqrt{2}}$:

$$\Phi_q = \begin{bmatrix} 2*X_C & 2*Y_C & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2*X_D & 2*Y_D & 0 & 0 & 0 & 0 & 0 \\ 2*(X_C-X_D) & 2*(Y_C-Y_D) & 0 & 0 & -2*(X_C-X_D) & -2*(Y_C-Y_D) & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2*(X_E-X_B) & 2*(Y_E-Y_B) & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -2*(X_B-X_F) & -2*(Y_B-Y_F) & 0 & 0 & 0 \\ 0 & 0 & -2*(X_F-X_E) & -2*(Y_F-Y_E) & 0 & 0 & 2*(X_F-X_E) & 2*(Y_F-Y_E) & 0 & 0 & 0 \\ -2*(X_E-X_C) & -2*(Y_E-Y_C) & 2*(X_E-X_C) & 2*(Y_E-Y_C) & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -2*(X_G-X_D) & -2*(Y_G-Y_D) & 0 & 0 & 2*(X_G-X_D) & 2*(Y_G-Y_D) & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2*(X_F-X_G) & 2*(Y_F-Y_G) & -2*(X_F-X_G) & -2*(Y_F-Y_G) & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -L_1*cos\theta \end{bmatrix}$$

(4.59)

Paso 6. Resolución del problema de posición

El proceso de resolución es exactamente igual que en los mecanismos anteriores (3.15), debido al gran número de coordenadas dependientes y del gran tamaño del jacobiano resulta complejo expresar la ecuación matricial que corresponde a la resolución del problema de posición para este mecanismo, el proceso que sigue no varía con respecto a los anteriores por tanto lo único que será necesario es definir un valor fijo para que el sistema de ecuaciones sea determinado.

$$\Phi_q * \Delta q = -\Phi$$

Este dato de partida será θ y para este problema permanecerá fijo e igual a 0.

4.3.1.2 Problema de velocidad

Seguimos el mismo proceso de resolución del problema de velocidad estipulado anteriormente (3.20), a partir de la ecuación:

$$\Phi_q * \dot{q} = -\Phi_t$$

Donde Φ_t es la derivada de la matriz de restricciones respecto del tiempo y tendrá un valor de 0 en cada fila excepto en el valor que se le añada por defecto. Para este caso el valor que le hemos definido es el de la velocidad angular $\dot{\theta} = 1$.

4.3.1.3 Problema de aceleración

Para este problema tenemos la siguiente ecuación (3.23):

$$\Phi_q * \ddot{q} = -\dot{\Phi}_q * \dot{q} - \dot{\Phi}_t$$

Donde todos los datos ya han sido calculados previamente excepto la derivada del jacobiano.

-Si $cos\theta < \frac{1}{\sqrt{2}}$:

$$\dot{\Phi}_q = \begin{bmatrix} 2*\dot{X}_C & 2*\dot{Y}_C & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2*\dot{X}_D & 2*\dot{Y}_D & 0 & 0 & 0 & 0 & 0 \\ 2*\dot{X}_C & 2*\dot{Y}_C & 0 & 0 & 2*\dot{X}_D & 2*\dot{Y}_D & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2*\dot{X}_E & 2*\dot{Y}_E & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2*\dot{X}_F & 2*\dot{Y}_F & 0 & 0 & 0 \\ 0 & 0 & 2*\dot{X}_E & 2*\dot{Y}_E & 0 & 0 & 2*\dot{X}_F & 2*\dot{Y}_F & 0 & 0 & 0 \\ 2*\dot{X}_C & 2*\dot{Y}_C & 2*\dot{X}_E & 2*\dot{Y}_E & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2*\dot{X}_D & 2*\dot{Y}_D & 0 & 0 & 2*\dot{X}_G & 2*\dot{Y}_G & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2*\dot{X}_F & 2*\dot{Y}_F & 2*\dot{X}_G & 2*\dot{Y}_G & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & L_1*cos\theta \end{bmatrix} \quad (4.60)$$

-Si $cos\theta > \frac{1}{\sqrt{2}}$:

$$\dot{\Phi}_q = \begin{bmatrix} 2*\dot{X}_C & 2*\dot{Y}_C & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2*\dot{X}_D & 2*\dot{Y}_D & 0 & 0 & 0 & 0 & 0 \\ 2*\dot{X}_C & 2*\dot{Y}_C & 0 & 0 & 2*\dot{X}_D & 2*\dot{Y}_D & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2*\dot{X}_E & 2*\dot{Y}_E & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2*\dot{X}_F & 2*\dot{Y}_F & 0 & 0 & 0 \\ 0 & 0 & 2*\dot{X}_E & 2*\dot{Y}_E & 0 & 0 & 2*\dot{X}_F & 2*\dot{Y}_F & 0 & 0 & 0 \\ 2*\dot{X}_C & 2*\dot{Y}_C & 2*\dot{X}_E & 2*\dot{Y}_E & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2*\dot{X}_D & 2*\dot{Y}_D & 0 & 0 & 2*\dot{X}_G & 2*\dot{Y}_G & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2*\dot{X}_F & 2*\dot{Y}_F & 2*\dot{X}_G & 2*\dot{Y}_G & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & L_1*sin\theta \end{bmatrix} \quad (4.61)$$

El matriz $\dot{\Phi}_t$ es 0 y ahora se procede a realizar la multiplicación $\dot{\Phi}_q * \dot{q}$ para simplificar la resolución del sistema, añadiendo el dato de partida de la aceleración angular que en este caso será $\ddot{\theta} = 0$:

$$\dot{\Phi}_q * \dot{q} = \begin{bmatrix} 2 * \dot{X_C}^2 + 2 * \dot{Y_C}^2 \\ 2 * \dot{X_D}^2 + 2 * \dot{Y_D}^2 \\ 2 * \dot{X_C}^2 + 2 * \dot{Y_C}^2 + 2 * \dot{X_D}^2 + 2 * \dot{Y_D}^2 \\ 2 * \dot{X_E}^2 + 2 * \dot{Y_E}^2 \\ 2 * \dot{X_F}^2 + 2 * \dot{Y_F}^2 \\ 2 * \dot{X_E}^2 + 2 * \dot{Y_E}^2 + 2 * \dot{X_F}^2 + 2 * \dot{Y_F}^2 \\ 2 * \dot{X_C}^2 + 2 * \dot{Y_C}^2 + 2 * \dot{X_E}^2 + 2 * \dot{Y_E}^2 \\ 2 * \dot{X_D}^2 + 2 * \dot{Y_D}^2 + 2 * \dot{X_G}^2 + 2 * \dot{Y_G}^2 \\ 2 * \dot{X_F}^2 + 2 * \dot{Y_F}^2 + 2 * \dot{X_G}^2 + 2 * \dot{Y_G}^2 \\ \dot{\theta}^2 * L_1 * cos\theta \\ 1 \end{bmatrix} \quad (4.62)$$

4.3.2 Implementación en Python

4.3.2.1 Problema de posición

Paso 1. Posición inicial

```
print ('MECANISMO DE STEPHENSON')
print ('============================')
# Lectura de datos por teclado

meca = {} # dictionary vacio, con propiedades de mecanismo

meca["L1"] = 3
meca["L2"] = 5
meca["L3"] = 4
meca["L4"] = 8
meca["L5"] = 6
meca["L6"] = 8
meca["L7"] = 3
meca["L8"] = 9
meca["L9"] = 8
meca["XA"] = 0
meca["XB"] = 7
meca["YA"] = 0
meca["YB"] = 0
meca["theta"] = float (input ('Introduce angulo inicial theta:')) #En rad


# Defino posicion inicial:
q = np.array ([[meca["XA"]+meca["L1"]*math.cos(meca["theta"])], [meca["YA"]+meca["L1"]*math.sin(meca["theta"])], [7], [7], [-3]
print('q: ' + str(q))
```

```
MECANISMO DE STEPHENSON
============================
Introduce angulo inicial theta:0.5
q: [[ 2.63274769]
 [ 1.43827662]
 [ 7.        ]
 [ 7.        ]
 [-3.        ]
 [ 5.        ]
 [ 8.        ]
 [10.        ]
 [ 5.        ]
 [10.        ]
 [ 0.5       ]]
```

Paso 2. Matriz de restricciones

```
def Phi (q,meca):
    #Inicializa a cero Phi
    Phi = np.zeros((11,1))

    #Extraer coordenadas
    XC = q[0]
    YC = q[1]
    XE = q[2]
    YE = q[3]
    XD = q[4]
    YD = q[5]
    XF = q[6]
    YF = q[7]
    XG = q[8]
    YG = q[9]
    theta = q[10]


    Phi[0] = XC**2 + YC**2 - meca["L1"]**2
    Phi[1] = XD**2 + YD**2 - meca["L2"]**2
    Phi[2] = (XC-XD)**2 + (YC-YD)**2 - meca["L3"]**2
    Phi[3] = (XE - meca["XB"])**2 + (YE - meca["YB"])**2 - meca["L5"]**2
    Phi[4] = (meca["XB"] - XF)**2 + (meca["YB"] - YF)**2 - meca["L6"]**2
    Phi[5] = (XF-XE)**2 + (YF-YE)**2 - meca["L7"]**2
    Phi[6] = (XE-XC)**2 + (YE-YC)**2 - meca["L4"]**2
    Phi[7] = (XG-XD)**2 + (YG-YD)**2 - meca["L9"]**2
    Phi[8] = (XF-XG)**2 + (YF-YG)**2 - meca["L8"]**2

    if (abs(math.cos(theta)) < (math.sqrt(2)/2) ):
        Phi[9] = XC-meca["L1"]*math.cos(theta)
    else:
        Phi[9] = YC-meca["L1"]*math.sin(theta)


    return Phi
```

Paso 3. Matriz Jacobiana

```
def jacob_Phiq(q,meca):
    #Inicializa a cero la matriz jacobiana
    Jacob = np.zeros((11,11))

    #Extraer coordenadas

    XC = q[0]
    YC = q[1]
    XE = q[2]
    YE = q[3]
    XD = q[4]
    YD = q[5]
    XF = q[6]
    YF = q[7]
    XG = q[8]
    YG = q[9]
    theta = q[10]

    #Montar matriz

    Jacob[0,0] = 2*XC
    Jacob[0,1] = 2*YC
    Jacob[1,4] = 2*XD
    Jacob[1,5] = 2*YD
    Jacob[2,0] = 2*(XC-XD)
    Jacob[2,1] = 2*(YC-YD)
    Jacob[2,4] = -2*(XC-XD)
    Jacob[2,5] = -2*(YC-YD)
    Jacob[3,2] = 2*(XE-meca["XB"])
    Jacob[3,3] = 2*(YE-meca["YB"])
    Jacob[4,6] = -2*(meca["XB"]-XF)
    Jacob[4,7] = -2*(meca["YB"]-YF)
    Jacob[5,2] = -2*(XF-XE)
    Jacob[5,3] = -2*(YF-YE)
    Jacob[5,6] = 2*(XF-XE)
    Jacob[5,7] = 2*(YF-YE)
    Jacob[6,0] = -2*(XE-XC)
    Jacob[6,1] = -2*(YE-YC)
    Jacob[6,2] = 2*(XE-XC)
    Jacob[6,3] = 2*(YE-YC)
    Jacob[7,4] = -2*(XG-XD)
    Jacob[7,5] = -2*(YG-YD)
    Jacob[7,8] = 2*(XG-XD)
    Jacob[7,9] = 2*(YG-YD)
    Jacob[8,6] = 2*(XF-XG)
    Jacob[8,7] = 2*(YF-YG)
    Jacob[8,8] = -2*(XF-XG)
    Jacob[8,9] = -2*(YF-YG)
    Jacob[10,10] = 1



    if (abs(math.cos(theta)) < (math.sqrt(2)/2) ):
        Jacob[9,10] = meca["L1"]*math.sin(theta)
        Jacob[9,0] = 1
    else:
        Jacob[9,10] = -meca["L1"]*math.cos(theta)
        Jacob[9,1] = 1

    return Jacob
```

Paso 4. Resolver problema de posición

```
def resuelve_prob_posicion(q_init, meca):
    #Inicializacion de variables
    error = 1e10
    tolerancia = 1e-10

    #Inicializacion en cero de deltaQ, fi y q
    deltaQ = np.zeros ((11,1))
    q = q_init
    i=0

    # Iteraciones hasta conseguir que el error sea menor que la tolerancia

    while (error > tolerancia and i<=100):
        print("q=")
        pprint.pprint(q)


        #Extraer las coordenadas

        XC = q[0]
        YC = q[1]
        XE = q[2]
        YE = q[3]
        XD = q[4]
        YD = q[5]
        XF = q[6]
        YF = q[7]
        XG = q[8]
        YG = q[9]
        theta = q[10]


        fi=Phi(q,meca)
        print ("Phi" + "=")
        pprint.pprint(fi)
        J = jacob_Phiq(q,meca)
        print ("jacob" + "=")
        pprint.pprint(J)
        rango = np.linalg.matrix_rank(J, 1e-5)
        print("rango=" + str(rango) + "\n")
        deltaQ = np.linalg.solve(J,-fi)
        q = q + deltaQ
        error = np.linalg.norm(deltaQ)
        i=i+1

        print("error iter" + str(i) + "=")
        pprint.pprint(error)
    print("num iters:" + str(i))
    if (error > tolerancia):
        raise Exception ('No se puede alcanzar la posición')
    return q

q=resuelve_prob_posicion(q,meca)
```

```
q=
array([[ 2.63274769],
       [ 1.43827662],
       [ 9.57066058],
       [ 5.42141164],
       [ 0.71504553],
       [ 4.94860686],
       [12.55145541],
       [ 5.76032489],
       [ 5.50346632],
       [11.35727493],
       [ 0.5       ]])
Phi=
array([[ 0.00000000e+00],
       [ 0.00000000e+00],
       [ 3.55271368e-15],
       [ 0.00000000e+00],
       [ 0.00000000e+00],
       [ 5.32907052e-15],
       [ 0.00000000e+00],
       [-1.42108547e-14],
       [ 0.00000000e+00],
       [ 0.00000000e+00],
       [ 0.00000000e+00]])
jacob=
array([[  5.26549537,   2.87655323,   0.        ,   0.        ,
          0.        ,   0.        ,   0.        ,   0.        ,
          0.        ,   0.        ,   0.        ],
       [  0.        ,   0.        ,   0.        ,   0.        ,
          1.43009106,   9.89721373,   0.        ,   0.        ,
          0.        ,   0.        ,   0.        ],
       [  3.83540431,  -7.0206605 ,   0.        ,   0.        ,
         -3.83540431,   7.0206605 ,   0.        ,   0.        ,
          0.        ,   0.        ,   0.        ],
       [  0.        ,   0.        ,   5.14132117,  10.84282328,
          0.        ,   0.        ,   0.        ,   0.        ,
          0.        ,   0.        ,   0.        ],
       [  0.        ,   0.        ,   0.        ,   0.        ,
          0.        ,   0.        ,  11.10291083,  11.52064977,
          0.        ,   0.        ,   0.        ],
       [  0.        ,   0.        ,  -5.96158966,  -0.6778265 ,
          0.        ,   0.        ,   5.96158966,   0.6778265 ,
          0.        ,   0.        ,   0.        ],
       [-13.8758258 ,  -7.96627005,  13.8758258 ,   7.96627005,
          0.        ,   0.        ,   0.        ,   0.        ,
          0.        ,   0.        ,   0.        ],
       [  0.        ,   0.        ,   0.        ,   0.        ,
         -9.57684158, -12.81733613,   0.        ,   0.        ,
          9.57684158,  12.81733613,   0.        ],
       [  0.        ,   0.        ,   0.        ,   0.        ,
          0.        ,   0.        ,  14.09597819, -11.19390008,
        -14.09597819,  11.19390008,   0.        ],
       [  0.        ,   1.        ,   0.        ,   0.        ,
          0.        ,   0.        ,   0.        ,   0.        ,
          0.        ,   0.        ,  -2.63274769],
       [  0.        ,   0.        ,   0.        ,   0.        ,
          0.        ,   0.        ,   0.        ,   0.        ,
          0.        ,   0.        ,   1.        ]])
rango=11

error iter8=
2.4210370327928987e-15
num iters:8
```

Paso 5. Representación y diseño del mecanismo

```
def dibuja_mecanismo(q, meca):


    # Extraer los puntos moviles del mecanismo

    XC = q[0]
    YC = q[1]
    XE = q[2]
    YE = q[3]
    XD = q[4]
    YD = q[5]
    XF = q[6]
    YF = q[7]
    XG = q[8]
    YG = q[9]
    theta = q[10]


    plt.axis('equal')

    plt.plot ([meca["XA"], XC], [meca["XA"], YC])
    plt.plot ([XD, meca["XA"]], [YD, meca["YA"]])
    plt.plot ([XC, XD], [YC, YD])
    plt.plot ([XE, XC], [YE, YC])
    plt.plot([meca["XB"], XE], [meca["YB"], YE])
    plt.fill([meca["XB"], XE], [meca["YB"], YE], facecolor='lightsalmon')
    plt.plot([XF, meca["XB"]], [YF, meca["YB"]])
    plt.plot ([XE, XF], [YE, YF])
    plt.plot ([XD, XG], [YD, YG])
    plt.plot ([XF, XG], [YF, YG])


    plt.plot(meca["XA"], meca["YA"], 'bo', marker="^", markersize=10)
    plt.plot(meca["XB"], meca["YB"], 'go', marker="^", markersize=10)
    plt.plot(XC, YC, color='r', marker='o')
    plt.plot(XD, YD, color='grey', marker='o')
    plt.plot(XE, YE, color='y', marker='o')
    plt.plot(XF, YF, color='c', marker='o')
    plt.plot(XG, YG, color='m', marker='o')



    plt.show()#block=False)
    return

dibuja_mecanismo(q,meca)
```

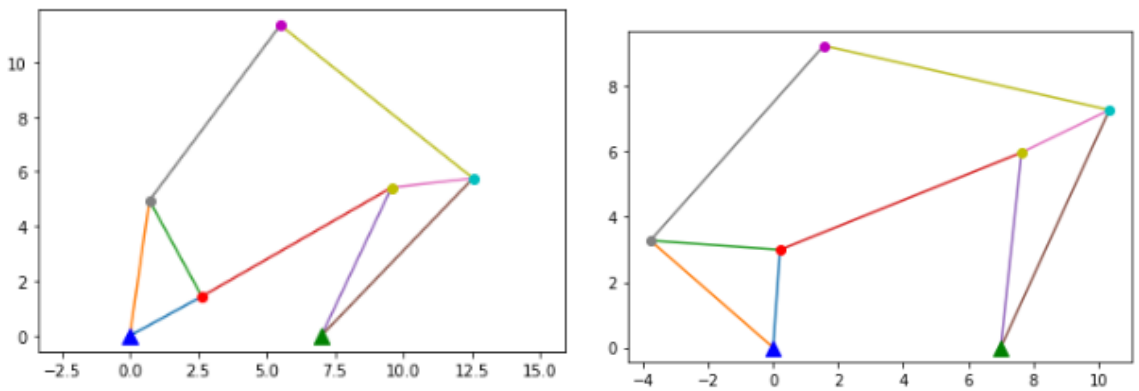


Figura 4. 19. Representación gráfica Stephenson

4.3.2.2 Problema de velocidad

El primer paso será abrir un nuevo notebook y copiar el código mediante el cual se elaboró el problema de posición.

Posteriormente introducimos la ecuación matricial que corresponde a la resolución del problema de velocidad.

```
#PROBLEMA DE VELOCIDAD

def resuelve_prob_velocidad(q,qp, meca):

    qp = np.linalg.solve(jacob_Phiq(q,meca),qp)

    return qp


qp = np.zeros ((11,1))
    #Velocidad del gdl.
qp[10]=1
qp=resuelve_prob_velocidad (q,qp, meca)

qp
```

```
array([[-1.43827662],
       [ 2.63274769],
       [ 0.10059968],
       [-0.04770116],
       [-4.94860686],
       [ 0.71504553],
       [ 0.10688855],
       [-0.10301277],
       [-1.36801691],
       [-1.9602953 ],
       [ 1.        ]])
```

4.3.2.3 Problema de aceleración

En este apartado es necesario introducir la matriz calculada en la parte teórica $\dot{\Phi}_q * \dot{q}$, que llamaremos "b".

```
#PROBLEMA DE ACELERACION

def resuelve_prob_aceleracion (q,qp,qpp,meca):

    #Extraer las coordenadas

    XC = q[0]
    YC = q[1]
    XE = q[2]
    YE = q[3]
    XD = q[4]
    YD = q[5]
    XF = q[6]
    YF = q[7]
    XG = q[8]
    YG = q[9]
    theta = q[10]
    #Extraer las velocidades

    XCq = qp[0]
    YCq = qp[1]
    XEq = qp[2]
    YEq = qp[3]
    XDq = qp[4]
    YDq = qp[5]
    XFq = qp[6]
    YFq = qp[7]
    XGq = qp[8]
    YGq = qp[9]
    thetaq = qp[10]
    b=qpp
```

```
    b[0] = 2*(XCq)**2 + 2*(YCq)**2
    b[1] = 2*(XDq)**2 + 2*(YDq)**2
    b[2] = 2*(XCq)**2 + 2*(YCq)**2 + 2*(XDq)**2 + 2*(YDq)**2
    b[3] = 2*(XEq)**2 + 2*(YEq)**2
    b[4] = 2*(XCq)**2 + 2*(YCq)**2
    b[5] = 2*(XEq)**2 + 2*(YEq)**2 + 2*(XFq)**2 + 2*(YFq)**2
    b[6] = 2*(XCq)**2 + 2*(YCq)**2 + 2*(XEq)**2 + 2*(YEq)**2
    b[7] = 2*(XDq)**2 + 2*(YDq)**2 + 2*(XGq)**2 + 2*(YGq)**2
    b[8] = 2*(XFq)**2 + 2*(YFq)**2 + 2*(XGq)**2 + 2*(YGq)**2


    if (abs(math.cos(theta)) < (math.sqrt(2)/2) ):
        b[9] = thetaq**2 * (meca["L1"] * math.cos(theta))
    else:
        b[9] =  thetaq**2 * (meca["L1"] * math.sin(theta))


    qpp = np.linalg.solve(jacob_Phiq(q,meca),-b)


    return qpp

qpp=np.zeros ((11,1))
qpp[10] = 0 #Aceleracion conocida
qpp = resuelve_prob_aceleracion(q,qp, qpp,meca)
qpp
```

```
array([[-2.63274769],
       [-1.43827662],
       [-6.53522721],
       [ 3.09650999],
       [ 6.70786476],
       [-6.02117516],
       [-6.75752861],
       [ 4.95008864],
       [-9.08898993],
       [ 0.98929239],
       [ 0.        ]])
```

4.3.2.4 Gráfica de velocidades y aceleraciones

El método implementado es el mismo que para los mecanismos anteriores. De la misma manera obtenemos nuestras gráficas de velocidades y aceleraciones.

```
#PASO 4: GRÁFICAS DE VELOCIDADES
def grafica_velocidad(q,meca):

    th = np.linspace(0,2*3.1416,200)

    VXC = np.zeros((50,0))
    VYC = np.zeros((50,0))
    VXE = np.zeros((50,0))
    VYE = np.zeros((50,0))
    VXD = np.zeros((50,0))
    VYD = np.zeros((50,0))
    VXF = np.zeros((50,0))
    VYF = np.zeros((50,0))
    VXG = np.zeros((50,0))
    VYG = np.zeros((50,0))

    i=0
    for t in th:

        q[10] = t
        q = resuelve_prob_posicion (q,meca)

        qp = np.zeros((11,1))
        qp[10] = 1  #inicializar qp en 0 con qp[10] = 1 rad/s
        qp = resuelve_prob_velocidad(q, qp, meca)


        VXC = np.append(VXC, qp[0])
        VYC = np.append(VYC, qp[1])
        VXE = np.append(VXE, qp[2])
        VYE = np.append(VYE, qp[3])
        VXD = np.append(VXD, qp[4])
        VYD = np.append(VYD, qp[5])
        VXF = np.append(VXF, qp[6])
        VYF = np.append(VYF, qp[7])
        VXG = np.append(VXG, qp[8])
        VYG = np.append(VYG, qp[9])
        i=i+1
```

```
fig, axs = plt.subplots(ncols=2, figsize=(20, 20))
plt.subplot(3,4,1)
plt.plot(th,VXC)
plt.xlabel ('Angulo')
plt.ylabel ('Velocidad')
plt.title ('Velocidad de XC')

plt.subplot(3,4,2)
plt.plot(th,VYC)
plt.xlabel ('Angulo')
plt.ylabel ('Velocidad')
plt.title ('Velocidad de YC')

plt.subplot(3,4,3)
plt.plot(th,VXE)
plt.xlabel ('Angulo')
plt.ylabel ('Velocidad')
plt.title ('Velocidad de XE')

plt.subplot(3,4,4)
plt.plot(th,VYE)
plt.xlabel ('Angulo')
plt.ylabel ('Velocidad')
plt.title ('Velocidad de YE')

plt.subplot(3,4,5)
plt.plot(th,VXD)
plt.xlabel ('Angulo')
plt.ylabel ('Velocidad')
plt.title ('Velocidad de XD')

plt.subplot(3,4,6)
plt.plot(th,VYD)
plt.xlabel ('Angulo')
plt.ylabel ('Velocidad')
plt.title ('Velocidad de YD')
```

```
    plt.subplot(3,4,7)
    plt.plot(th,VXF)
    plt.xlabel ('Angulo')
    plt.ylabel ('Velocidad')
    plt.title ('Velocidad de XF')

    plt.subplot(3,4,8)
    plt.plot(th,VYF)
    plt.xlabel ('Angulo')
    plt.ylabel ('Velocidad')
    plt.title ('Velocidad de YF')

    plt.subplot(3,4,9)
    plt.plot(th,VXG)
    plt.xlabel ('Angulo')
    plt.ylabel ('Velocidad')
    plt.title ('Velocidad de XG')

    plt.subplot(3,4,10)
    plt.plot(th,VYG)
    plt.xlabel ('Angulo')
    plt.ylabel ('Velocidad')
    plt.title ('Velocidad de YG')


    plt.show()
    return

grafica_velocidad(q,meca)
```

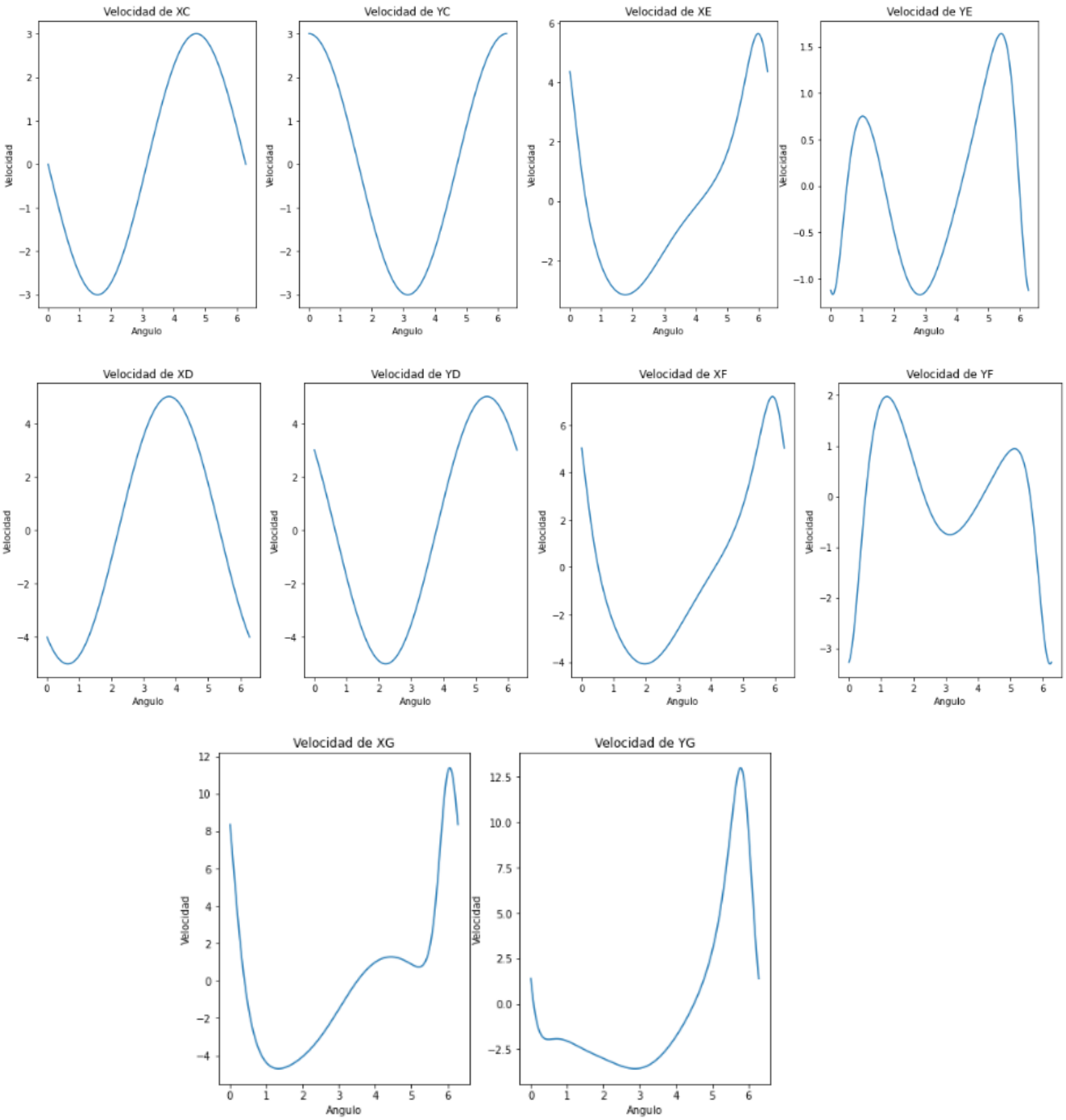


Figura 4. 20. Gráficas velocidades Stephenson

Para el mecanismo de Stephenson se puede apreciar que no hay picos altos ni bajos de velocidad como en los mecanismos anteriores, salvo en el punto G ya que es el extremo del mecanismo y donde más va variar esta.

```
#GRÁFICAS ACELERACIONES

def grafica_aceleracion(q,meca):

    th = np.linspace(0,2*3.1416,200)

    AXC = np.zeros((50,0))
    AYC = np.zeros((50,0))
    AXE = np.zeros((50,0))
    AYE = np.zeros((50,0))
    AXD = np.zeros((50,0))
    AYD = np.zeros((50,0))
    AXF = np.zeros((50,0))
    AYF = np.zeros((50,0))
    AXG = np.zeros((50,0))
    AYG = np.zeros((50,0))

    i=0
    for t in th:

        q[10] = t
        q = resuelve_prob_posicion (q,meca)

        qp = np.zeros((11,1))
        qp[10] = 1  #inicializar qp en 0 con qp[10] = 1 rad/s
        qp = resuelve_prob_velocidad(q, qp, meca)

        qpp = np.zeros((11,1))
        qpp[10] = 0  #inicializar qp en 0 con qpp[4] = 1 rad/s**2
        qpp = resuelve_prob_aceleracion(q,qp, qpp, meca)


        AXC = np.append(AXC, qpp[0])
        AYC = np.append(AYC, qpp[1])
        AXE = np.append(AXE, qpp[2])
        AYE = np.append(AYE, qpp[3])
        AXD = np.append(AXD, qpp[4])
        AYD = np.append(AYD, qpp[5])
        AXF = np.append(AXF, qpp[6])
        AYF = np.append(AYF, qpp[7])
        AXG = np.append(AXG, qpp[8])
        AYG = np.append(AYG, qpp[9])
        i=i+1
```

```
fig, axs = plt.subplots(ncols=2, figsize=(20, 20))
plt.subplot(3,4,1)
plt.plot(th,AXC)
plt.xlabel ('Angulo')
plt.ylabel ('Aceleración')
plt.title ('Aceleración de XC')

plt.subplot(3,4,2)
plt.plot(th,AYC)
plt.xlabel ('Angulo')
plt.ylabel ('Aceleración')
plt.title ('Aceleración de YC')

plt.subplot(3,4,3)
plt.plot(th,AXE)
plt.xlabel ('Angulo')
plt.ylabel ('Aceleración')
plt.title ('Aceleración de XE')

plt.subplot(3,4,4)
plt.plot(th,AYE)
plt.xlabel ('Angulo')
plt.ylabel ('Aceleración')
plt.title ('Aceleración de YE')

plt.subplot(3,4,5)
plt.plot(th,AXD)
plt.xlabel ('Angulo')
plt.ylabel ('Aceleración')
plt.title ('Aceleración de XD')

plt.subplot(3,4,6)
plt.plot(th,AYD)
plt.xlabel ('Angulo')
plt.ylabel ('Aceleración')
plt.title ('Aceleración de YD')
```

```
    plt.subplot(3,4,7)
    plt.plot(th,AXF)
    plt.xlabel ('Angulo')
    plt.ylabel ('Aceleración')
    plt.title ('Aceleración de XF')

    plt.subplot(3,4,8)
    plt.plot(th,AYF)
    plt.xlabel ('Angulo')
    plt.ylabel ('Aceleración')
    plt.title ('Aceleración de YF')

    plt.subplot(3,4,9)
    plt.plot(th,AXG)
    plt.xlabel ('Angulo')
    plt.ylabel ('Aceleración')
    plt.title ('Aceleración de XG')

    plt.subplot(3,4,10)
    plt.plot(th,AYG)
    plt.xlabel ('Angulo')
    plt.ylabel ('Aceleración')
    plt.title ('Aceleración de YG')


    plt.show()
    return

grafica_aceleracion(q,meca)
```

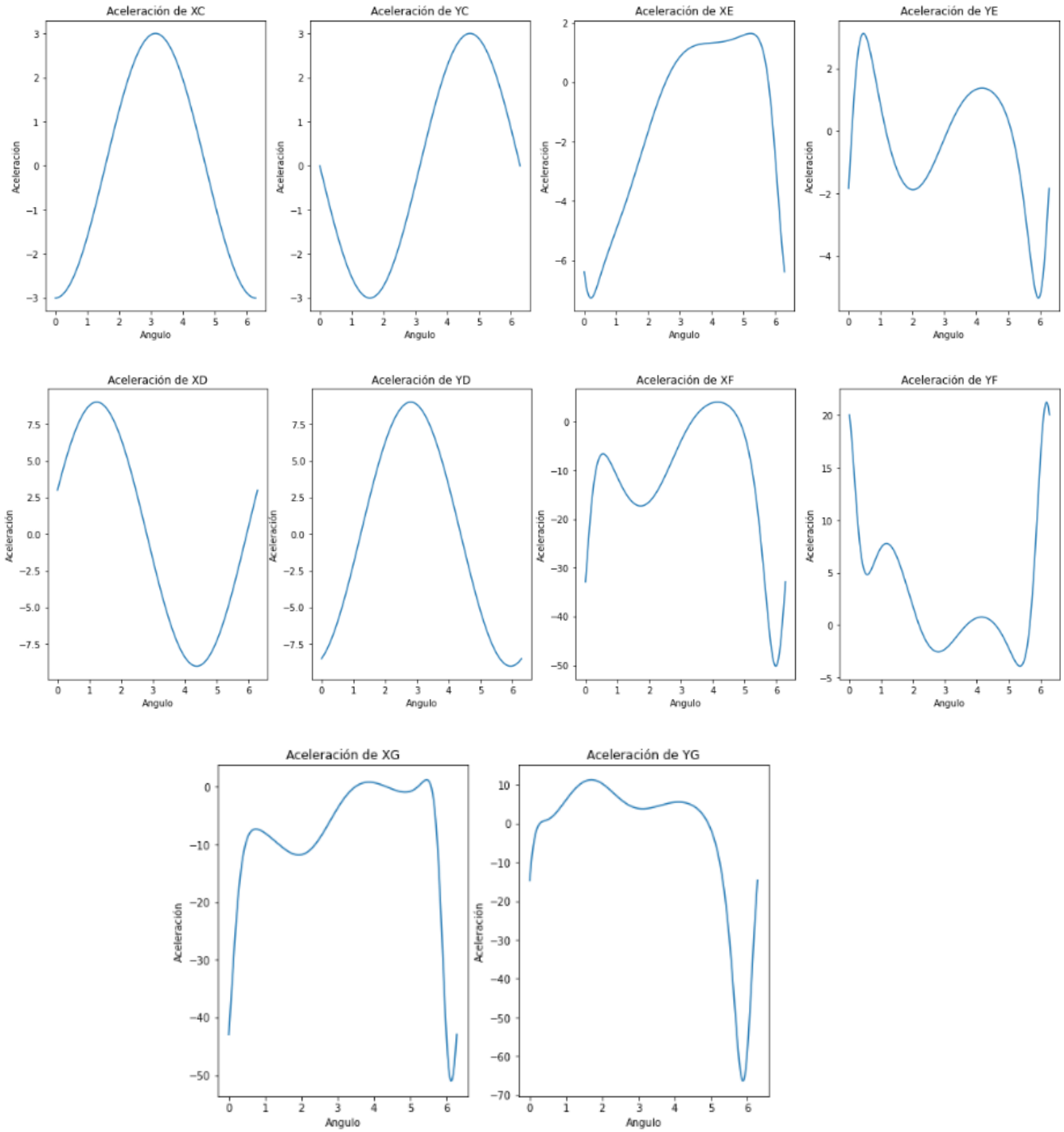


Figura 4. 21. Gráficas de aceleraciones Stephenson

4.3.2.5 Animación

Utilizando la librería matplotlib.animation y HTML creamos la animación del mecanismo de Stephenson al igual que en los problemas anteriores.

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt

from matplotlib import animation, rc
from IPython.display import HTML

fig, ax = plt.subplots()

ax.set_xlim(( -10, 20))
ax.set_ylim((-10,20))
ax.set_aspect('equal')

line, = ax.plot([], [], lw=4, marker='o', markersize=6)

last_q = q

def init():
    line.set_data([], [])
    return (line,)

def animate(i,q,meca):
    global last_q
    q = last_q
    omega=2*3.14159/200  # vel. angular
    q[10] = i*omega

    #Llamar problema de pos:
    q = resuelve_prob_posicion(q, meca)
    last_q = q

    # Extraer los puntos moviles del mecanismo

    XC = q[0]
    YC = q[1]
    XE = q[2]
    YE = q[3]
    XD = q[4]
    YD = q[5]
    XF = q[6]
    YF = q[7]
    XG = q[8]
    YG = q[9]
    theta = q[10]


    x=[meca["XA"], XC, meca["XA"], XD, XC, XE, meca["XB"], XF, XE, XF, XG, XD, XC]
    y=[meca["YA"], YC, meca["YA"], YD, YC, YE, meca["YB"], YF, YE, YF, YG, YD, YC]


    line.set_data(x, y)
    return (line,)


anim = animation.FuncAnimation(fig, animate, init_func=init, fargs=(q,meca),
                               frames=200, interval=20,
                               blit=True)

HTML(anim.to_html5_video())
```

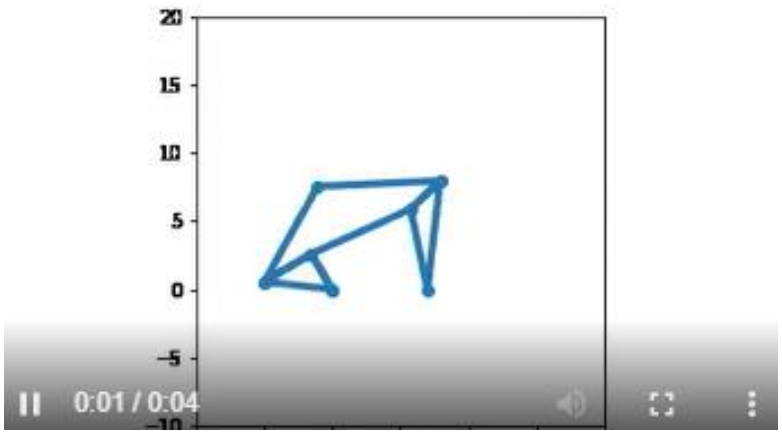


Figura 4. 22. Animación mecanismo de Stephenson

4.4 Mecanismo Biela-manivela corredera

El mecanismo biela-manivela corredera un mecanismo plano compuesto por 4 sólidos rígidos conectados entre sí mediante 3 pares cinemáticos de revolución y un par prismático de deslizamiento.

Los 4 sólidos rígidos que componen este mecanismo se denominan:

- Barra fija: es el sólido que permanece inmóvil y representa la estructura fija que soporta a los elementos móviles.
- Manivela: sólido dotado de un movimiento de rotación que está conectado a la barra fija mediante un par de revolución.
- Deslizadera: sólido dotado de un movimiento de traslación pura conectado a la barra fija mediante un par prismático de traslación.
- Biela: sólido que conecta el extremo e la manivela con la deslizadera.

El mecanismo biela manivela se utiliza para transformar el movimiento de rotación (manivela) en un movimiento de traslación (biela) o viceversa.

Es el mecanismo más popular en el mundo debido, fundamentalmente, en su utilización en motores de combustión interna (pistones).

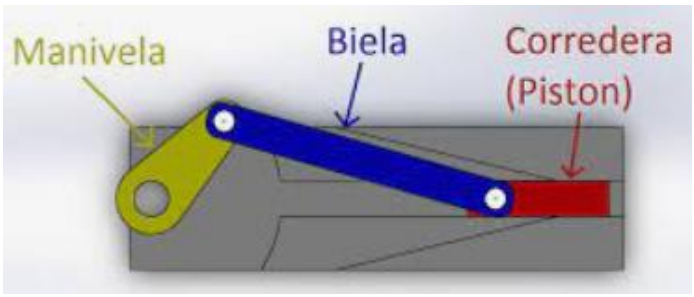


Figura 4. 23. Mecanismo biela-manivela corredera

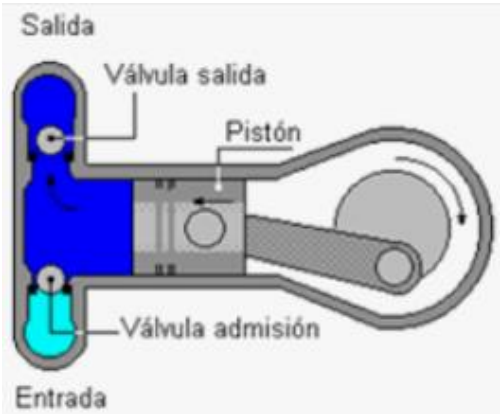


Figura 4. 24. Pistón con mecanismo biela-manivela corredera

4.4.1 Cálculo Teórico

4.4.1.1 Problema de posición

Paso 1. Modelado del mecanismo

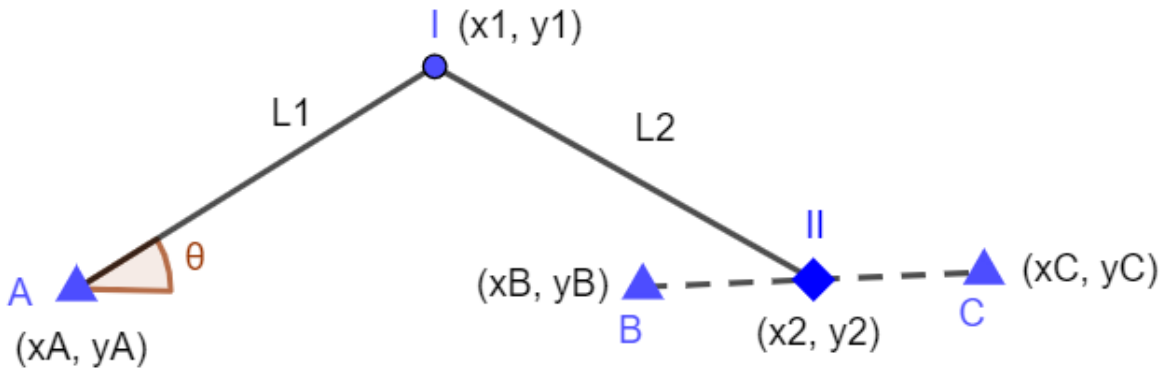


Figura 4. 25. Modelado biela-manivela corredera

Paso 2. Grados de libertad

$$G = 3 * (N - 1) - 2P_I - P_{II} = 3 * (4 - 1) - 2 * 4 = 1$$

Paso 3. Definición del vector q

Las coordenadas que definen el vector q para el biela-manivela corredera son:

$$q = \begin{bmatrix} x1 \\ y1 \\ x2 \\ y2 \\ \theta \end{bmatrix} \qquad (4.63)$$

Parámetros iniciales:

$$L1 = 3$$

$$L2 = 1$$

$$\theta = 0.5$$

$$X_A = 0$$

$$Y_A = 0$$

$$X_B = 2$$

$$Y_B = 0$$

$$X_C = 3$$

$$Y_C = 0$$

Paso 4. Matriz de restricciones

En este caso hemos especificado las ecuaciones de restricción ya simplificadas sabiendo que el punto A es el origen.

- Sólido A-1:

$$X_1{}^2 + Y_1{}^2 - L_1{}^2 = 0 \qquad (4.64)$$

- Sólido 1-2:

$$(X_2 - X_1)^2 + (Y_2 - Y_1)^2 - L_2{}^2 = 0 \qquad (4.65)$$

- Par prismático entre el punto 2 y la barra B-C:

$$(X_C - X_B) * (Y_2 - Y_B) - (Y_C - Y_B) * (X_2 - X_B) = 0 \qquad (4.66)$$

- Coordenada relativa θ:

$$X_1 - L_1 * cos\theta = 0 \qquad (4.67)$$

$$Y_1 - L_1 * sin\theta = 0 \qquad (4.68)$$

La matriz de restricciones queda de la siguiente manera:

-Si $cos\theta < \frac{1}{\sqrt{2}}$:

$$\Phi = \begin{bmatrix} X_1{}^2 + Y_1{}^2 - L_1{}^2 \\ (X_2 - X_1)^2 + (Y_2 - Y_1)^2 - L_2{}^2 \\ (X_C - X_B) * (Y_2 - Y_B) - (Y_C - Y_B) * (X_2 - X_B) \\ X_1 - L_1 * cos\theta \end{bmatrix} \qquad (4.69)$$

-Si $cos\theta > \frac{1}{\sqrt{2}}$:

$$\Phi = \begin{bmatrix} X_1{}^2 + Y_1{}^2 - L_1{}^2 \\ (X_2 - X_1)^2 + (Y_2 - Y_1)^2 - L_2{}^2 \\ (X_C - X_B) * (Y_2 - Y_B) - (Y_C - Y_B) * (X_2 - X_B) \\ Y_1 - L_1 * sin\theta \end{bmatrix} \qquad (4.70)$$

Paso 5. Matriz Jacobiana

Se procede a calcular la matriz Jacobiana derivando la matriz de restricciones Φ respecto del vector *q.*

-Si $cos\theta < \frac{1}{\sqrt{2}}$:

$$\Phi_q = \begin{bmatrix} 2*X_1 & 2*Y_1 & 0 & 0 & 0 \\ -2*(X_2-X_1) & -2*(Y_2-Y_1) & 2*(X_2-X_1) & 2*(Y_2-Y_1) & 0 \\ 0 & 0 & -(Y_C-Y_B) & X_C-X_B & 0 \\ 1 & 0 & 0 & 0 & L_1*sin\theta \end{bmatrix} \quad (4.71)$$

-Si $cos\theta > \frac{1}{\sqrt{2}}$:

$$\Phi_q = \begin{bmatrix} 2*X_1 & 2*Y_1 & 0 & 0 & 0 \\ -2*(X_2-X_1) & -2*(Y_2-Y_1) & 2*(X_2-X_1) & 2*(Y_2-Y_1) & 0 \\ 0 & 0 & -(Y_C-Y_B) & X_C-X_B & 0 \\ 0 & 1 & 0 & 0 & -L_1*cos\theta \end{bmatrix} \quad (4.72)$$

Paso 6. Resolución del problema de posición

Como en el resto de mecanismo, definimos el valor del ángulo θ como dato de partida para poder resolver el problema de posición (3.15).

$$\Phi_q * \Delta q = -\Phi$$

-Si $cos\theta < \frac{1}{\sqrt{2}}$:

$$\begin{bmatrix} 2*X_1 & 2*Y_1 & 0 & 0 & 0 \\ -2*(X_2-X_1) & -2*(Y_2-Y_1) & 2*(X_2-X_1) & 2*(Y_2-Y_1) & 0 \\ 0 & 0 & -(Y_C-Y_B) & X_C-X_B & 0 \\ 1 & 0 & 0 & 0 & L_1*sin\theta \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x1_1-x1_0 \\ y1_1-y1_0 \\ x2_1-x2_0 \\ y2_1-y2_0 \\ \theta_1-\theta_0 \end{bmatrix} =$$

$$= -\begin{bmatrix} {X_1}^2+{Y_1}^2-{L_1}^2 \\ (X_2-X_1)^2+(Y_2-Y_1)^2-{L_2}^2 \\ (X_C-X_B)*(Y_2-Y_B)-(Y_C-Y_B)*(X_2-X_B) \\ X_1-L_1*cos\theta \\ 0 \end{bmatrix} \quad (4.73)$$

-Si $cos\theta > \frac{1}{\sqrt{2}}$:

$$\begin{bmatrix} 2*X_1 & 2*Y_1 & 0 & 0 & 0 \\ -2*(X_2-X_1) & -2*(Y_2-Y_1) & 2*(X_2-X_1) & 2*(Y_2-Y_1) & 0 \\ 0 & 0 & -(Y_C-Y_B) & X_C-X_B & 0 \\ 0 & 1 & 0 & 0 & -L_1*cos\theta \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x1_1-x1_0 \\ y1_1-y1_0 \\ x2_1-x2_0 \\ y2_1-y2_0 \\ \theta_1-\theta_0 \end{bmatrix} =$$

$$= -\begin{bmatrix} {X_1}^2+{Y_1}^2-{L_1}^2 \\ (X_2-X_1)^2+(Y_2-Y_1)^2-{L_2}^2 \\ (X_C-X_B)*(Y_2-Y_B)-(Y_C-Y_B)*(X_2-X_B) \\ Y_1-L_1*sin\theta \\ 0 \end{bmatrix} \quad (4.74)$$

4.4.1.2 Problema de velocidad

El dato del que partimos para resolver el problema de velocidad (3.20) biela-manivela corredera será el de la velocidad angular $\dot{\theta} = 1$.

$$\Phi_q * \dot{q} = -\Phi_t$$

-Si $cos\theta < \frac{1}{\sqrt{2}}$:

$$\begin{bmatrix} 2*X_1 & 2*Y_1 & 0 & 0 & 0 \\ -2*(X_2-X_1) & -2*(Y_2-Y_1) & 2*(X_2-X_1) & 2*(Y_2-Y_1) & 0 \\ 0 & 0 & -(Y_C-Y_B) & X_C-X_B & 0 \\ 1 & 0 & 0 & 0 & L_1*sin\theta \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} \dot{x1} \\ \dot{y1} \\ \dot{x2} \\ \dot{y2} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \quad (4.75)$$

-Si $cos\theta > \frac{1}{\sqrt{2}}$:

$$\begin{bmatrix} 2*X_1 & 2*Y_1 & 0 & 0 & 0 \\ -2*(X_2-X_1) & -2*(Y_2-Y_1) & 2*(X_2-X_1) & 2*(Y_2-Y_1) & 0 \\ 0 & 0 & -(Y_C-Y_B) & X_C-X_B & 0 \\ 0 & 1 & 0 & 0 & -L_1*cos\theta \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} \dot{x1} \\ \dot{y1} \\ \dot{x2} \\ \dot{y2} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \quad (4.76)$$

4.4.1.3 Problema de aceleración

$$\Phi_q * \ddot{q} = -\dot{\Phi}_q * \dot{q} - \dot{\Phi}_t$$

Para la obtención de la derivada del jacobiano se ha procedido de la misma manera que con el resto de mecanismos:

$$\dot{\Phi}_q = \begin{bmatrix} 2*\dot{X_1} & 2*\dot{Y_1} & 0 & 0 & 0 \\ 0 & 0 & 2*\dot{X_2} & 2*\dot{Y_2} & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & \dot{\theta}*L_1*cos\theta \end{bmatrix} \quad (4.77)$$

De nuevo esta matriz la multiplicamos por el vector de velocidades para simplificar el sistema y añadimos el valor de la aceleración $\ddot{\theta} = 1$:

$$\dot{\Phi}_q * \dot{q} = \begin{bmatrix} 2*\dot{X_1}^2 + 2*\dot{Y_1}^2 \\ 2*\dot{X_2}^2 + 2*\dot{Y_2}^2 \\ 0 \\ \dot{\theta}^2 * L_1 * cos\theta \\ 1 \end{bmatrix} \quad (4.78)$$

Y de esta manera nos queda el siguiente sistema:

$$\begin{bmatrix} 2*X_1 & 2*Y_1 & 0 & 0 & 0 \\ -2*(X_2-X_1) & -2*(Y_2-Y_1) & 2*(X_2-X_1) & 2*(Y_2-Y_1) & 0 \\ 0 & 0 & -(Y_C-Y_B) & X_C-X_B & 0 \\ 1 & 0 & 0 & 0 & L_1*sin\theta \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} \ddot{x}1 \\ \ddot{y}1 \\ \ddot{x}2 \\ \ddot{y}2 \\ \ddot{\theta} \end{bmatrix} =$$

$$= \begin{bmatrix} 2*\dot{X_1}^2 + 2*\dot{Y_1}^2 \\ 2*\dot{X_2}^2 + 2*\dot{Y_2}^2 \\ 0 \\ \dot{\theta}^2 * L_1 * cos\theta \\ 1 \end{bmatrix} \qquad (4.79)$$

4.4.2 Implementación en Python

4.4.2.1 Problema de posición

Paso 1. Posición inicial

```
import numpy as np
import math as math
import pprint
import matplotlib.pyplot as plt
import scipy.integrate as integrate
```

```
print ('BIELA-MANIVELA')
print ('============================')


meca["L1"] = float (input ('Introduce longitud L1:'))
meca["L2"] = float (input ('Introduce longitud L2:'))
meca["theta"] = float (input ('Introduce angulo inicial theta:')) #En rad
meca["XB"] = float (input ('Introduce coordenada en x del punto B:'))
meca["XC"] = float (input ('Introduce coordenada en x del punto C:'))
meca["XA"] = 0
meca["YA"] = 0
meca["YB"] = float(input('Introduce coordenada en y del punto B: '))
meca["YC"] = float(input('Introduce coordenada en y del punto C: '))

# Defino posicion inicial:
q = np.array ([[0.1], [meca["L1"]], [1], [0.2], [meca["theta"]]])
print('q: ' + str(q))
```

```
BIELA-MANIVELA
============================
Introduce longitud L1:1
Introduce longitud L2:2
Introduce angulo inicial theta:0.5
Introduce coordenada en x del punto B:2
Introduce coordenada en x del punto C:3
Introduce coordenada en y del punto B: 0
Introduce coordenada en y del punto C: 0
q: [[0.1]
 [1. ]
 [1. ]
 [0.2]
 [0.5]]
```

Paso 2. Matriz de restricciones

```
def Phi (q,meca):
    #Inicializa a cero Phi
    Phi = np.zeros((5,1))

    #Extraer coordenadas
    X1 = q[0]
    Y1 = q[1]
    X2 = q[2]
    Y2 = q[3]
    theta = q[4]

    Phi[0] = X1**2 + Y1**2 - meca["L1"]**2
    Phi[1] = (X2-X1)**2 + (Y2-Y1)**2 -meca["L2"]**2
    Phi[2] = (X2 - meca["XB"])*(meca["YC"]-meca["YB"]) - (Y2-meca["YB"])*(meca["XC"]-meca["XB"])


    if (abs(math.cos(theta)) < (math.sqrt(2)/2) ):
        Phi[3] = X1-meca["L1"]*math.cos(theta)
    else:
        Phi[3] = Y1-meca["L1"]*math.sin(theta)



    return Phi
```

Paso 3. Matriz Jacobiana

```
def jacob_Phiq(q,meca):
    #Inicializa a cero la matriz jacobiana
    Jacob = np.zeros((5,5))

    #Extraer coordenadas

    X1 = q[0]
    Y1 = q[1]
    X2 = q[2]
    Y2 = q[3]
    theta = q[4]

    #Montar matriz

    Jacob[0,0] = 2*X1
    Jacob[0,1] = 2*Y1
    Jacob[1,0] = -2*(X2-X1)
    Jacob[1,1] = -2*(Y2-Y1)
    Jacob[1,2] = 2*(X2-X1)
    Jacob[1,3] = 2*(Y2-Y1)
    Jacob[2,2] = -(meca["YC"] - meca["YB"])
    Jacob[2,3] = meca["XC"] - meca["XB"]

    if (abs(math.cos(theta)) < (math.sqrt(2)/2) ):
        Jacob[3,4] = meca["L1"]*math.sin(theta)
        Jacob[3,0] = 1
    else:
        Jacob[3,4] = -meca["L1"]*math.cos(theta)
        Jacob[3,1] = 1


    Jacob[4,4] = 1

    return Jacob
```

Paso 4. Resolver problema de posición

```
def resuelve_prob_posicion(q_init, meca):
    #Inicializacion de variables
    error = 1e10
    tolerancia = 1e-10

    deltaQ = np.zeros ((5,1))
    q = q_init
    i=0



    while (error > tolerancia and i<=100):
        print("q=")
        pprint.pprint(q)


        #Extraer las coordenadas
        X1 = q[0]
        Y1 = q[1]
        X2 = q[2]
        Y2 = q[3]
        theta = q[4]

        fi=Phi(q,meca)
        print ("Phi" + "=")
        pprint.pprint(fi)
        J = jacob_Phiq(q,meca)
        print ("jacob" + "=")
        pprint.pprint(J)
        rango = np.linalg.matrix_rank(J, 1e-5)
        print("rango=" + str(rango) + "\n")

        deltaQ = np.linalg.solve(J,-fi)
        q = q + deltaQ
        error = np.linalg.norm(deltaQ)
        i=i+1


        print("error iter" + str(i) + "=")
        pprint.pprint(error)
    print("num iters:" + str(i))
    if (error > tolerancia):
        raise Exception ('No se puede alcanzar la posición')
    return q

q=resuelve_prob_posicion(q,meca)
```

```
q=
array([[0.87758256],
       [0.47942554],
       [2.81927027],
       [0.        ],
       [0.5       ]])
Phi=
array([[-1.11022302e-16],
       [-8.88178420e-16],
       [ 0.00000000e+00],
       [ 0.00000000e+00],
       [ 0.00000000e+00]])
jacob=
array([[ 1.75516512,  0.95885108,  0.        ,  0.        ,  0.        ],
       [-3.88337541,  0.95885108,  3.88337541, -0.95885108,  0.        ],
       [ 0.        ,  0.        ,  0.        , -1.        ,  0.        ],
       [ 0.        ,  1.        ,  0.        ,  0.        , -0.87758256],
       [ 0.        ,  0.        ,  0.        ,  0.        ,  1.        ]])
rango=5

error iter9=
2.987410829605125e-16
num iters:9
```

Paso 5. Representación y diseño del mecanismo

```
def dibuja_mecanismo(q, meca):


    # Extraer los puntos moviles del mecanismo
    X1 = q[0]
    Y1 = q[1]
    X2 = q[2]
    Y2 = q[3]
    theta = q[4]

    plt.axis('equal')


    plt.plot ([meca["XA"], X1], [meca["YA"], Y1], linewidth=10, solid_capstyle='round')
    plt.plot ([X1, X2], [Y1, Y2], linewidth=10, solid_capstyle='round')
    plt.plot ([meca["XB"],meca["XC"]], [meca["YB"],meca["YC"]], linewidth=12, solid_capstyle='butt', color='tab:red')
    plt.plot ([meca["XB"],meca["XC"]], [meca["YB"],meca["YC"]], linewidth=6, solid_capstyle='round', color='white')

    plt.plot(meca["XA"], meca["YA"], 'bo', marker='^', markersize=15)
    plt.plot(meca["XB"], meca["YB"], 'go', marker='^', markersize=15)
    plt.plot(meca["XC"], meca["YC"], 'go', marker='^', markersize=15)
    plt.plot(X1, Y1, marker='o', color='grey')
    plt.plot(X2, Y2, marker="s", markersize=12)
    plt.show()#block=False)
    return

dibuja_mecanismo(q,meca)
```

Para el diseño se han elaborado dos distintos, uno utilizando marcadores para definir los puntos fijos B y C para que resultase representativo en la docencia y otro sin los puntos fijos para que se viese mejor la corredera.

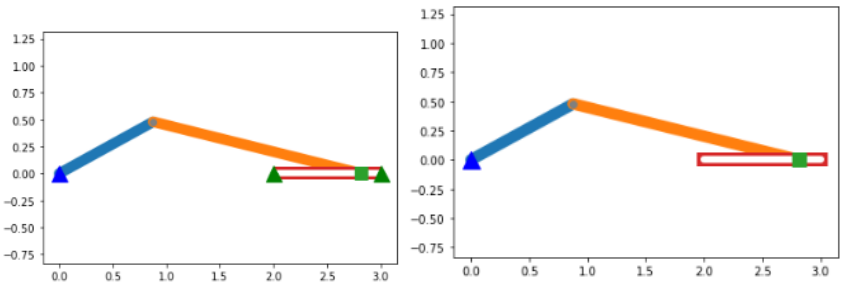


Figura 4. 26. Representación y diseño biela-manivela corredera

4.4.2.2 Problema de velocidad y aceleración

Reescribimos de nuevo el código para resolver el problema de posición para poder resolver los problemas de velocidad y aceleración.

Luego, abrimos una nueva celda para resolver el problema de velocidad.

```
def resuelve_prob_velocidad(q,qp,meca):
    qp = np.linalg.solve(jacob_Phiq(q,meca),qp)
    #print ("qp=")
    #pprint.pprint(qp)

    return qp

qp = np.zeros ((5,1))
#Velocidad del gdl. En una vuelta completa del angulo se cumple angulo=2*Pi*t
qp[4]=1
qp = resuelve_prob_velocidad (q,qp, meca)
qp
```

```
array([[ 0.47942554],
       [-0.87758256],
       [ 0.696111  ],
       [ 0.        ],
       [ 1.        ]])
```

Seguidamente resolvemos el problema de velocidad con las matrices obtenidas del cálculo teórico.

```
def resuelve_prob_aceleracion (q,qp, qpp, meca):

    #Extraer las posiciones
    X1 = q[0]
    Y1 = q[1]
    X2 = q[2]
    Y2 = q[3]
    theta = q[4]
    #Extraer las velocidades
    X1q = qp[0]
    Y1q = qp[1]
    X2q = qp[2]
    Y2q = qp[3]
    thetaq = qp[4]
    b=qpp

    b[0] = 2*(X1q)**2 + 2*(Y1q)**2
    b[1] = -2*X1q*(X2q-X1q) - 2*Y1q*(Y2q-Y1q) + 2*X2q*(X2q-X1q) + 2*Y2q*(Y2q-Y1q)
    if (abs(math.cos(theta)) < (math.sqrt(2)/2) ):
        b[3] = thetaq**2 * (meca["L1"] * math.cos(theta))
    else:
        b[3] =  thetaq**2 * (meca["L1"] * math.sin(theta))

    qpp = np.linalg.solve(-jacob_Phiq(q,meca),b)


    return qpp


qpp=np.zeros((5,1))
qpp[4] = 1 #Aceleracion conocida
qpp=resuelve_prob_aceleracion(q,qp,qpp,meca)
qpp
```

```
array([[-1.3570081 ],
       [ 0.39815702],
       [-1.87613915],
       [-0.        ],
       [-1.        ]])
```

4.4.2.3 Gráfica de velocidades y aceleraciones

El proceso seguido es el mismo que para el resto de los mecanismos.

```
def grafica_velocidad(q,meca):

    th = np.linspace(0,2*3.1416,50)

    VX1 = np.zeros((50,0))
    VY1 = np.zeros((50,0))
    VX2 = np.zeros((50,0))
    VY2 = np.zeros((50,0))
    i=0
    for t in th:

        q[4] = t

        q = resuelve_prob_posicion (q, meca)
        qp = np.zeros ((5,1))
        #Velocidad del gdl. En una vuelta completa del angulo se cumple angulo=2*Pi*t
        qp[4]=1
        qp = resuelve_prob_velocidad (q,qp, meca)


        VX1 = np.append(VX1, qp[0])
        VY1 = np.append(VY1, qp[1])
        VX2 = np.append(VX2, qp[2])
        VY2 = np.append(VY2, qp[3])
        i=i+1


    fig, axs = plt.subplots(ncols=2, figsize=(15, 15))
    plt.subplot(2,2,1)
    plt.plot(th,VX1)
    plt.xlabel ('Angulo')
    plt.ylabel ('Velocidad')
    plt.title ('Velocidad de X1')

    plt.subplot(2,2,2)
    plt.plot(th,VY1)
    plt.xlabel ('Angulo')
    plt.ylabel ('Velocidad')
    plt.title ('Velocidad de Y1')

    plt.subplot(2,2,3)
    plt.plot(th,VX2)
    plt.xlabel ('Angulo')
    plt.ylabel ('Velocidad')
    plt.title ('Velocidad de X2')

    plt.subplot(2,2,4)
    plt.plot(th,VY2)
    plt.xlabel ('Angulo')
    plt.ylabel ('Velocidad')
    plt.title ('Velocidad de Y2')

    plt.show()
    return

grafica_velocidad (q,meca)
```

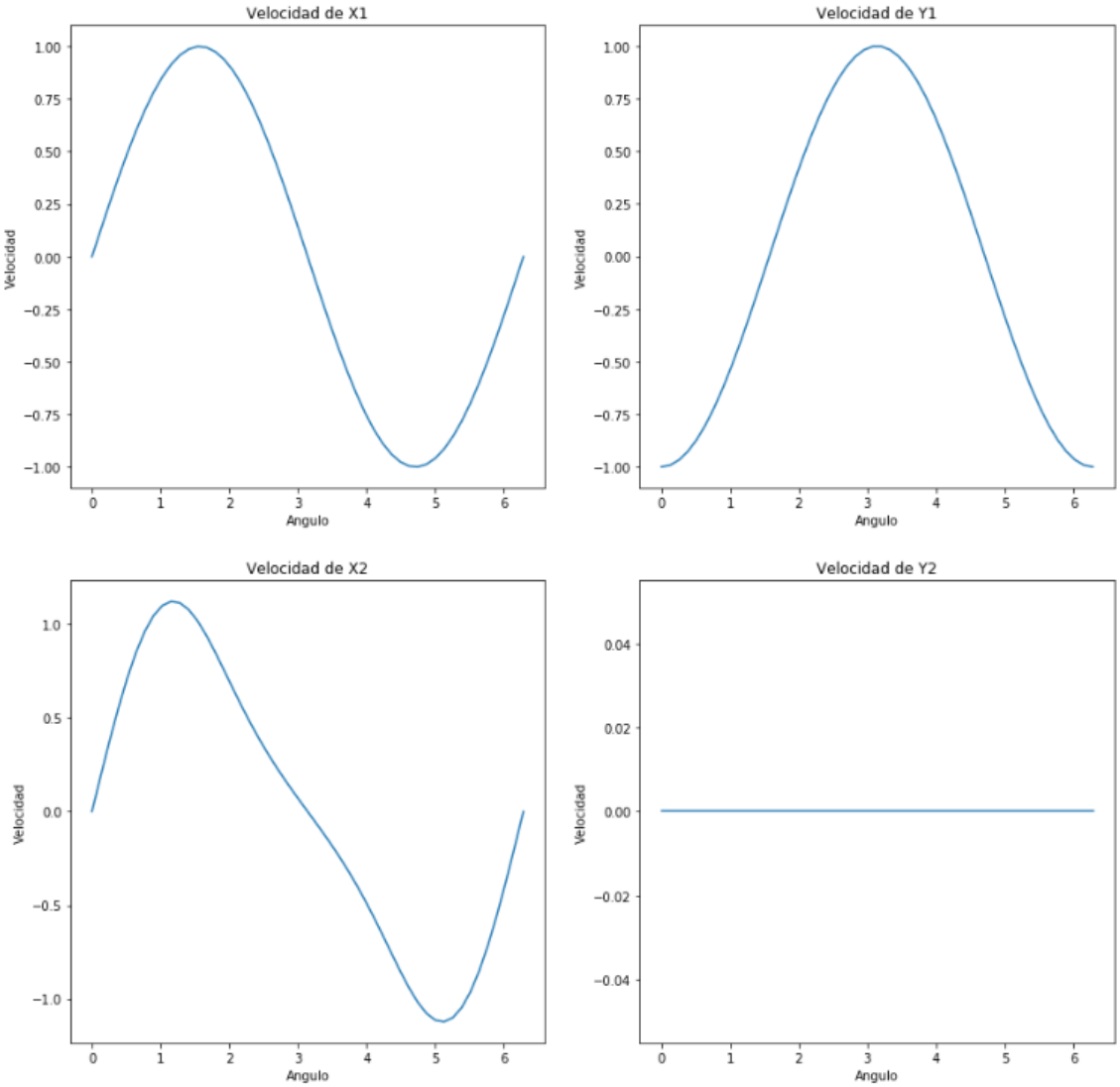


Figura 4. 27. Gráficas velocidades biela-manivela corredera

```
def grafica_aceleracion(q,meca):

    th = np.linspace(0,2*3.1416,50)

    AX1 = np.zeros((50,0))
    AY1 = np.zeros((50,0))
    AX2 = np.zeros((50,0))
    AY2 = np.zeros((50,0))


    i=0
    for t in th:

        q[4] = t
        q = resuelve_prob_posicion (q,meca)

        qp = np.zeros((5,1))
        qp[4] = 1
        qp = resuelve_prob_velocidad(q, qp, meca)

        qpp = np.zeros((5,1))
        qpp[4] = 1
        qpp = resuelve_prob_aceleracion(q,qp, qpp, meca)

        AX1 = np.append(AX1, qpp[0])
        AY1 = np.append(AY1, qpp[1])
        AX2 = np.append(AX2, qpp[2])
        AY2 = np.append(AY2, qpp[3])

        i=i+1


    fig, axs = plt.subplots(ncols=2, figsize=(15, 15))
    plt.subplot(2,2,1)
    plt.plot(th,AX1)
    plt.xlabel ('Angulo')
    plt.ylabel ('Aceleración')
    plt.title ('Aceleración de X1')

    plt.subplot(2,2,2)
    plt.plot(th,AY1)
    plt.xlabel ('Angulo')
    plt.ylabel ('Aceleración')
    plt.title ('Aceleración de Y1')

    plt.subplot(2,2,3)
    plt.plot(th,AX2)
    plt.xlabel ('Angulo')
    plt.ylabel ('Aceleración')
    plt.title ('Aceleración de X2')

    plt.subplot(2,2,4)
    plt.plot(th,AY2)
    plt.xlabel ('Angulo')
    plt.ylabel ('Aceleración')
    plt.title ('Aceleración de Y2')

    plt.show()
    return

grafica_aceleracion (q,meca)
```

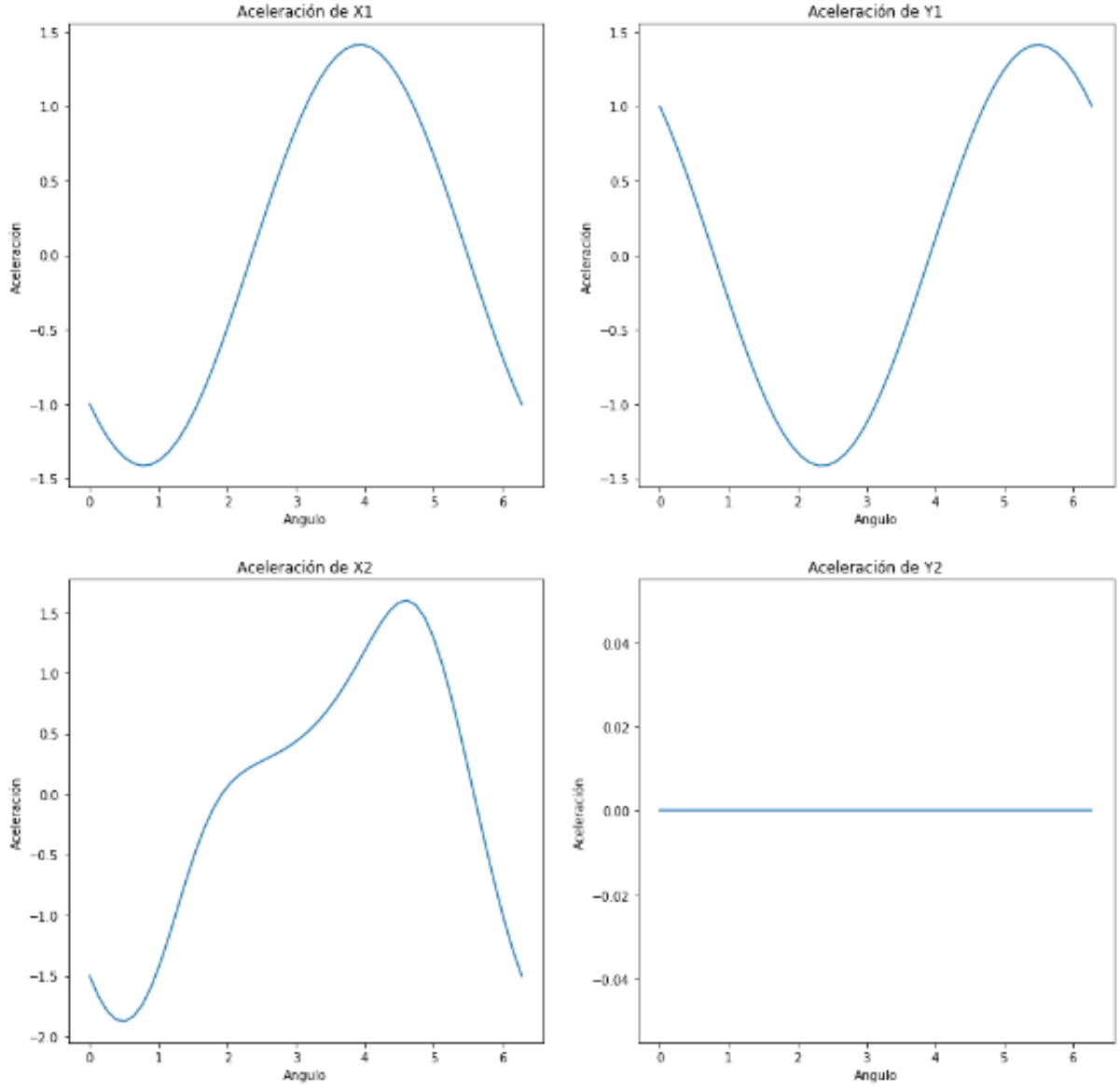


Figura 4. 28. Gráficas aceleraciones biela-manivela corredera

4.4.2.4 Animación

Implementamos la librería matplotlib.animation al igual que para los casos anteriores y generamos nuestra animación del mecanismo.

```
%matplotlib inline
import matplotlib.pyplot as plt

from matplotlib import animation, rc
from IPython.display import HTML

fig, ax = plt.subplots()

ax.set_xlim(( -5, 5))
ax.set_ylim((-5,5))
ax.set_aspect('equal')

line, = ax.plot([], [], lw=2)

last_q = q

def init():
    line.set_data([], [])
    return (line,)

def animate(i,q,meca):
    global last_q
    q = last_q

    omega=2*3.14159/100
    q[4] = i*omega


    q = resuelve_prob_posicion(q, meca)
    last_q = q

    #Extraer las coordenadas
    X1 = q[0]
    Y1 = q[1]
    X2 = q[2]
    Y2 = q[3]
    theta = q[4]

    x=[meca["XA"], X1, X2, meca["XB"], meca["XC"]]
    y=[meca["YA"], Y1, Y2, meca["YB"], meca["YC"]]


    line.set_data(x, y)
    return (line,)


anim = animation.FuncAnimation(fig, animate, init_func=init, fargs=(q,meca),
                               frames=100, interval=20,
                               blit=True)

HTML(anim.to_html5_video())
```

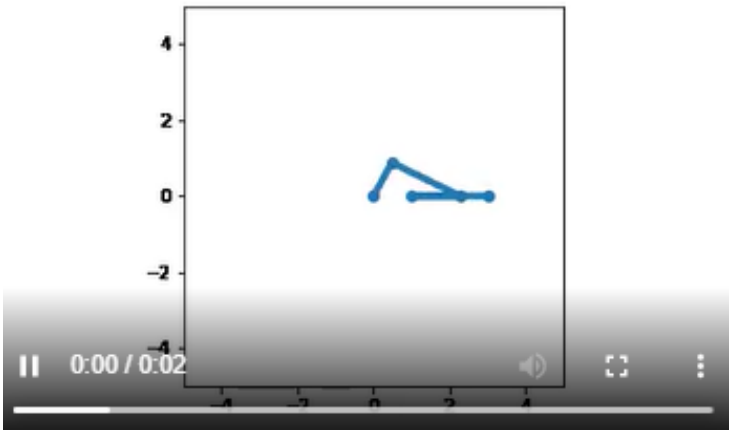


Figura 4. 29. Animación biela-manivela corredera

4.5 Cuatro Barras

,

El mecanismo de cuatro barras es la cadena cinemática cerrada más simple de eslabones unidos con un grado de libertad. Es, posiblemente, el más representativo y a nivel docente de los más importantes en la asignatura de teoría de mecanismo ya que es el primero que se emplea a la hora de explicar el análisis cinemático debido a su simplicidad.

Figura 4. 30. Ejemplo aplicación mecanismo cuatro barras

Es un mecanismo plano compuesto por 4 sólidos rígidos conectados entre sí mediante 4 pares cinemáticos de revolución.

4.5.1 Cálculo Teórico

4.5.1.1 Problema de posición

Paso 1. Modelado del mecanismo

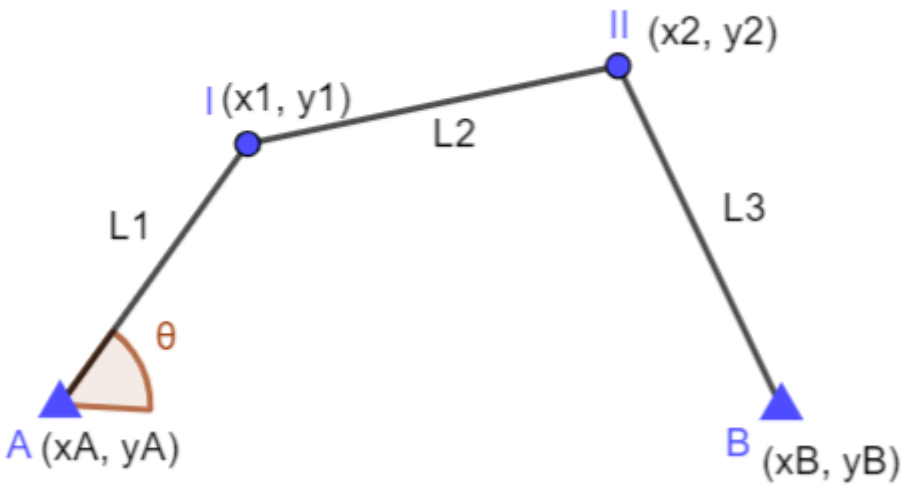


Figura 4. 31. Modelado mecanismo cuatro barras

Paso 2. Grados de libertad

$$G = 3 * (N - 1) - 2P_I - P_{II} = 3 * (4 - 1) - 2 * 4 - 0 = 1$$

Paso 3. Definición del vector q

$$q = \begin{bmatrix} x1 \\ y1 \\ x2 \\ y2 \\ \theta \end{bmatrix} \qquad (4.80)$$

Los valores iniciales para el mecanismo de 4 barras serán los siguientes:

$$L1 = 1$$

$$L2 = 3$$

$$L3 = 1$$

$$\theta = 1$$

$$X_B = 3$$

$$Y_B = 0$$

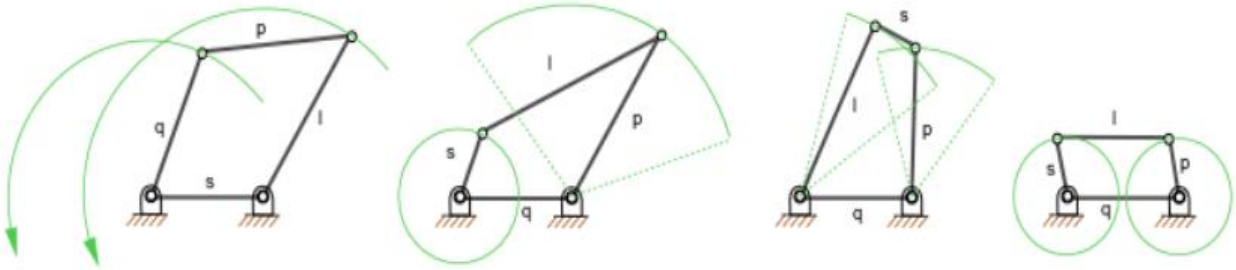
$$X_A = 0$$

$$Y_A = 0$$

Paso 4. Ley de Grashoff

El mecanismo cuatro barras es del tipo cuadrilátero articulado, razón por la cual habrá que evaluar si se cumple la Ley de Grashoff.

La Ley de Grashof establece que un mecanismo de cuatro barras tiene al menos una articulación de revolución completa, si y solo si la suma de las longitudes de la barra más corta y la barra más larga es menor o igual que la suma de las longitudes de las barras restantes.



$$S + L \leq P + Q \qquad (4.81)$$

Paso 5. Matriz de restricciones

Para definir la matriz de restricciones aplicamos las siguientes ecuaciones de restricción:

- Sólido A-1:

$$(X_1 - X_A)^2 + (Y_1 - Y_A)^2 - {L_1}^2 = 0$$

- Sólido 1-2:

$$(X_2 - X_1)^2 + (Y_2 - Y_1)^2 - {L_2}^2 = 0$$

- Sólido 2-B:

$$(X_B - X_2)^2 + (Y_B - Y_2)^2 - {L_2}^2 = 0$$

- Coordenada relativa θ:

$$(X_1 - X_A) - L_1 * cos\theta = 0$$
$$(Y_1 - Y_A) - L_2 * sin\theta = 0$$

A partir de estas ecuaciones creamos la matriz de restricciones:

-Si $cos\theta < \frac{1}{\sqrt{2}}$:

$$\Phi = \begin{bmatrix} (X_1 - X_A)^2 + (Y_1 - Y_A)^2 - {L_1}^2 \\ (X_2 - X_1)^2 + (Y_2 - Y_1)^2 - {L_2}^2 \\ (X_B - X_2)^2 + (Y_B - Y_2)^2 - {L_2}^2 \\ (X_1 - X_A) - L_1 * cos\theta \end{bmatrix} \quad (4.82)$$

-Si $cos\theta > \frac{1}{\sqrt{2}}$:

$$\Phi = \begin{bmatrix} (X_1 - X_A)^2 + (Y_1 - Y_A)^2 - {L_1}^2 \\ (X_2 - X_1)^2 + (Y_2 - Y_1)^2 - {L_2}^2 \\ (X_B - X_2)^2 + (Y_B - Y_2)^2 - {L_2}^2 \\ (Y_1 - Y_A) - L_2 * sin\theta \end{bmatrix} \quad (4.83)$$

Paso 6. Matriz Jacobiana

Se procede a calcular la matriz Jacobiana derivando la matriz de restricciones Φ respecto del vector q.

-Si $cos\theta < \frac{1}{\sqrt{2}}$:

$$\Phi_q = \begin{bmatrix} 2 * X_1 & 2 * Y_1 & 0 & 0 & 0 \\ -2 * (X_2 - X_1) & -2 * (Y_2 - Y_1) & 2 * (X_2 - X_1) & 2 * (Y_2 - Y_1) & 0 \\ 0 & 0 & -2 * (X_B - X_2) & -2 * (Y_B - Y_2) & 0 \\ 1 & 0 & 0 & 0 & L_1 * sin\theta \end{bmatrix} \quad (4.84)$$

-Si $cos\theta > \frac{1}{\sqrt{2}}$:

$$\Phi_q = \begin{bmatrix} 2 * X_1 & 2 * Y_1 & 0 & 0 & 0 \\ -2 * (X_2 - X_1) & -2 * (Y_2 - Y_1) & 2 * (X_2 - X_1) & 2 * (Y_2 - Y_1) & 0 \\ 0 & 0 & -2 * (X_B - X_2) & -2 * (Y_B - Y_2) & 0 \\ 1 & 0 & 0 & 0 & -L_1 * cos\theta \end{bmatrix} \quad (4.85)$$

Paso 7. Resolución del problema de posición

$$\Phi_q * \Delta q = -\Phi$$

-Si $cos\theta < \frac{1}{\sqrt{2}}$:

$$\begin{bmatrix} 2*X_1 & 2*Y_1 & 0 & 0 & 0 \\ -2*(X_2-X_1) & -2*(Y_2-Y_1) & 2*(X_2-X_1) & 2*(Y_2-Y_1) & 0 \\ 0 & 0 & -2*(X_B-X_2) & -2*(Y_B-Y_2) & 0 \\ 1 & 0 & 0 & 0 & L_1*sin\theta \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x1_1-x1_0 \\ y1_1-y1_0 \\ x2_1-x2_0 \\ y2_1-y2_0 \\ \theta_1-\theta_0 \end{bmatrix} =$$

$$= -\begin{bmatrix} (X_1-X_A)^2+(Y_1-Y_A)^2-{L_1}^2 \\ (X_2-X_1)^2+(Y_2-Y_1)^2-{L_2}^2 \\ (X_B-X_2)^2+(Y_B-Y_2)^2-{L_2}^2 \\ (X_1-X_A)-L_1*cos\theta \\ 0 \end{bmatrix} \qquad (4.86)$$

-Si $cos\theta > \frac{1}{\sqrt{2}}$:

$$\begin{bmatrix} 2*X_1 & 2*Y_1 & 0 & 0 & 0 \\ -2*(X_2-X_1) & -2*(Y_2-Y_1) & 2*(X_2-X_1) & 2*(Y_2-Y_1) & 0 \\ 0 & 0 & -2*(X_B-X_2) & -2*(Y_B-Y_2) & 0 \\ 0 & 1 & 0 & 0 & -L_1*cos\theta \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x1_1-x1_0 \\ y1_1-y1_0 \\ x2_1-x2_0 \\ y2_1-y2_0 \\ \theta_1-\theta_0 \end{bmatrix} =$$

$$= -\begin{bmatrix} (X_1-X_A)^2+(Y_1-Y_A)^2-{L_1}^2 \\ (X_2-X_1)^2+(Y_2-Y_1)^2-{L_2}^2 \\ (X_B-X_2)^2+(Y_B-Y_2)^2-{L_2}^2 \\ (Y_1-Y_A)-L_2*sin\theta \\ 0 \end{bmatrix} \qquad (4.87)$$

4.5.1.2 Problema de velocidad

Resolvemos el problema de velocidad con la velocidad angular como dato conocido $\dot{\theta} = 1$.

$$\Phi_q * \dot{q} = -\Phi_t$$

-Si $cos\theta < \frac{1}{\sqrt{2}}$:

$$\begin{bmatrix} 2*X_1 & 2*Y_1 & 0 & 0 & 0 \\ -2*(X_2-X_1) & -2*(Y_2-Y_1) & 2*(X_2-X_1) & 2*(Y_2-Y_1) & 0 \\ 0 & 0 & -2*(X_B-X_2) & -2*(Y_B-Y_2) & 0 \\ 1 & 0 & 0 & 0 & L_1*sin\theta \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} \dot{x1} \\ \dot{y1} \\ \dot{x2} \\ \dot{y2} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \qquad (4.88)$$

-Si $cos\theta > \frac{1}{\sqrt{2}}$:

$$\begin{bmatrix} 2*X_1 & 2*Y_1 & 0 & 0 & 0 \\ -2*(X_2-X_1) & -2*(Y_2-Y_1) & 2*(X_2-X_1) & 2*(Y_2-Y_1) & 0 \\ 0 & 0 & -2*(X_B-X_2) & -2*(Y_B-Y_2) & 0 \\ 0 & 1 & 0 & 0 & -L_1*cos\theta \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} \dot{x1} \\ \dot{y1} \\ \dot{x2} \\ \dot{y2} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \quad (4.89)$$

4.5.1.3 Problema de aceleración

$$\Phi_q * \ddot{q} = -\dot{\Phi}_q * \dot{q} - \dot{\Phi}_t$$

La matriz $\dot{\Phi}_t$ va a tener un valor nulo puesto que Φ_t solo tiene un valor y es una constante por tanto su derivada es 0. La única matriz que queda por calcular es la derivada del Jacobiano $\dot{\Phi}_q$.

-Si $cos\theta < \frac{1}{\sqrt{2}}$:

$$\dot{\Phi}_q = \begin{bmatrix} 2*\dot{X}_1 & 2*\dot{Y}_1 & 0 & 0 & 0 \\ 2*\dot{X}_1 & 2*\dot{Y}_1 & 2*\dot{X}_2 & 2*\dot{Y}_2 & 0 \\ 0 & 0 & 2*\dot{X}_2 & 2*\dot{Y}_2 & 0 \\ 0 & 0 & 0 & 0 & \dot{\theta}*L_1*cos\theta \end{bmatrix} \quad (4.90)$$

-Si $cos\theta > \frac{1}{\sqrt{2}}$:

$$\dot{\Phi}_q = \begin{bmatrix} 2*\dot{X}_1 & 2*\dot{Y}_1 & 0 & 0 & 0 \\ 2*\dot{X}_1 & 2*\dot{Y}_1 & 2*\dot{X}_2 & 2*\dot{Y}_2 & 0 \\ 0 & 0 & 2*\dot{X}_2 & 2*\dot{Y}_2 & 0 \\ 0 & 0 & 0 & 0 & \dot{\theta}*L_2*sen\theta \end{bmatrix} \quad (4.91)$$

Ahora procedemos a calcular la multiplicación de $\dot{\Phi}_q * \dot{q}$, añandiendole el dato de la aceleración angular $\ddot{\theta} = 1$.

-Si $cos\theta < \frac{1}{\sqrt{2}}$:

$$\dot{\Phi}_q * \dot{q} = \begin{bmatrix} 2*\dot{X_1}^2 + 2*\dot{Y_1}^2 \\ 2*\dot{X_1}^2 + 2*\dot{Y_1}^2 + 2*\dot{X_2}^2 + 2*\dot{Y_2}^2 \\ 2*\dot{X_2}^2 + 2*\dot{Y_2}^2 \\ \dot{\theta}^2*L_1*cos\theta \\ 1 \end{bmatrix} \quad (4.79)$$

-Si $cos\theta > \frac{1}{\sqrt{2}}$:

$$\dot{\Phi}_q * \dot{q} = \begin{bmatrix} 2*\dot{X_1}^2 + 2*\dot{Y_1}^2 \\ 2*\dot{X_1}^2 + 2*\dot{Y_1}^2 + 2*\dot{X_2}^2 + 2*\dot{Y_2}^2 \\ 2*\dot{X_2}^2 + 2*\dot{Y_2}^2 \\ \dot{\theta}^2*L_1*sin\theta \\ 1 \end{bmatrix} \quad (4.79)$$

4.5.2 Implementación en Python

4.5.2.1 Problema de posición

Paso 1. Posición inicial y ley de Grashoff

```
print ('MECANISMO DE CUATRO BARRAS')
print ('============================')

meca["L1"] = float (input ('Introduce longitud L1:'))
meca["L2"] = float (input ('Introduce longitud L2:'))
meca["L3"] = float (input ('Introduce longitud L3:'))
meca["theta"] = float (input ('Introduce angulo inicial theta:'))
meca["XB"] = float (input ('Introduce coordenada en x del punto B:'))
meca["XA"] = 0
meca["YA"] = 0
meca["YB"] = 0

# Defino posicion inicial:
q = np.array ([[0.1], [meca["L1"]], [1], [0.2], [meca["theta"]]])
print('q: ' + str(q))
```

```
MECANISMO DE CUATRO BARRAS
============================
Introduce longitud L1:1
Introduce longitud L2:3
Introduce longitud L3:1
Introduce angulo inicial theta:1.5
Introduce coordenada en x del punto B:3
q: [[0.1]
 [1. ]
 [1. ]
 [0.2]
 [1.5]]
```

```
# LEY DE GRASHOFF

a = meca["XB"] - meca ["XA"]
b = meca["L1"]
c = meca["L2"]
d = meca["L3"]

ListaDeLongitudes = [d, c, b, a]

def ordenar(lista):

    for x in range (1, len(lista)):
        for y in range(len(lista)-1):
            if lista[y] < lista[y+1]:
                aux = lista[y]
                lista[y] = lista[y+1]
                lista[y+1] = aux

ordenar(ListaDeLongitudes)


print(ListaDeLongitudes)

a = (ListaDeLongitudes[3])
b = (ListaDeLongitudes[2])
c = (ListaDeLongitudes[1])
d = (ListaDeLongitudes[0])

if ((b+c)<(a+d)):
    print ("No cumple la desigualdad de Grashoff, por lo que la simulación no se ejecutará correctamente.")
```

Paso 2. Matriz de restricciones

```
def Phi (q,meca):

    Phi = np.zeros((5,1))

    #Extraer coordenadas
    X1 = q[0]
    Y1 = q[1]
    X2 = q[2]
    Y2 = q[3]
    theta = q[4]

    Phi[0] = X1**2 + Y1**2 - meca["L1"]**2
    Phi[1] = (X2-X1)**2 + (Y2-Y1)**2 -meca["L2"]**2
    Phi[2] = (meca["XB"]-X2)**2 + Y2**2 - meca["L3"]**2

    if (abs(math.cos(theta)) < (math.sqrt(2)/2) ):
        Phi[3] = X1-meca["L1"]*math.cos(theta)
    else:
        Phi[3] = Y1-meca["L1"]*math.sin(theta)

    return Phi
```

Paso 3. Matriz Jacobiana

```
def jacob_Phiq(q,meca):

    Jacob = np.zeros((5,5))

    #Extraer coordenadas

    X1 = q[0]
    Y1 = q[1]
    X2 = q[2]
    Y2 = q[3]
    theta = q[4]

    #Montar matriz

    Jacob[0,0] = 2*X1
    Jacob[0,1] = 2*Y1
    Jacob[1,0] = -2*(X2-X1)
    Jacob[1,1] = -2*(Y2-Y1)
    Jacob[1,2] = 2*(X2-X1)
    Jacob[1,3] = 2*(Y2-Y1)
    Jacob[2,2] = -2*(meca["XB"]-X2)
    Jacob[2,3] = -2*(0-Y2)

    if (abs(math.cos(theta)) < (math.sqrt(2)/2) ):
        Jacob[3,4] = meca["L1"]*math.sin(theta)
        Jacob[3,0] = 1
    else:
        Jacob[3,4] = -meca["L1"]*math.cos(theta)
        Jacob[3,1] = 1


    Jacob[4,4] = 1

    return Jacob
```

Paso 4. Resolver problema de posición

```
def resuelve_prob_posicion(q_init, meca):
    #Inicializacion de variables
    error = 1e10
    tolerancia = 1e-10

    deltaQ = np.zeros ((5,1))
    q = q_init
    i=0


    while (error > tolerancia and i<=100):
        print("q=")
        pprint.pprint(q)


        #Extraer las coordenadas
        X1 = q[0]
        Y1 = q[1]
        X2 = q[2]
        Y2 = q[3]
        theta = q[4]

        fi=Phi(q,meca)
        print ("Phi" + "=")
        pprint.pprint(fi)
        J = jacob_Phiq(q,meca)
        print ("jacob" + "=")
        pprint.pprint(J)
        rango = np.linalg.matrix_rank(J, 1e-5)
        print("rango=" + str(rango) + "\n")

        deltaQ = np.linalg.solve(J,-fi)
        q = q + deltaQ
        error = np.linalg.norm(deltaQ)
        i=i+1

        print("error iter" + str(i) + "=")
        pprint.pprint(error)
    print("num iters:" + str(i))
    if (error > tolerancia):
        raise Exception ('No se puede alcanzar la posición')
    return q

q=resuelve_prob_posicion(q,meca)
```

```
q=
array([[ 0.0707372 ],
       [ 0.99749499],
       [ 2.44727841],
       [-0.83336598],
       [ 1.5       ]])
Phi=
array([[0.00000000e+00],
       [1.82964754e-12],
       [1.82831528e-12],
       [0.00000000e+00],
       [0.00000000e+00]])
jacob=
array([[ 0.1414744 ,  1.99498997,  0.        ,  0.        ,  0.        ],
       [-4.75308243,  3.66172192,  4.75308243, -3.66172192,  0.        ],
       [ 0.        ,  0.        , -1.10544317, -1.66673195,  0.        ],
       [ 1.        ,  0.        ,  0.        ,  0.        ,  0.99749499],
       [ 0.        ,  0.        ,  0.        ,  0.        ,  1.        ]])
rango=5

error iter8=
9.453608957454925e-13
num iters:8
```

Paso 5. Representación y diseño del mecanismo

```
def dibuja_mecanismo(q, meca):

    q = resuelve_prob_posicion(q,meca)

    # Extraer los puntos moviles del mecanismo
    X1 = q[0]
    Y1 = q[1]
    X2 = q[2]
    Y2 = q[3]


    plt.axis('equal')

    filled_marker_style = dict(marker='o', linestyle='solid', markersize=15,
                           color='darkgrey',
                           markerfacecolor='tab:blue',
                           markerfacecoloralt='lightsteelblue',
                           markeredgecolor='brown')


    plt.plot ([meca["XA"], X1], [meca["YA"], Y1], linewidth= 10, solid_capstyle='round', color='salmon')
    plt.plot ([X1, X2], [Y1, Y2], linewidth= 10, solid_capstyle='round', color='gold')
    plt.plot ([X2, meca["XB"]], [Y2, meca ["YB"]], linewidth= 10, solid_capstyle='round', color='silver')

    plt.plot(meca["XA"], meca["YA"], 'bo', marker='^', markersize=15)
    plt.plot(meca["XB"], meca["YB"], 'go', marker='^', markersize=15)
    plt.plot(X1, Y1, marker="o", markersize=10)
    plt.plot(X2, Y2,  marker='o', color='m', markersize=10)

    plt.show()#block=False)
    return

dibuja_mecanismo(q,meca)
```

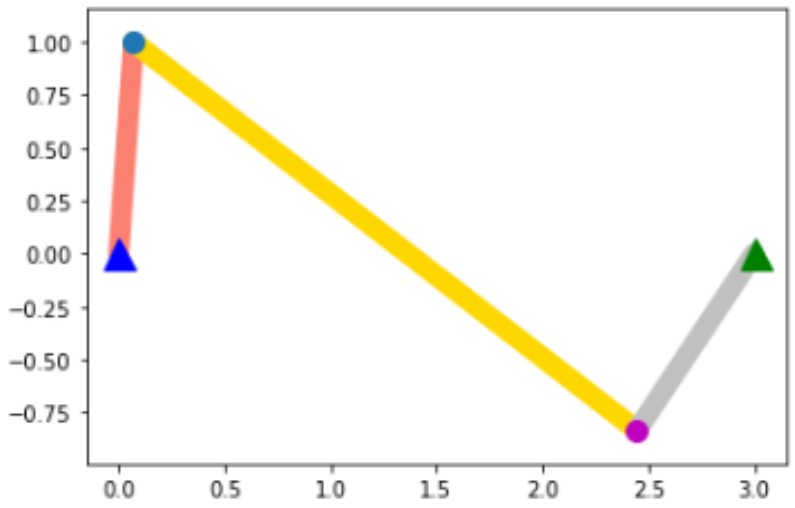


Figura 4. 32. Representación gráfica mecanismo cuatro barras

4.5.2.2 Problema de velocidad

```
#VELOCIDADES 4 BARRAS

def resuelve_prob_velocidad(q,qp,meca):
    qp = np.linalg.solve(jacob_Phiq(q,meca),qp)
    #print ("qp=")
    #pprint.pprint(qp)

    return qp

qp = np.zeros ((5,1))
#Velocidad del gdl. En una vuelta completa del angulo se cumple angulo=2*Pi*t
qp[4]=1

qp = resuelve_prob_velocidad (q,qp, meca)
qp
```

```
array([[-0.99749499],
       [ 0.0707372 ],
       [-0.69624295],
       [ 0.46177612],
       [ 1.        ]])
```

4.5.2.3 Problema de aceleración

```
#ACELERACIONES 4 BARRAS

def resuelve_prob_aceleracion (q,qp,qpp,meca):
    #Extraer las posiciones
    X1 = q[0]
    Y1 = q[1]
    X2 = q[2]
    Y2 = q[3]
    theta = q[4]
    #Extraer las velocidades
    X1q = qp[0]
    Y1q = qp[1]
    X2q = qp[2]
    Y2q = qp[3]
    thetaq = qp[4]
    b=qpp

    b[0] = 2*(X1q)**2 + 2*(Y1q)**2
    b[1] = -2*X1q*(X2q-X1q) - 2*Y1q*(Y2q- Y1q) + 2*X2q*(X2q-X1q) + 2*Y2q*(Y2q-Y1q)
    b[2] = 2*X2q**2 + 2*Y2q**2

    if (abs(math.cos(theta)) < (math.sqrt(2)/2) ):
        b[3] = thetaq**2 * (meca["L1"] * math.cos(theta))
    else:
        b[3] =  thetaq**2 * (meca["L1"] * math.sin(theta))

    qpp = np.linalg.solve(-jacob_Phiq(q,meca),b)
    print ("qpp=")
    pprint.pprint(qpp)

    return qpp

qpp=np.zeros((5,1))
qpp[4] = 1 #Aceleracion conocida

qpp=resuelve_prob_aceleracion(q,qp,qpp,meca)
qpp
```

```
array([[ 0.92675778],
       [-1.06823219],
       [ 1.51720676],
       [-0.16871519],
       [-1.        ]])
```

4.5.2.4 Gráfica de velocidades y aceleraciones

```
def grafica_velocidad(q,meca):

    th = np.linspace(0,2*3.1416,50)
    #print ("th=")
    #pprint.pprint(th)
    VX1 = np.zeros((50,0))
    VY1 = np.zeros((50,0))
    VX2 = np.zeros((50,0))
    VY2 = np.zeros((50,0))
    i=0
    for t in th:

        q[4] = t

        q = resuelve_prob_posicion (q, meca)
        qp = np.zeros ((5,1))
        qp[4]=1
        qp = resuelve_prob_velocidad (q,qp, meca)


        VX1 = np.append(VX1, qp[0])
        VY1 = np.append(VY1, qp[1])
        VX2 = np.append(VX2, qp[2])
        VY2 = np.append(VY2, qp[3])
        i=i+1

    fig, axs = plt.subplots(ncols=2, figsize=(15, 15))
    plt.subplot(2,2,1)
    plt.plot(th,VX1)
    plt.xlabel ('Angulo')
    plt.ylabel ('Velocidad')
    plt.title ('Velocidad de X1')

    plt.subplot(2,2,2)
    plt.plot(th,VY1)
    plt.xlabel ('Angulo')
    plt.ylabel ('Velocidad')
    plt.title ('Velocidad de Y1')

    plt.subplot(2,2,3)
    plt.plot(th,VX2)
    plt.xlabel ('Angulo')
    plt.ylabel ('Velocidad')
    plt.title ('Velocidad de X2')

    plt.subplot(2,2,4)
    plt.plot(th,VY2)
    plt.xlabel ('Angulo')
    plt.ylabel ('Velocidad')
    plt.title ('Velocidad de Y2')

    plt.show()
    return

grafica_velocidad (q,meca)
```

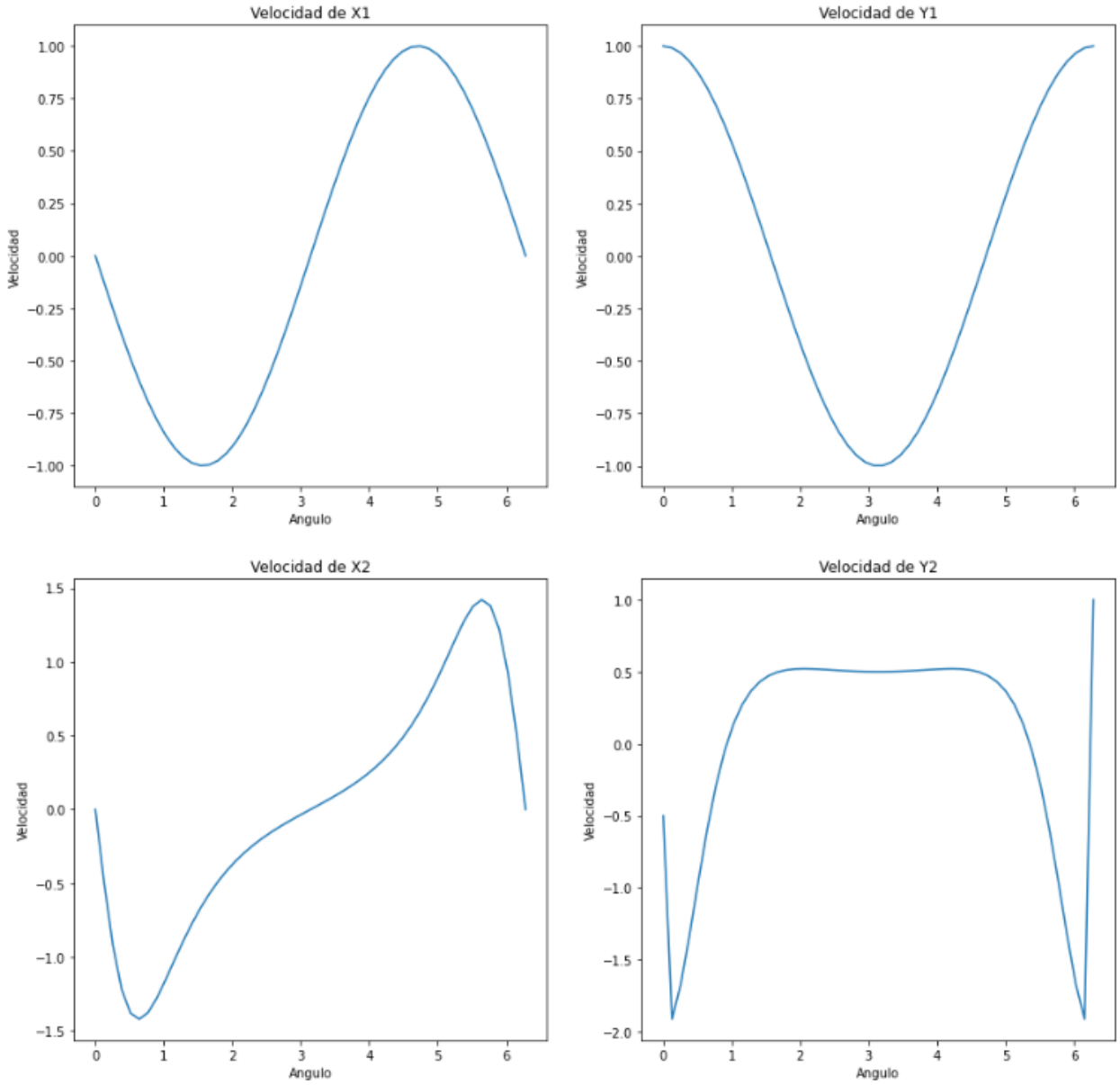


Figura 4. 33. Gráficas velocidades mecanismo cuatro barras

```
def grafica_aceleracion(q,meca):

    th = np.linspace(0,2*3.1416,50)

    AX1 = np.zeros((50,0))
    AY1 = np.zeros((50,0))
    AX2 = np.zeros((50,0))
    AY2 = np.zeros((50,0))


    i=0
    for t in th:

        q[4] = t
        q = resuelve_prob_posicion (q,meca)

        qp = np.zeros((5,1))
        qp[4] = 1
        qp = resuelve_prob_velocidad(q, qp, meca)

        qpp = np.zeros((5,1))
        qpp[4] = 1
        qpp = resuelve_prob_aceleracion(q,qp, qpp, meca)

        AX1 = np.append(AX1, qpp[0])
        AY1 = np.append(AY1, qpp[1])
        AX2 = np.append(AX2, qpp[2])
        AY2 = np.append(AY2, qpp[3])

        i=i+1


    fig, axs = plt.subplots(ncols=2, figsize=(15, 15))
    plt.subplot(2,2,1)
    plt.plot(th,AX1)
    plt.xlabel ('Angulo')
    plt.ylabel ('Aceleración')
    plt.title ('Aceleración de X1')

    plt.subplot(2,2,2)
    plt.plot(th,AY1)
    plt.xlabel ('Angulo')
    plt.ylabel ('Aceleración')
    plt.title ('Aceleración de Y1')

    plt.subplot(2,2,3)
    plt.plot(th,AX2)
    plt.xlabel ('Angulo')
    plt.ylabel ('Aceleración')
    plt.title ('Aceleración de X2')

    plt.subplot(2,2,4)
    plt.plot(th,AY2)
    plt.xlabel ('Angulo')
    plt.ylabel ('Aceleración')
    plt.title ('Aceleración de Y2')

    plt.show()
    return

grafica_aceleracion (q,meca)
```

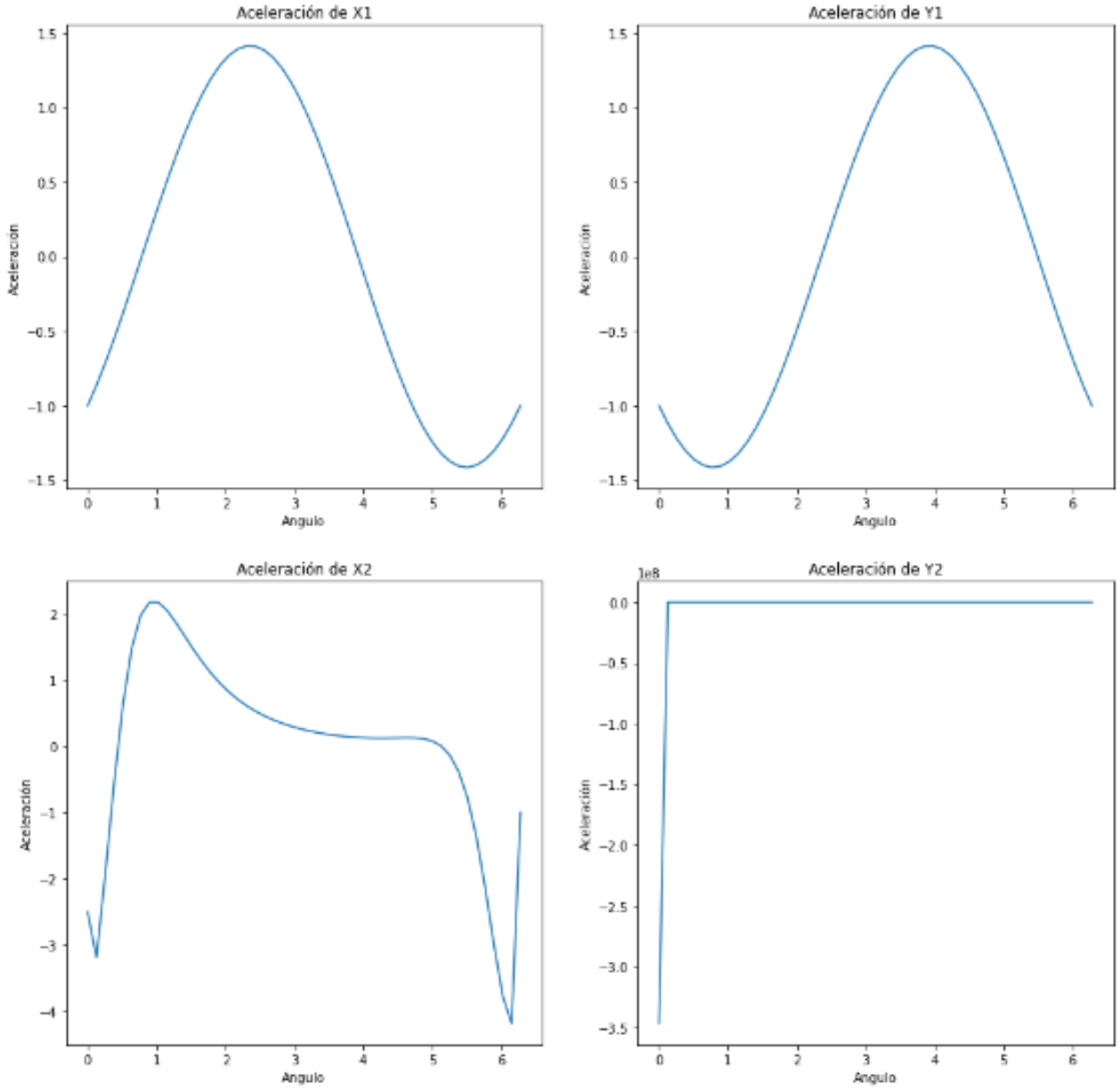


Figura 4. 34. Gráficas aceleraciones cuatro barras

4.5.3 Animación

```
%matplotlib inline
import matplotlib.pyplot as plt

from matplotlib import animation, rc
from IPython.display import HTML

fig, ax = plt.subplots()

ax.set_xlim(( -5, 5))
ax.set_ylim((-5,5))
ax.set_aspect('equal')

line, = ax.plot([], [], lw=2)

last_q = q

def init():
    line.set_data([], [])
    return (line,)

def animate(i,q,meca):
    global last_q
    q = last_q
    # i: contador de iteracion: hay que mapearla a un ángulo de la manivela
    omega=2*3.14159/100  # vel. angular
    q[4] = i*omega

    #llamar problema de pos:
    q = resuelve_prob_posicion(q, meca)
    last_q = q

    #Extraer las coordenadas
    X1 = q[0]
    Y1 = q[1]
    X2 = q[2]
    Y2 = q[3]
    theta = q[4]

    x=[meca["XA"], X1, X2, meca["XB"]]
    y=[meca["YA"], Y1, Y2, meca["YB"]]


    line.set_data(x, y)
    return (line,)


anim = animation.FuncAnimation(fig, animate, init_func=init, fargs=(q,meca),
                               frames=100, interval=20,
                               blit=True)

HTML(anim.to_html5_video())
```

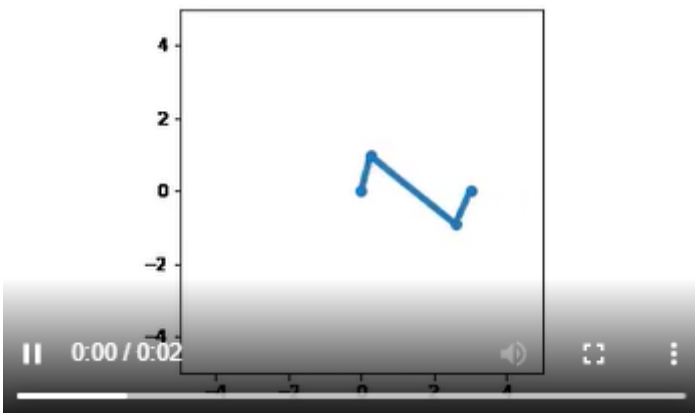


Figura 4. 35. Animación mecanismo cuatro barras

5 CONCLUSIONES Y TRABAJOS FUTUROS

Con la realización de este Trabajo Fin de Grado se ha conseguido implementar una nueva herramienta interactiva que no solo ayuda al alumnado a entender mejor la asignatura de teoría de mecanismo si no que, además, crea un nuevo reto al alumno que curse la asignatura de manejarse en un nuevo entorno informático.

Con la creación de estos mecanismos se ha observado que esta herramienta es muy útil para a el aprendizaje de cualquier materia, puesto que la creación de dichos mecanismos no necesita de un gran conocimiento a nivel de programación.

Los mecanismos creados a partir de Jupyter Notebook serán un gran refuerzo para el estudio autónomo de los alumnos que cursen la asignatura. Dispondrán de una herramienta con la que podrán comprobar los resultados obtenidos en sus cálculos además de múltiples posibilidades de ejercicios variando únicamente los datos que se introduzcan. Esto permitirá que no necesiten de páginas y páginas de ejercicios resueltos y que con solo una aplicación sea posible realizar un amplio trabajo de cara a la preparación de la asignatura. Con esta facilidad el interés del alumno por la asignatura será mayor gracias a la visualización del mecanismo que ellos mismos puedan crear según sus intereses.

No cabe duda de que además de ayudar a la compresión de la asignatura y el aumento de interés por esta, también mejorará la compresión del alumno en el campo de la programación informática, esto llevará al alumno a desarrollar una serie de conceptos y destrezas muy útiles de cara a su formación profesional puesto que uso de programas informáticos en el campo de la ingeniería es cada vez mayor y el conocimiento de este tipo de herramientas es fundamental si pensamos en el objetivo de la inserción laboral.

Mejorar y facilitar la labor docente han sido los objetivos propuestos para la realización de este trabajo fin de grado y se espera que pueda cumplirse en un futuro cercano implementando este tipo de herramientas.

5.1 Trabajos futuros

El desarrollo de estas herramientas para la labor docente es tan amplio que pueden (y deben) realizarse diversos trabajos en base a este proyecto. La mejora tanto mecánica como de diseño es palpable pues para algunos de ellos solo permiten variar pequeños detalles del mecanismo pues si se le dejase más libertad a la hora de elegir los componentes podría llevar a diversos errores.

Además de la mejora de los mecanismos realizados en este trabajo también existe la posibilidad de desarrollar la infinidad de mecanismos que existen en los libros y que se utilizan para diversos objetivos. El método realizado para este trabajo viene de un trabajo anterior y, por tanto, también puede servir para trabajos posteriores.

6 BIBLIOGRAFÍA

[1] Bahit, Eugenia. *Python para principiantes.* Buenos Aires: Creative Commons, 2012.

[2] Claraco, Jose Luis Blanco, Jose Luis Torres Moreno, y Antonio Giménez Fernández. *Teoría de Mecanismos: Apuntes y problemas resueltos.* Almería, 2015.

[3] Fernandez, Antonio Giménez, Jose Luis Torres Moreno, y Jose Luis Blanco Claraco. *Análisis cinemático por métodos numéricos.* Almería, 2017.

[4] Geogebra. s.f. https://www.geogebra.org/.

[5] González, Jose María Arias. *Cálculo y diseño de mecanismo de barras configurables para prácticas.* Sevilla, 2013.

[6] Granado, Eduardo Cabrera, y Elena Díaz García. *Manual de uso de Jupyter Notebook para aplicaciones docentes.* Madrid: Universidad Complutense de Madrid, s.f.

[7] Interactive Chaos. 21 de Enero de 2019. https://interactivechaos.com/.

[8] Iturriagagoitia, Alejo Avello. *Teoría de Máquinas 2nd ed.* Navarra, 2014.

[9] Lozano, Beatriz Díaz. *Desarrollo de material de prácticas interactivo para Teoría de Mecanismos usando Jupyter Notebook.* Almería, 2019.

[11] Marín, Francisco Sánchez. *Mecapedia.* 19 de Diciembre de 2020. http://www.mecapedia.uji.es/ (último acceso: 10 de Junio de 2022).

[12] Matplotlib. 2012. https://matplotlib.org/.

[13] Rincón, Alfonso Fernández del. *Métodos Numéricos de Análisis Cinemático.* Cantabria: Creative Commons, s.f.

[14] Rondón, Izary. «¿Qué es Anaconda?» *Blog Python*, 2022.

[15] Santos, Paloma Recuero de los. «Python para todos (2): ¿Qué son los Jupyter Notebooks?» *Think Big*, 2018.

[16] Toro, Luigys. «Anaconda Distribution: La Suite más completa para la Ciencia de datos con Python.» *Desde Linux*, 2018.

[17] Universidades, Santander. «Python: qué es y por qué deberías aprender a utilizarlo.» *Becas Santander*, 2021.

UNIVERSIDAD DE ALMERÍA

Resumen/Abstract

El proyecto llevado a cabo consiste en el desarrollo de un material de apoyo interactivo para mejorar la forma de explicar y entender la asignatura de Teoría de Mecanismos. Se basa en la creación de un código en el programa Jupyter Notebook con el cual será posible mostrar el comportamiento de los distintos mecanismos de una manera más interesante para el alumno ya que en lugar de una imagen podrán ser ellos mismos los que creen el mecanismo.

GRADO EN INGENIERÍA MECÁNICA, 2020/2021