UNIVERSIDAD DE ALMERÍA

ESCUELA SUPERIOR DE INGENIERÍA

TRABAJO FIN DE GRADO

INGENIERÍA MECÁNICA

Desarrollo de material de prácticas interactivo para Teoría de Mecanismos usando Jupyter Notebook

Autor:

Beatriz Díaz Lozano

Tutores:

José Luis Blanco Claraco

José Luis Torres Moreno

A mi familia,

a mi pareja Dani

Agradecimientos

En estas líneas quiero mostrar mi agradecimiento a todas las personas que en estos duros años me han dado su apoyo para aprender y superarme en esta difícil carrera en la Escuela Superior de Ingeniería de la Universidad de Almería.

Para comenzar, agradecer el apoyo aportado por toda mi familia, ya que siempre han estado ahí para apoyarme en los momentos difíciles. También quiero agradecer a mi pareja, Dani, por demostrarme que podía con todo incluso cuando sentía que había llegado al límite.

Alguien de quien no puedo olvidarme es de mi compañero Agui, que ha estado conmigo prácticamente todos los días durante todos estos años, haciendo que las tardes en la universidad haciendo trabajos eternos y estudiando fuesen tan llevaderos que podría incluso decirse divertidos.

Por último, quiero agradecer especialmente a mi tutor José Luis por su ayuda, comprensión y paciencia, ya que sin él este trabajo definitivamente no habría sido posible.

Muchas gracias a todos.

Índice

Resumen

En este proyecto se desarrolla un material de apoyo interactivo que pretende mejorar la experiencia docente en los grupos de prácticas de la asignatura Teoría de Mecanismos a partir del curso 2019/2020. Se ha realizado utilizando el programa Jupyter Notebook, contenido en la distribución Anaconda, y programando en lenguaje Python.

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

Abstract

The aim of this Project is creating support material whith the goal of improving the learning experience in groups of practices of the subjetct “Teoría de Mecanismos” in the academic course 2019/2020 and on. It has been developed using Jupyter Notebook, which is contained in Anaconda, and programmed in Python.

Keywords: Teoría de mecanismos, Jupyter Notebook, Python.

Índice de figuras

Índice de ecuaciones

Tabla de símbolos

Símbolo	Nombre	Unidad	Símbolo unidad
G	Grado de libertad	Adimensional	
n	Número de barras	Adimensional	
PI	Pares binarios de 1 grado de libertad	Adimensional	
PII	Pares binarios de 2 grados de libertad	Adimensional	
q	Vector de coordenadas dependientes	Metro	m
ϕ	Matriz de restricciones	Varias unidades	
Φ_q	Matriz jacobiana	Varias unidades	
Δ_q	Variación de q	Metro	m
q_p	Vector velocidad	Metros/segundo	ms^{-1}
Φ_t	Derivada parcial de las ecuaciones de restricción respecto al tiempo	Varias unidades	
q_{pp}	Vector velocidad	Metros/segundo^2	ms^{-2}
Φ_{qp}	Derivada parcial del jacobiano respecto al tiempo	Varias unidades	

Capítulo 1:

Introducción

1. Introducción

1.1 Introducción y motivación del TFG

En los últimos años han sido diseñados diferentes mecanismos físicos con la intención de servir de ayuda para la asignatura Teoría de Mecanismos. Otro modo de prestar ayuda a la asignatura ha sido a base de aplicaciones, generalmente realizadas con Matlab.

Los profesores responsables de la asignatura han realizado diversos proyectos de innovación educativa, en los cuales hicieron tanto mecanismos físicos como aplicaciones.

El primero, de nombre MecaSENS [1], fue el complemento idóneo que se esperaba para la asignatura. En él, se construyeron varios mecanismos para su uso en las clases prácticas.

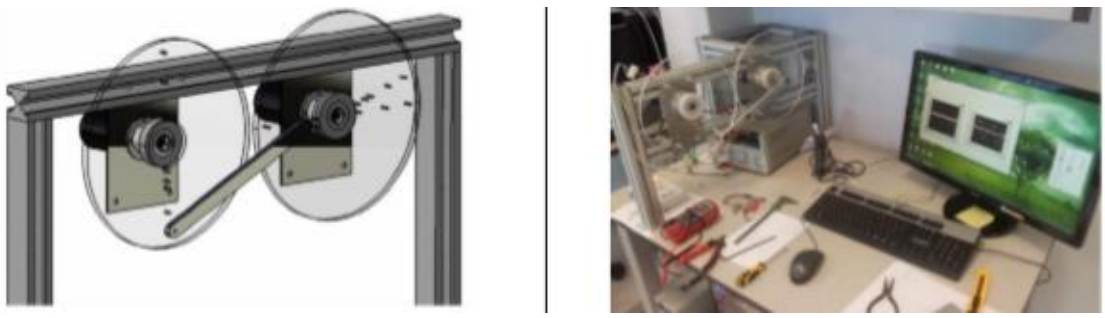


Figura 1: Prototipo virtual y diseño final (Fuente: [1])

Dado el éxito de este primer proyecto, se llevó a cabo un segundo que fue llamado MecaServer [4]. Éste seguía la misma temática que MecaSENS [1], pero tenía objetivos ligeramente distintos: con él se pretendía publicar material interactivo y crear un laboratorio remoto.

La principal ventaja que conlleva usar un laboratorio virtual es que tras realizar una simulación completa, se pueden obtener los resultados como gráficas (*Figura 3*). Esto significa una gran ventaja respecto a la solución manual de los problemas, dado que de manualmente las soluciones obtenidas son para un momento concreto.

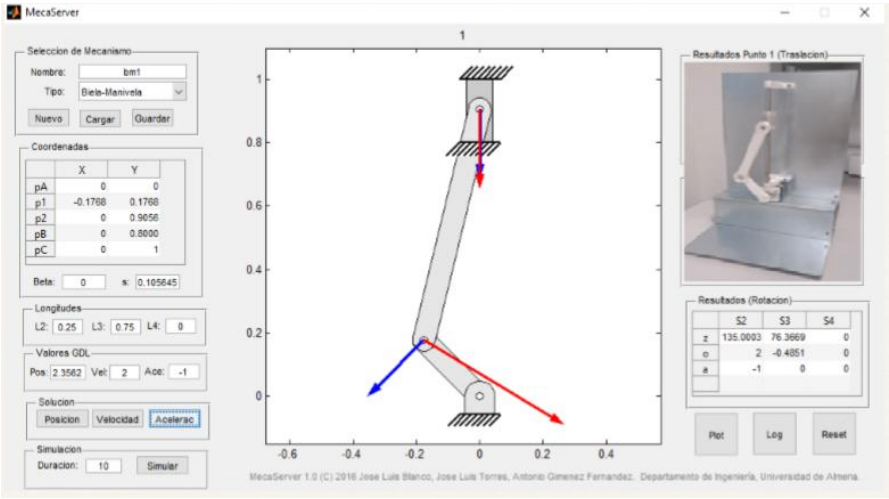


Figura 2: Interfaz del usuario (Fuente: [4])

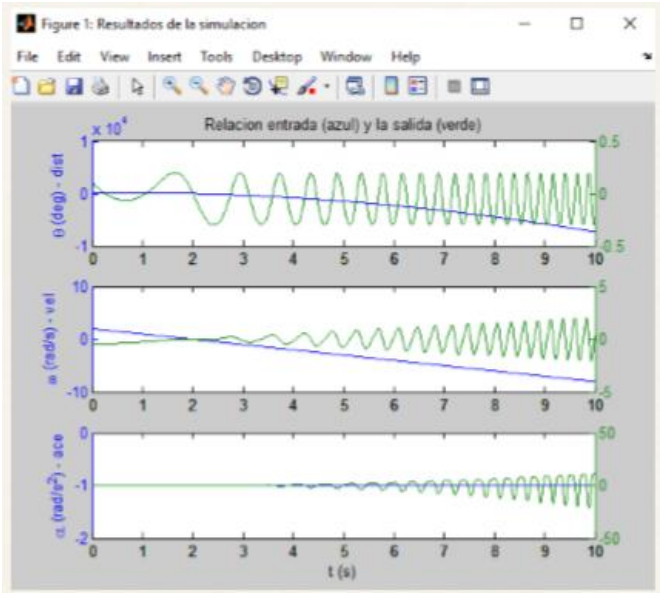


Figura 3: Gráficas obtenidas como resultado de la simulación (Fuente: [4])

Este TFG (Trabajo Fin de Grado) pretender ser una actualización de los proyectos mencionados, con la principal diferencia de utilizar otro entorno para el desarrollo del mismo, más concretamente Jupyter Notebook. Esto se debe a que Matlab se trata de un *software* propietario, lo que significa que es necesario tener una licencia para su uso, a diferencia de Jupyter, que se trata de una aplicación de código libre.

Otra particularidad de Jupyter Notebook es que permite el acceso desde cualquier lugar sin necesidad de instalar otros servicios y puede ser ejecutado en un servidor remoto. Solo es necesario un navegador de Internet, lo que permite que sea utilizado incluso en un teléfono móvil.

Otro elemento distintivo de este *software* es su funcionamiento por celdas, las cuales pueden ser ejecutadas individualmente o en conjunto, además de poder ser de diferentes modalidades como texto o código. Esta característica dota al programa de gran interés para el uso académico, ya que al poder estructurar el código tan gráficamente y poder insertar extensas y claras explicaciones facilita notoriamente su comprensión. Además, el hecho de ejecutar cada función por separado sirve para conocer en detalle la utilidad de cada bloque. Por otro lado, en las celdas de texto es posible añadir gráficas y fórmulas a las explicaciones. Todo ello en conjunto hace que al finalizar se tenga un documento que sirve tanto de apuntes o material de estudios y un ejercicio práctico.

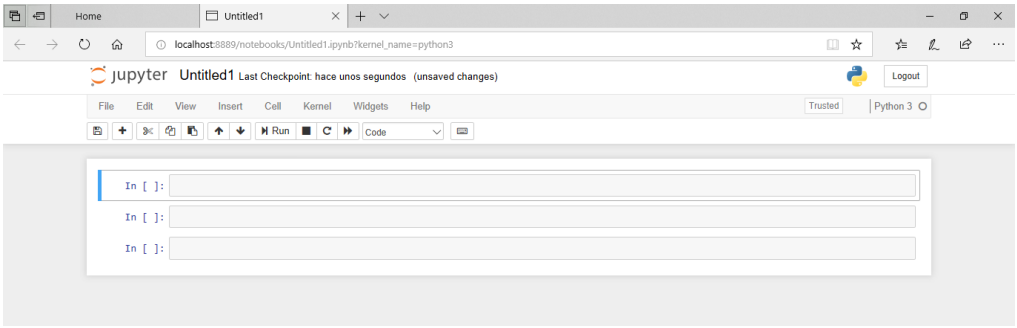


Figura 4: Apariencia de un Notebook

1.2 Objetivos

Lo que se espera conseguir con la realización de este proyecto es diseñar e implementar *notebooks Jupyter* que mejoren la experiencia docente en prácticas y que sirvan como material de estudio autónomo para el alumno en la asignatura Teoría de Mecanismos.

Consiste en desarrollar un programa que resuelva por métodos cinemáticos los problemas posición, velocidad y aceleración de los mecanismos más comunes, como son el Cuatro Barras, Cinco Barras, Biela-Manivela y Biela-Manivela invertida.

La intención es crear unos apuntes que ayuden a la comprensión de la asignatura en cuestión. Para ello el usuario debe introducir unos datos iniciales y verá en pantalla la solución junto con una explicación textual de los pasos a seguir y una simulación del mecanismo en movimiento.

Por otro lado, con la intención de aprovechar la ventaja que aporta Jupyter Notebook de poder ser ejecutado desde un servidor remoto, se ha planteado como objetivo que los programas creados puedan simularse en el teléfono móvil.

En el futuro se ampliará para que los programas puedan ser ejecutados por varios ordenadores o móviles simultáneamente desde un servidor remoto.

1.3 Contexto

En la UAL (Universidad de Almería) hay cuatro ramas de ingeniería industrial: Ingeniería Química Industrial, Ingeniería Eléctrica, Ingeniería Electrónica Industrial e Ingeniería Mecánica.

Estos cuatro grados en conjunto reúnen a una gran cantidad de alumnos, que los dos primeros años tienen un programa común a excepción de una asignatura en segundo, para ya especializarse a partir del tercer curso.

Una de estas asignaturas comunes es Teoría de Mecanismos, que se imparte en el Primer Cuatrimestre del segundo curso y es de carácter obligatorio y consta de 6 créditos. Es estudiada por una media de entre 150 y 175 alumnos cada curso académico.

Además, también se imparte en el Máster de Ingeniería Industrial, que se trata de un Máster Oficial. En este caso, la asignatura es del tipo Complementos de Formación, siendo también impartida en el Primer Cuatrimestre.

Son tres profesores los que imparten la asignatura en la actualidad: José Luis Torres Moreno, José Luis Blanco Claraco y Antonio Giménez Fernández.

Según la guía docente [6], es la primera asignatura del plan de estudios donde se adquieren los conocimientos básicos de la cinemática y dinámica de máquinas, sirviendo como introducción a la Teoría de Máquinas. Tras el aprendizaje de la asignatura, el alumno será capaz de analizar los grados de libertad de una cadena cinemática abierta o cerrada, aprenderá a analizar tanto cinemática como dinámicamente el comportamiento de un mecanismo y su equilibrado y además se estudian los principales tipos de engranajes, su clasificación y los criterios para diseñar trenes de engranajes de distintas tipologías.

En resumen, el TFG está orientado a facilitar el aprendizaje del análisis cinemático del comportamiento de un mecanismo.

En el marco establecido ya han sido desarrollados otros proyectos, habiendo sido utilizados ya en clase. Tres de estos proyectos son MecaSENS [1], MecaSENS-3D [2][3] y MecaServer [4]. Estos proyectos fueron llevados a cabo por los profesores del departamento.

1.4 Resumen de resultados

Inicialmente se consideraron diferentes alternativas de software para realizar las prácticas interactivas que se están planteando. En un principio, se empezó utilizando EasyJava. Este programa se trata también de una herramienta de *software* gratuito. Forma parte del proyecto *Open Source Physics*, cuya función es la de crear simulaciones de computadora discretas.

Este programa tiene la ventaja de tener un proceso de instalación sencillo. Basta con descargar el instalador y ejecutarlo. En pocos minutos ya se puede utilizar el programa sin inconvenientes.

No obstante, tras haber programado dos modelos de prueba, se planteó la posibilidad de utilizar Jupyter Notebooks. Hay algunos motivos por los que se propuso este nuevo *software*: mayor gratificación al programar, un resultado más gráfico y por tanto más didáctico y por último, la posibilidad de poder ejecutarse en el móvil.

Como aclaración, se considera más gratificante al programar al tratarse de un *software* en el que se escriben líneas de código como en otros lenguajes comunes como C o C++. Además, el resultado es más gráfico en el sentido de que al poder ver todas las líneas de código con las ecuaciones, aclaraciones y celdas completas de texto con explicaciones se ven más fácilmente los pasos seguidos.

1.5 Competencias

Durante el desarrollo de este TFG, se ha debido hacer uso de una serie de competencias. Las básicas adquiridas son:

- CB1 → Se ha demostrado poseer y comprender conocimientos en procedentes del cambo de estudio de la Teoría de Máquinas y de la programación, puesto que en su totalidad se han realizado cálculos relativos al ámbito de la ingeniería mecánica implementándolos en código Python.
- CB2 → Esta competencia ha sido adquirida al tener que aplicar los conocimientos adquiridos a lo largo de la carrera para crear algo que sirva en el ámbito docente, resolviendo problemas dentro del área de estudio en cuestión.
- CB4 → Con la elaboración del proyecto y la posterior defensa se adquirirá completamente la competencia de transmitir información, ideas, problemas y soluciones a un público especializado. Además, al ser un trabajo con fines educativos, servirá para transmitirlo también a personas no especializadas en el tema.

- CB5 → Se han desarrollado habilidades de aprendizaje necesarias para emprender estudios posteriores con un alto grado de autonomía, dado que ha sido un trabajo de gran envergadura desarrollado de forma autónoma, a diferencia de los múltiples trabajos propuestos durante la carrera, que estaban más enfocados al trabajo en equipo.

Respecto a las competencias transversales, las más destacables son:

- UAL002→ Habilidad en el uso de las TIC. Durante la elaboración del proyecto se han utilizado únicamente herramientas TIC, por lo que se puede asegurar con certeza que se posee la competencia.

- UAL003 → Capacidad de resolver problemas. Esta competencia se demuestra que ha sido adquirida ya que sin ella no habría sido posible finalizar el proyecto, dado la cantidad de problemas que han surgido durante el desarrollo.

- UAL004 → Comunicación oral y escrita en la propia lengua. Al elaborar un proyecto académico de esta magnitud, la comunicación escrita mejora notablemente, por lo que inevitablemente esta competencia es adquirida durante el desarrollo de la memoria.

Pasando a las competencias específicas, se va a hacer especial mención a cuatro:

- CT003 → Durante la carrera se estudiaron las asignaturas Teoría de Mecanismos y Programación. En este TFG se ha hecho uso de las dos, con la salvedad de que ha sido con un lenguaje de programación diferente, lo cual indica versatilidad para adaptarse a nuevas situaciones.

- CB001 → Para resolver los mecanismos era necesario realizar previamente cálculos complejos de sistemas de matrices, lo cual indica que esta competencia se posee.

- CB003 → Al realizar un trabajo exclusivamente de programación se demuestra tener conocimientos básicos sobre el uso y programación de los ordenadores y programas informáticos con aplicaciones en ingeniería.

- CRI007 → Del mismo modo que ocurre con la competencia anterior, se puede afirmar que se tiene esta competencia, dado que se trata de un trabajo de resolución de mecanismos con el fin de utilizarse posteriormente como material de estudio en la asignatura Teoría de mecanismos.

1.6 Cronograma

Actividad	*Fecha*	*Horas*
Estudio de EasyJava	5/07/2018 – 15/07/2018	10
Programación péndulo simple EasyJava	20/07/2018 – 7/08/2018	20
Estudio de lenguaje Python	23/09/2018 – 20/12/2018	20
Instalación de Anaconda y Notepad++	10/02/2019	3
Estudio asignatura Teoría de Mecanismos	10/02/2019 – 21/02/201	20
Desarrollo del mecanismo Cuatro Barras	3/03/2019 – 4/04/2019	100
Elaboración del anteproyecto	14/03/2019 – 21/03/2019	15
Desarrollo del mecanismo Cinco Barras	6/04/2019 – 13/04/2019	50
Desarrollo del mecanismo Biela-Manivela	16/04/2019 – 20/04/2019	35
Desarrollo del mecanismo Biela-Manivela Invertida	21/04/2019 – 28/04/2019	40
Desarrollo de la memoria de prácticas	1/05/2019 – 22/05/2019	55
	Total:	368

Se ha trabajado una media de cuatro horas diarias exceptuando un par de días en semana desde febrero, sumando un total de 368 horas.

1.7 Estructura del TFG

El TFG comienza con unas páginas previas en la que se incluyen agradecimientos y el resumen del tema, para posteriormente dividirse en cinco secciones principales. La primera de ellas es la introducción, que a su vez se divide en seis apartados: Introducción y motivación del TFG, objetivos, contexto, resumen de resultados, competencias, cronograma y estructura del TFG. El segundo gran apartado es la revisión bibliográfica, en la cual se citan proyectos similares al que se está planteando en este trabajo. El tercer apartado describe los materiales y métodos utilizados para la realización del proyecto. El cuarto, que ha sido nombrado como “Notebooks”, recoge lo que ha sido el desarrollo del TFG en sí. En el quinto apartado se expresan las conclusiones finales y se plantean trabajos futuros. Por último se encuentra la bibliografía y el apartado de anexos, donde se pueden ver los *Notebooks* al completo y el procedimiento para utilizar Jupyter Notebook en el teléfono móvil.

Capítulo 2:

Revisión Bibliográfica

2. Revisión Bibliográfica

2.1 Monografías sobre Teoría de Mecanismos

Alejo Avello Iturriagagoitia es Doctor Ingeniero Industrial y profesor ordinario de Ingeniería Mecánica en la Universidad de Navarra. Además, es director General de CEIT-Ik4 desde el año 2000. Esto es un proyecto europeo en el que colaboran diferentes empresas como Nokia, Ceit o ST microelectronics y otros centros de España, Francia, Finlandia e Italia, donde trabajan en las futuras redes celulares más allá del 5G.

Alejo es de interés para el desarrollo de este proyecto principalmente por un libro que publicó llamado "Teoría de Máquinas"[7]. Este libro es de gran utilidad para el uso docente, ya que se explica de manera eficiente todo lo relativo a la asignatura Teoría de Mecanismos y presenta diversos ejemplos y ejercicios propuestos, siendo una ayuda notoria para el estudiante.

En el libro se desarrollan los siguientes temas: análisis estructural de mecanismos, cinemática del sólido rígido, dinámica del solido rígido, análisis cinemático por métodos numéricos, análisis dinámico por métodos numéricos, levas, engranajes cilíndricos rectos, engranajes helicoidales, trenes de engranajes, equilibrado de rotores, vibraciones en sistemas con un grado de libertad y vibraciones en sistemas con varios grados de libertad.

En el transcurso de la elaboración del proyecto ha sido un material de consulta habitual, más concretamente el capítulo tres, en el que se explica lo relativo al análisis cinemático por métodos numéricos.

2.2 MecaSENS

MecaSENS [1] es un proyecto que fue desarrollado por los profesores responsables de la asignatura Teoría de Mecanismos con el fin de utilizarlo como complemento práctico para la asignatura en cuestión.

Se construyó una maqueta basada en el mecanismo de cuatro barras, también conocido como cuadrilátero articulado. Se trata del mecanismo plano más sencillo, dado que se compone únicamente de eslabones rígidos y pares de rotación. Es un ejemplo básico en el estudio de Teoría de Mecanismos dada su sencillez y sus diversas aplicaciones en máquinas y mecanismos.

Para llevar a cabo un análisis cinemático del mismo, es necesario conocer los datos de posición, velocidad y aceleración de al menos un punto de sus eslabones.

Estas variables, llamadas grados de libertad, se introducen en el sistema mediante la actuación controlada de un motor de corriente continua, que actúa como manivela.

Una vez se tiene el mecanismo con los parámetros introducidos, se procede a comprobar que el resto de eslabones tienen la misma velocidad que la estimada previamente mediante cálculos teóricos. Para ello se requiere la implementación de sensores de movimiento que deben ser monitorizados. Para los sensores se utilizaron dos unidades inerciales de medida (IMU), unos giróscopos de bajo coste y un encoder incremental en cuadratura. Para su monitorización, se desarrolló un sistema modular capaz de gestionar sincronizadamente señales heterogéneas, además de desarrollarse una interfaz que se encargaba de la representación y escritura de las variables que se registraban.

En el diseño y desarrollo del mecanismo se utilizaron herramientas CAD, lo que permitió un control dimensional de todos los elementos. La utilidad radicaba en que de esta manera se podía conocer la posición exacta de los sensores. Además, se trabajó con perfiles de aluminio, planchas de hierro y piezas construidas por impresión 3D.

Como resultado se obtuvo que las mediciones concordaban con bastante fidelidad a las predicciones de los modelos matemáticos, con lo cual se valida el diseño y construcción del prototipo y tiene la intención de motivar al alumnado al mostrar que los problemas estudiados sobre papel tienen una aplicación práctica. Cabe destacar que resultados obtenidos con esta maqueta fueron publicados en congresos internaciones de Ingeniería Mecánica.

Finalmente, gracias al proyecto se incorporó una práctica para ser realizada en el curso 2015-2016 en la cual los alumnos, organizados por parejas, debían introducir diferentes velocidades de entrada, almacenar los datos en una memoria USB y realizar el postprocesado y la validación de sus modelos teóricos.

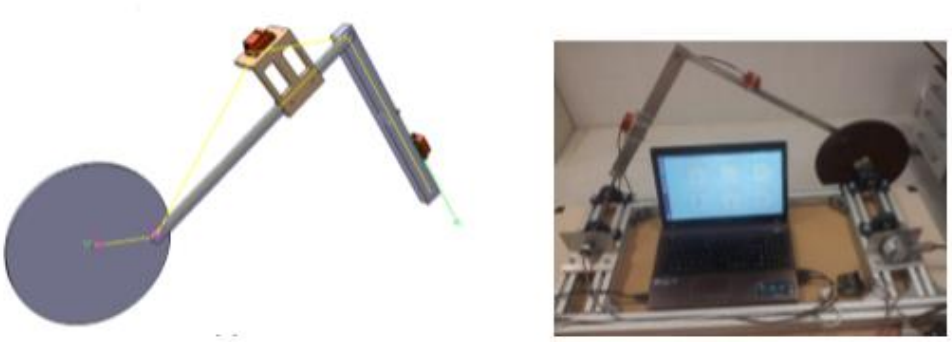


Figura 5: Prototipo virtual y diseño final MecaSens (Fuente: [1])

2.3 MecaSENS 3D

Gracias al proyecto MecaSENS [1] se construyó una maqueta se amplió la perspectiva de la asignatura a la par que se fomentó la interactividad del alumno y su motivación. No obstante, el coste y la complejidad de su fabricación eran elevados, por lo que no se podía contar con un equipo por cada pareja de alumnos. Este hecho llevaba a que el manejo de la maqueta por parte de los alumnos fuera muy limitado.

De este inconveniente surge el Proyecto de Innovación Docente Mecasens 3D [2][3], pretendiendo reducir los costes en la fabricación de mecanismos sensorizados con el objetivo de que cada grupo de trabajo pueda disponer de una maqueta durante las sesiones de prácticas.

Durante el primer año se construyeron los mecanismos biela-manivela y cinco barras a partir de diseños 3D. Se emplearon técnicas CAD en la fase de diseño y CAM para generar el código numérico que se ejecutó en la impresora 3D. Una vez obtenidas las piezas 3D, se necesita la incorporación de elementos comerciales que sirven tanto de soporte y unión de las piezas impresas como para dotar al mecanismo de actuadores, sensores y elementos de comunicación con un computador.

En el segundo año se puso en marcha el laboratorio multi-puesto en las prácticas, se diseñó un método de evaluación del grado de satisfacción del alumno en el que se realizaba una comparativa entre los objetivos alcanzados por los proyectos Mecasens y Mecasens 3D. Seguidamente se realizó la evaluación para posteriormente analizar los resultados y por último, se divulgó en el ámbito educativo.

Figura 6: Mecasens 3D (Fuente: [3])

2.4 Mecaserver

Como continuación de los proyectos MecaSens [1] y MecaSens3D [2][3], se presentó un proyecto de trabajo para un grupo docente titulado Servidor de contenido interactivo y gestión de un laboratorio remoto de Ingeniería Mecánica MecaServer [4]. A diferencia de los proyectos anteriores, donde había un gran trabajo en el ámbito de ingeniería mecánica e instrumentación, en MecaServer la parte fundamental son las comunicacione

Se plantearon los siguientes objetivos:

a) Publicación de un libro sobre cinemática y dinámica de máquinas, cuya principal aportación sea la introducción de material sobre nuevas técnicas de resolución de problemas, como por ejemplo las basadas en métodos numéricos.
b) Publicación de material interactivo basado en aplicaciones JavaScript.
c) Creación de un laboratorio remoto, con el objetivo de que las maquetas realizadas a partir de MecaSens y MecaSens 3D puedan ser utilizadas desde cualquier lugar con conexión a internet.
d) Centralización de sistema de gestión y utilización de los recursos que se pretenden desarrollar.

Se lograron cumplir todos los objetivos. El libro ya ha sido publicado, y para este trabajo de fin de grado ha sido de utilidad. Además, haciendo referencia al segundo objetivo, se ha desarrollado un espacio web con aplicaciones correspondientes a diferentes problemas en el ámbito de la ingeniería mecánica, acompañado de un software cuyo desarrollo fue pensado para su utilización en Matlab. El tercer objetivo también ha sido logrado al dotar a MecaServer de un sistema de comunicaciones hardware y software entre los equipos del laboratorio y el servidor instalado en el CPD de la UAL. Para cumplir el último objetivo fueron necesarios una serie de sensores y actuadores controlados por una tarjeta de adquisición de datos y una placa Arduino. Utilizando LabVIEW en el computador se pueden administrar todos los elementos mencionados, para seguidamente conectar el ordenador a la subred del CITE-IV junto con una cámara IP, contando los dos con dirección estática. En LabVIEW se deben ejecutar dos ejecuciones. Una de ellas es responsable de la comunicación con los elementos hardware, y otra es un servidor interno que conecta las variables correspondientes a entradas y salidas de la aplicación anterior con el servidor principal ubicado en el CPD de la UAL.

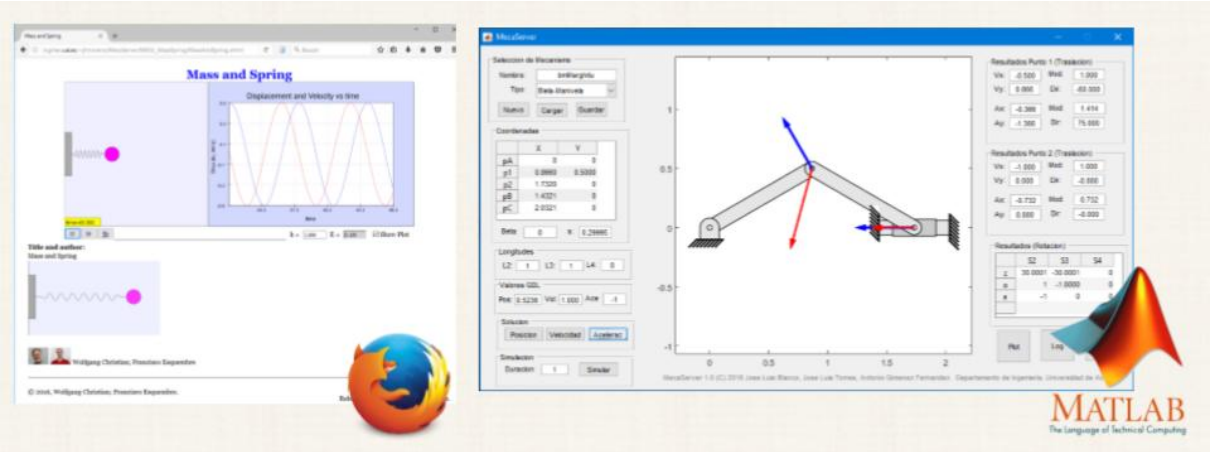


Figura 7: Interfaz Mecaserver (Fuente: [4])

2.5 Laboratorio Virtual para la Enseñanza de Control Climático de Invernaderos

Este trabajo fue realizado por J. L. Guzmán, F. Rodríguez, M. Berenguel y S. Dormido, siendo los tres primeros del Departamento de Lenguajes y Computación de la Universidad de Almería y el último del Departamento de Informática y Automática de la UNED.

El objetivo del proyecto era crear un laboratorio virtual para la enseñanza del control climático de invernaderos. Esta herramienta aporta un modelo completo de invernadero con un conjunto de controladores específicos, con acceso a través de una interfaz gráficamente estructurada. Con ello se logra que los alumnos pongan en práctica los conocimientos teóricos adquiridos en clase sin restricciones espacio-temporales.

El interés de este trabajo para el trabajo de fin de grado que se está desarrollando, es que se el programa que se utilizó para crear el laboratorio virtual fue EasyJava Simulations.

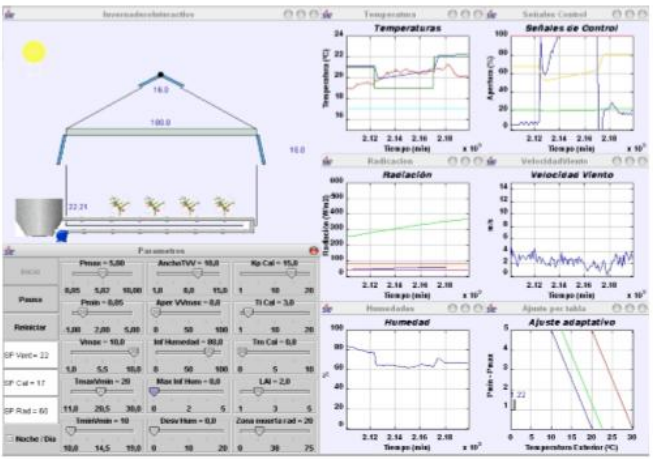


Figura 8: Laboratorio virtual para control climático en invernaderos (Fuente: [8])

Capítulo 3:

Materiales y métodos

3. Materiales y métodos

3.1 EasyJava

EJSS (Easy JavaScript Simulations) [9] es una herramienta de *software* gratuito. Se diseñó para crear simulaciones de sistemas en tiempo discreto. Estas simulaciones son programas de ordenador que intentan reproducir un fenómeno visualizando diferentes estados que puede tener.

En este entorno, a diferencia de otros, no es necesario introducir un código complejo que se desarrolla en largas líneas de comandos. Al ejecutar la consola, aparece una pantalla en la que se puede añadir una descripción de la función que se va a realizar, así como dos apartados más: modelo y vista.

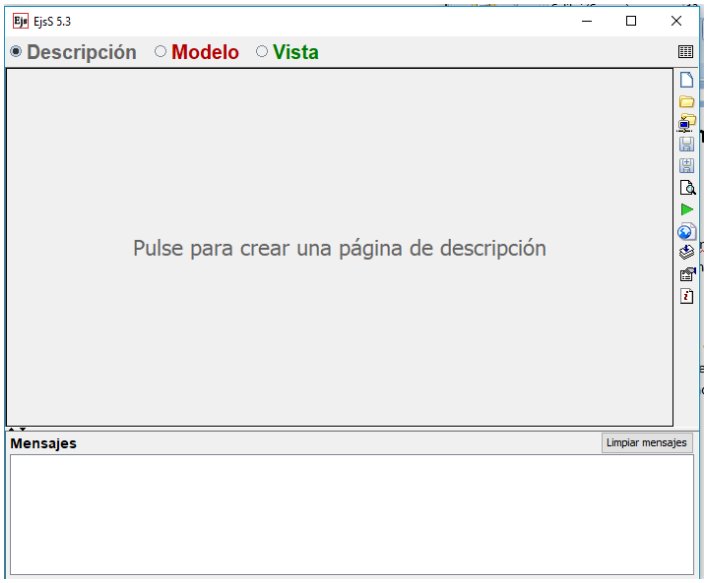


Figura 9: Consola de EJSS

Al seleccionar la pestaña “Modelo” aparecen diferentes apartados: Variables, inicialización, evolución, relaciones fijas, propio y elementos. Los más relevantes para el caso de interés son: Variables, evolución y relaciones fijas. En variables se indica el nombre de los datos conocidos, mientras que en evolución se indica la función que representa una incógnita que varía con el tiempo, como puede ser velocidad y aceleración. Por último, en relaciones fijas se expresan los datos que aun sin saber su valor se conoce cómo está relacionada con otras variables.

Por último está la pestaña "Vista". En ella se diseña la simulación que aparecerá posteriormente en pantalla. Se divide en dos partes, siendo la primera es el árbol de elementos, espacio que inicialmente aparece en blanco y donde se van arrastrando diferentes objetos y asignando parámetros a cada uno para su posterior ejecución. En la segunda parte se encuentran los elementos para la vista, que son los que se pueden añadir al árbol y que se dividen a su vez en interfaz, elementos para la vista, elementos de dibujo 2D y elementos de dibujo 3D. En el apartado interfaz se pueden añadir objetos como botones y graficas, mientras que en los dos apartados de elementos de dibujo lo que se añaden son las formas que en conjunto darán lugar a la animación.

En las figuras que se muestran a continuación se puede ver la apariencia de las pestañas "Modelo" y "Vista" para el caso de un péndulo simple, ejemplo que fue realizado cuando la intención era elaborar el proyecto con EJSS y que se propone en el manual de EJSS [10].

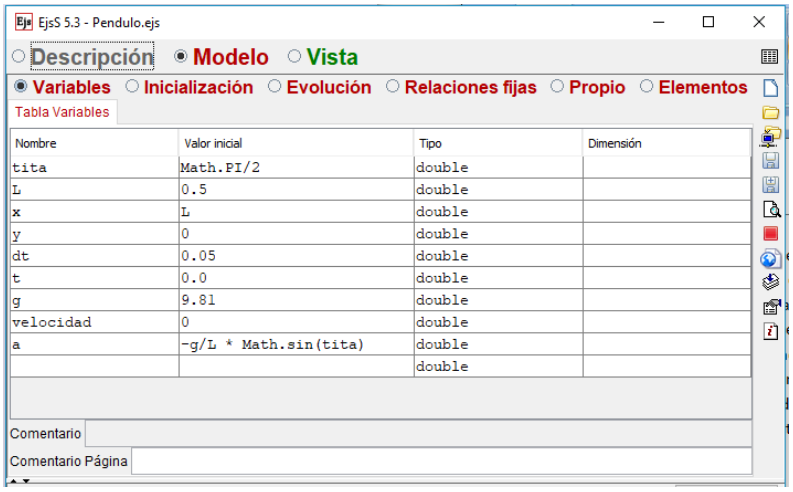


Figura 10: Variables del péndulo

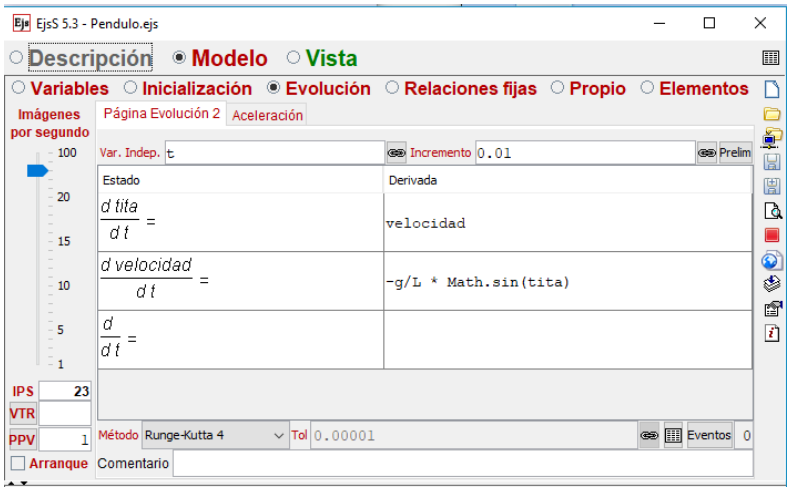


Figura 11: Evolución péndulo

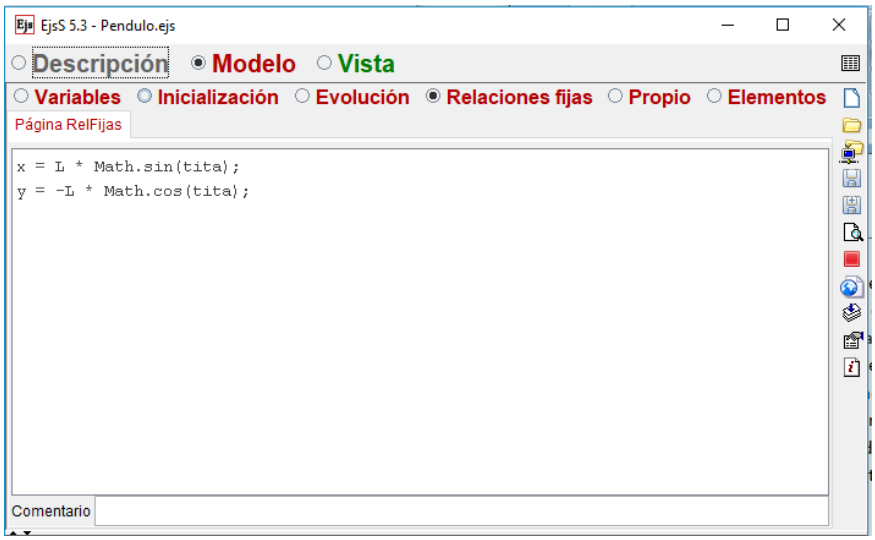


Figura 12: Relaciones fijas péndulo

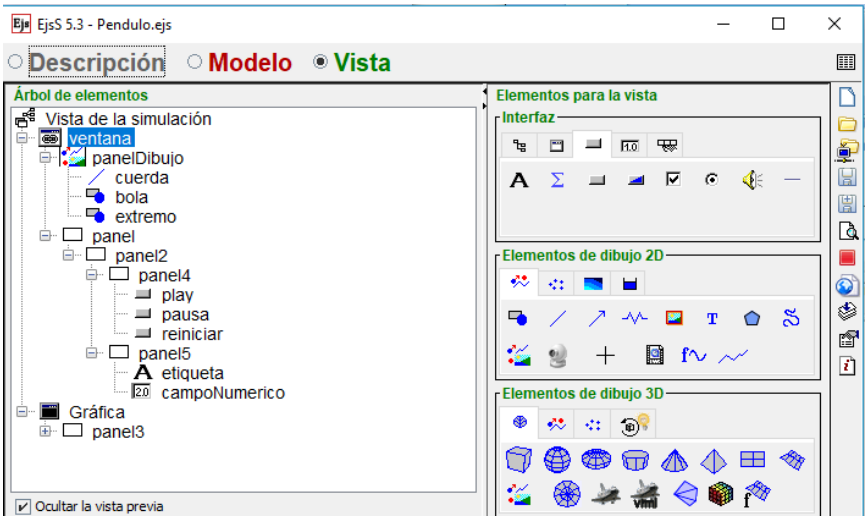


Figura 13: Vista Péndulo

Una vez establecidas todas las relaciones entre parámetros y completado el árbol de elementos se puede ejecutar. En el caso del péndulo simple se simuló su movimiento (*Figura 14)* y las gráficas que representan la variación del ángulo respecto del tiempo, la velocidad angular y la aceleración angular *(Figura 15)*.

Sin embargo, decidió descartarse este entorno para realizar el proyecto dado que su modelo de programar dificultaba notablemente la implementación de las ecuaciones de Teoría de Mecanismos.

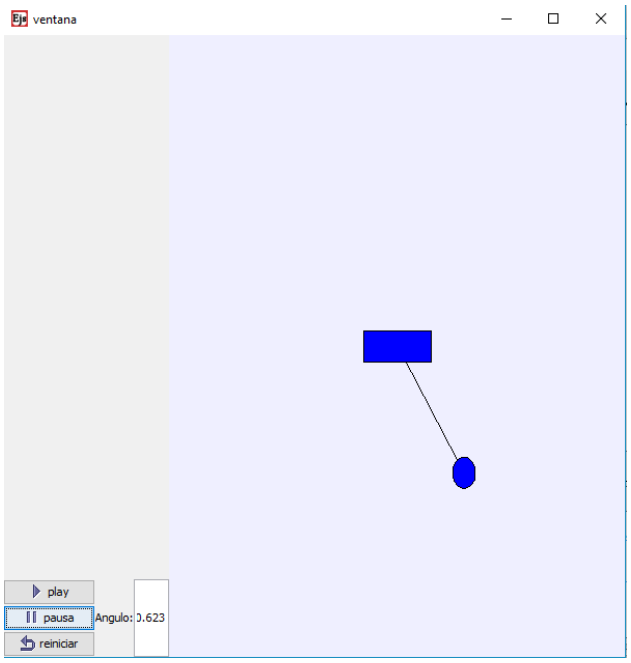


Figura 14: Animación péndulo

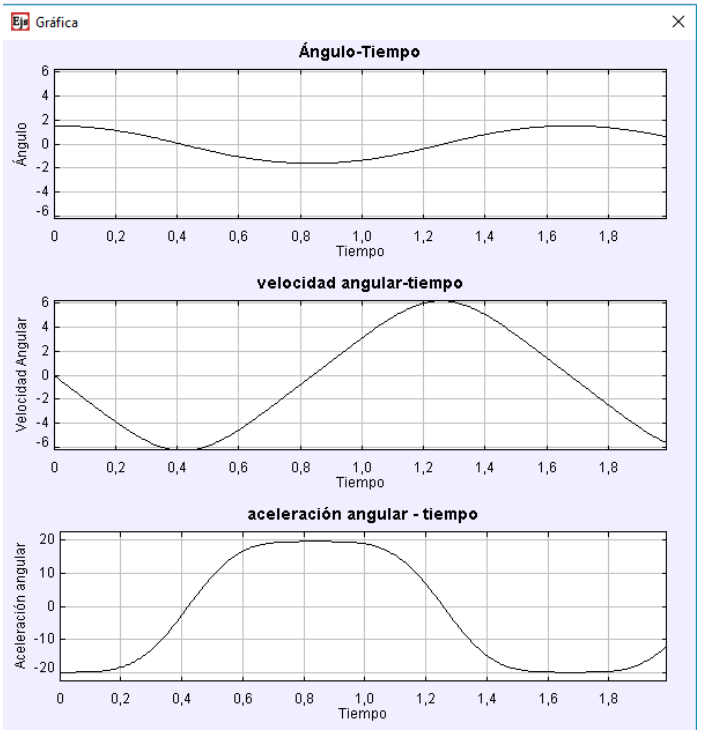


Figura 15: Gráficas péndulo

3.2 Python

En un curso de iniciación de programación en lenguaje Python que se puede encontrar en la plataforma coursera [11] se realiza una introducción detallada al lenguaje, y es la base que se ha utilizado para realizar este apartado, ya que en él se hace un resumen de dicha introducción.

Los computadores necesitan instrucciones claras y precisas para poder realizarlas, y para entenderlas es necesario una comunicación con ellas. Esto se logra mediante los lenguajes de programación.

Los humanos se comunican gracias al lenguaje, que se trata de un conjunto de símbolos y sonidos con un significado para ambas partes de la comunicación. El conjunto de símbolos que forman un lenguaje es la sintaxis, y su significado la semántica.

En el proceso de comunicación entre humanos hay un emisor que transmite una información, llamada mensaje. La persona que lo recibe se llama receptor, y si el mensaje está codificado en un lenguaje no conocido por el receptor, no hay comunicación.

Un computador, a diferencia de los humanos, comprende un lenguaje binario basado en ceros y unos. Este lenguaje se traduce en impulsos de corriente donde cero significa que no hay corriente y uno que sí. Sin embargo, los humanos no saben comunicarse de esta manera, por lo que se han inventado lenguajes para poder hablar con el computador.

Una de las primeras personas en empezar a pensar en este tipo de lenguajes fue Ada Lovelace, noble del siglo XIX e hija del famoso poeta Lord Byron. Era matemática y en vida trabajó con la idea de una máquina analítica. Realmente, se trataba de una propuesta de un conjunto de instrucciones pensada para que una máquina resolviera problemas matemáticos. Es considerada una de los primeros programadores de la historia y sus ideas fueron la base sobre la que se establecieron los primeros principios de los lenguajes de programación.

Hay muchos lenguaje y muy variados, y además han evolucionado con el paso del tiempo. Por ello se organizan en tres generaciones.

La primera generación se denomina lenguaje máquina o código máquina, y es el lenguaje que el computador puede interpretar directamente: ceros y unos. No obstante, ya se ha mencionado que este lenguaje es muy difícil de programar para los humanos, razón por la cual empiezan a aparecer los lenguajes de programación de bajo nivel.

El lenguaje de programación de bajo nivel es ligeramente más complejo que los ceros y unos, aunque siguen siendo instrucciones muy atómicas. Con ellas se ejerce un control directo sobre el hardware o la parte física de la computadora como pueden ser el teclado y la pantalla. Un ejemplo de este lenguaje es el ensamblador. Sin embargo, sigue siendo un

lenguaje con notoria complejidad para ser utilizado como base de la programación. Así aparecen los lenguajes denominados de alto nivel.

Los lenguajes de alto nivel surgieron en torno a los años 50. Este tipo de lenguajes codifica las instrucciones de la forma más similar al lenguaje humano. Con ellos se pueden construir programas con una sintaxis y semántica sencillas de entender, los cuales son capaces de enviar mensajes al computador que se traducen en unos y ceros. Entre los programas más famosos de alto nivel están por ejemplo el lenguaje C, C++ o Java. En este grupo se encuentra el lenguaje Python.

El lenguaje de programación Python debe su nombre a un grupo de humorista inglés llamado Monty Python, que llegó a alcanzar gran popularidad en los años 60 y 70. El creador de este lenguaje de programación, Guido Van Rossum, era un gran fan de estos humoristas.

Una de las características que lo han hecho tan famoso es su simplicidad, dado que se parece tanto a la forma que tienen los humanos de expresarse que facilita mucho su comprensión. Además de su simplicidad, tiene otras dos características que lo hacen uno de los lenguajes más interesantes de la actualidad. Una es ser multiplataforma, es decir, que los programas creados en este lenguaje pueden funcionar en cualquier sistema operativo, bien sea Windows, Apple o Linux. La segunda es que se trata de un lenguaje multiparadigma, lo que significa que soporta más de un paradigma de programación, es decir, que soporta distintos estilos de programación y puede utilizarse para cualquier tipo de programa. Por ejemplo, podría usarse para realizar las funciones de una calculadora. También podría usarse para el desarrollo de páginas web, incluso sirve para controlar los movimientos de un robot mediante el uso de algoritmos complejos. Últimamente se está usando sobre todo para *Data Science*, para hacer programas muy sofisticados que permitan analizar grandes datos como por ejemplo los millones de datos que se recogen en los centros científicos astronómicos, como por ejemplo el observatorio ALMA de Chile. Estas dos características han hecho que sea uno de los lenguajes más populares en la actualidad.

Por todo esto, empresas como Google, Dropbox o Netflix utilizan este lenguaje de programación como base para sus programas.

Lo más importante es que Python nunca deja de crecer, cuenta con una gran comunidad de desarrolladores que innova y crea nuevos códigos cada día, lo que hace que aparezcan cada día nuevos códigos, nuevas librerías y nuevos programas que hacen cada vez el lenguaje más completo y más capaz de adaptarse a las necesidades del momento. Para unirse a la comunidad de desarrolladores de Python hay que registrarse en *www.python.org*.

3.3 Numpy

Según la fuente [13], los módulos o librerías son la forma que tiene Python de almacenar instrucciones o variables en un archivo, de manera que puedan usarse posteriormente en un script o en una instancia interactiva del intérprete, como por ejemplo en Jupyter Notebook, para no tener que definirlas cada vez. El hecho de que Python permita la separación del programa en módulos tiene la ventaja de que podrán reutilizarse en otros programas o módulos, siendo necesario para ello importar los módulos que se pretendan utilizar.

Python organiza los módulos, con extensión **.py** en paquetes, que son directorios que contienen ficheros **.py** y un archivo de inicio llamado **_init_.py.** Los paquetes son una manera de estructurar los espacios de nombres de Python usando nombres de módulos con puntos. Por ejemplo, un módulo llamado A.B designa un submódulo de nombre B que se encuentra en un paquete A.

Para importar un modulo, se utiliza la función *import* seguida del nombre del paquete más el nombre del módulo sin el .py. Si las rutas son largas se puede generar un alias añadiendo "as".

```
In [ ]: import modulo # importar un módulo
        import paquete.modulo1 # importar un módulo que está de
        ntro de un paquete
        import paquete.subpaquete.modulo1 # importar un módulo
        que está dentro de un subpaquete

        # Si las rutas (lo que se conoce como "namespace" son l
        argas,se pueden generar alias por medio del modificado
        "as"

        import modulo as m
        import paquete.modulo1 as pm
        import paquete.subpaquete.modulo1 as psm
```

Figura 16: Llamada de módulos (Fuente: [13])

Es recomendable seguir un orden a la hora de importar los módulos. Hay que llamarlos al principio del programa, en orden alfabético e importando primero los propios de Python, después los de terceros y para finalizar los de la aplicación.

Python viene con una biblioteca de módulos estándar que incluye una gran variedad de módulos con muy diversas funciones. Esta librería suele estar incluida en los instaladores de Python para plataformas Windows. Algunos ejemplos de módulos incluidos en esta biblioteca estándar son el módulo *Os* y el módulo *math*. El primero ofrece funciones para interactuar con el sistema operativo como por ejemplo borrar la consola, decir en qué directorio estamos o cambiar a otro directorio, mientras que el módulo *math* incluye funciones trigonométricas, logarítmicas y estadísticas, entre otras.

Sin embargo, para la realización de programas más complejos es necesario algo más que las librerías de la biblioteca estándar, ya que aunque por ejemplo ofrezca funciones matemáticas, acaban siendo insuficientes. Por ello habrá que utilizar librerías no estándar.

Una de ellas es NumPy (*Numerical Python).* Ligdi González desarrolla en las fuentes [14] y [15] una introducción muy completa sobre esta librería, explicando su función y algunas acciones que pueden realizarse con ella. Según su definición, es la librería principal para la informática científica, ya que proporciona potentes estructuras de datos, implementando matrices y matrices multidimensionales, lo que garantiza cálculos eficientes con matrices.

NumPy array, es decir, el arreglo de matrices de Numpy, se presenta como un potente objeto de matriz N-dimensional con forma de filas y columnas, donde hay varios elementos almacenados en sus respectivas ubicaciones de memoria.

Para crear una matriz NumPy, se usa la función np.arrray(). Hay que colocarle una lista y de manera opcional se puede especificar el tipo de datos a utilizar. Para poder usar esta función, es necesario haber importado la librería definiéndola como np.

```
In [1]: import numpy as np

In [2]: array = np.array ([[1,2,3,4], [5,6,7,8]], dtype=np.int64)
        print(array)

        [[1 2 3 4]
         [5 6 7 8]]
```

Figura 17: Definición de array

Ocasionalmente se requiere la creación de matrices vacías, lo cual hace referencia a la necesidad de marcadores de posición iniciales que posteriormente pueden ser rellenados.

Permite realizar numerosas funciones con matrices. Por ejemplo se puede crear una matriz donde todos los valores sean igual a uno o a cero o bien crear matrices que se llenan con valores espaciados uniformemente o con valores constantes o aleatorios.

```
In [1]: import numpy as np

In [2]: # Crear una matriz de ceros 3filasx4columnas
        ceros = np.zeros((3,4))
        print (ceros)

[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
```

Figura 18: Crear matriz

```
In [3]: # Crear una matriz de números aleatorios 2x2
        aleatorios = np.random.random((2,2))
        print (aleatorios)

[[0.53767634 0.56908231]
 [0.52409276 0.88384306]]

In [4]: #Crear una matriz vacía
        vacia = np.empty((3,2))
        print (vacia)

[[0. 0.]
 [0. 0.]
 [0. 0.]]
```

Figura 19: Matriz vacía y matriz de números aleatorios

```
In [5]: # Crear una matriz de unos 4filas x 3columnas
        unos = np.ones ((4,3))
        print (unos)

[[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]

In [6]: #Crear una matriz con un solo valor
        full = np.full((2,2), 8)
        print (full)

[[8 8]
 [8 8]]
```

Figura 20: Matriz de unos y matriz con un solo valor

```
In [7]: #Crear una matriz con valores espaciados uniformemente
        espacio1 = np.arange (0,30,5)
        print (espacio1)

        espacio2 = np.linspace(0,2,5)
        print(espacio2)

    [ 0  5 10 15 20 25]
    [0.  0.5 1.  1.5 2. ]
```

Figura 21: Matriz con valores espaciados uniformemente

3.4 Matplotlib

Matplotlib es una librería no estándar, al igual que NumPy, que sirve para presentar datos de forma gráfica. Más concretamente, es una librería de trazado para gráficos 2D, muy flexible y con muchos valores predeterminados incorporados.

El alias utilizado a la hora de importar la librería suele ser plt. Para preparar un ejemplo sencillo, elegimos unos datos y posteriormente a la hora de graficarlos se define el color y el grosor de la línea. También se puede incluir un eje que muestre la leyenda de los datos. Por último se indica que debe mostrarlo en pantalla con la función show.

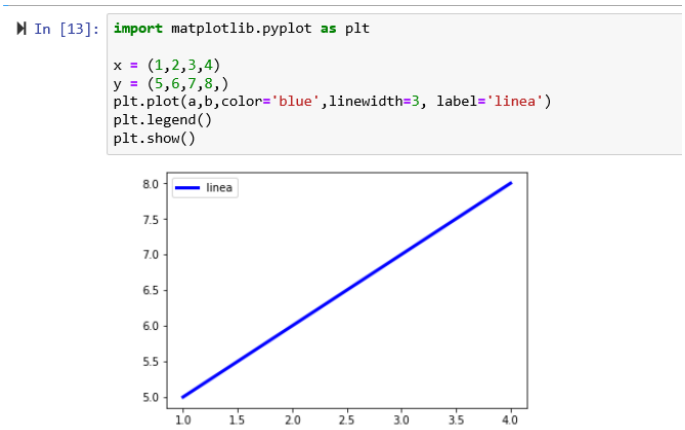


Figura 22: Ejemplo gráfica (Fuente: [16])

Hablando de Matplotlib se hace referencia a todo el paquete de visualización de datos de Python. Por otro lado, *pyplot* es un módulo dentro del paquete Matplotlib, siendo esta la razón por lo que frecuentemente se importa la librería como *matplotlib.pyplot*.

El módulo proporciona una interfaz que permite crear figuras y ejes de forma implícita y automática para lograr la trama deseada. Esto es útil principalmente cuando se desea trazar algo rápidamente sin definir ninguna figura o eje, como pasó en el ejemplo de la figura. No se definieron los componentes pero se obtuvo el resultado esperado, es decir, graficar los datos.

Otra biblioteca de gran utilidad dentro del paquete Matplotlib es *animation.* Con ella se pueden crear simulaciones y guardarlas como una animación HTML. Para ello hay que importar varias librerías. Para el ejemplo de simular la función seno, es necesario importar la librería Numpy, que se encarga de calcular la función y llamar al número pi, y también será necesario importar Matplotlib, usando dentro de ella *pyplot* y *animation*. Con *pyplot* se realizan funciones como definir los límites de la representación y mostrar en pantalla la figura, y con *animation* se crea la animación propiamente dicha. Por último, se exporta a HTML.

In [1]:
```
%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(( 0, 2))
ax.set_ylim((-2, 2))

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

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

def animate(i):
    x = np.linspace(0, 2, 1000)
    y = np.sin(2 * np.pi * (x - 0.01 * i))
    line.set_data(x, y)
    return (line,)
```

Figura 23: Código animación (Fuente: [20])

```
In [2]: anim = animation.FuncAnimation(fig, animate, init_func=init,
                                       frames=100, interval=20,
                                       blit=True)

        plt.show()

        HTML(anim.to_html5_video())
```

Out[2]:

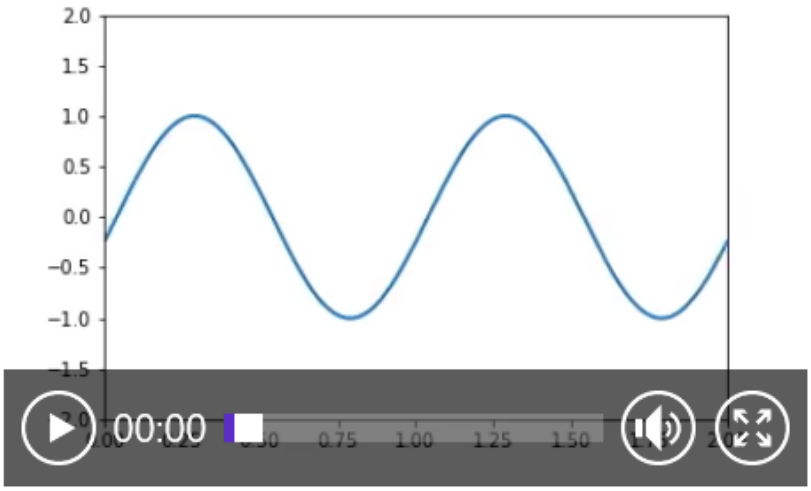


Figura 24: Código animación y simulación (Fuente: [20])

3.5 Distribución de Python: Anaconda

En la entrada de la Wikipedia [21] correspondiente al programa Anaconda, se puede encontrar una descripción de su utilidad. La define como una distribución libre y abierta, así como la manera más sencilla de programar utilizando Python y R, siendo utilizada en ciencia de datos (*data science*) y en aprendizaje automático (*machine learning*). Esto incluye procesamiento de grandes volúmenes de información, cómputos científicos y análisis predictivo. Se orienta a la simplificación del despliegue y administración de los paquetes de software y se utiliza en Linux, Windows y Mac OS X, y tiene en torno a 11 millones de usuarios alrededor del mundo.

Es el estándar de la industria para desarrollar, probar y entrenar en una sola máquina, lo cual permite a los científicos de datos individuales (*individual data scientists):*

- Una descarga rápida de más de 1500 paquetes de datos científicos (*data science packages)* para Python y R.

- Administrar librerías, dependencias y entornos con Conda.

- Desarrollar y capacitar modelos de aprendizaje automático y aprendizaje profundo con *scikit-learn*, *TensorFlow* y *Theano*.

- Analizar datos con escalabilidad y rendimiento con *Dask*, *NumPy*, *pandas* y *Numba*.

- Visualizar resultados con *Matplotlib*, *Datashader* y *Holoviews*.

La descarga de Anaconda se hace desde su página oficial, y tras la instalación se obtiene todo el entorno de desarrollo para Python y otras utilidades para la ciencia de datos.

Los pasos a seguir aparecen descritos en la fuente [23] y son los siguientes:

1. Ingresar al sitio oficial de Anaconda y acceder al enlace “Downloads”.

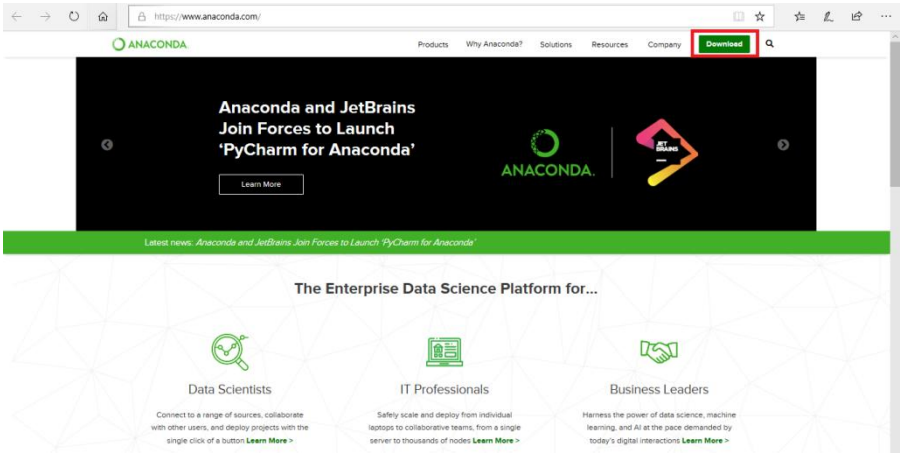


Figura 25: Sitio oficial de Anaconda (Fuente: [22])

2. Seleccionar el sistema operativo. Para la realización de este proyecto fue seleccionado Windows.

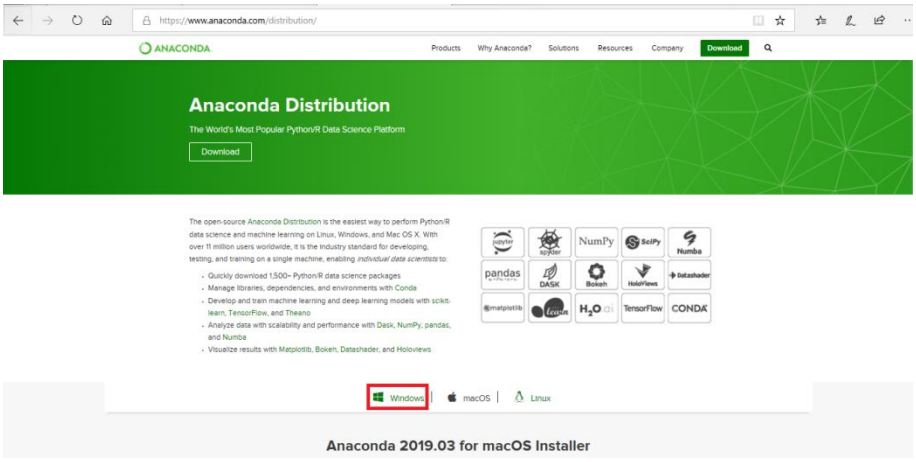


Figura 26: Elección de sistema operativo (Fuente: [23])

3. Elegir la arquitectura del procesador dependiendo del sistema del computador en el que se va a realizar la instalación. En este caso 64 bits. Elegir la versión de Python que se desee instalar en función del sistema operativo. En este caso fue seleccionada la versión 3.7.

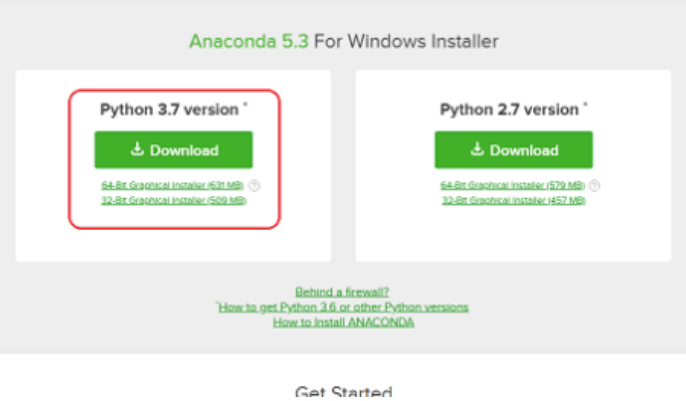


Figura 27: Elección de versión (Fuente: [23])

4. Una vez se tiene el instalador, se procede a ejecutarlo. La instalación es intuitiva, tan solo hay que presionar los botones “Siguiente” y “Acepto” cuando corresponda.

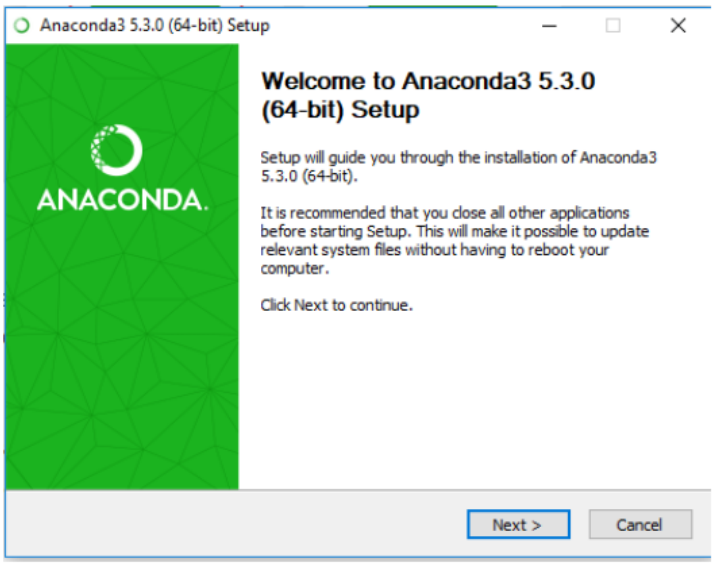


Figura 28: Instalador Anaconda parte 1 (Fuente: [23])

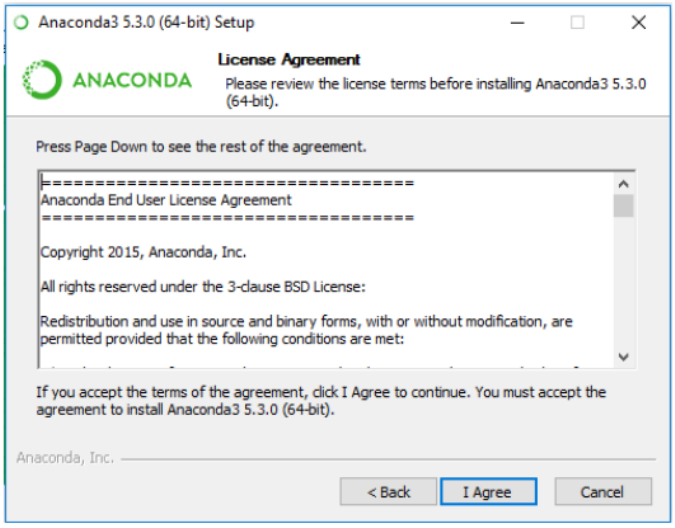


Figura 29: Instalador Anaconda parte 2 (Fuente: [23])

5. Al aceptar los términos de la licencia de Anaconda, habrá que elegir si la instalación va a estar disponible para todos los usuarios o solo para el administrador. Seguidamente, habrá que elegir el lugar donde se guardarán los archivos necesarios para ejecutar Anaconda. Tras esto, aparecerá una pantalla donde se decide instalar PATH o no, dependiendo de las preferencias personales y pudiéndose agregar posteriormente si no se marca la casilla. Seleccionarlo significa que se podrá utilizar Anaconda en el símbolo del sistema directamente.

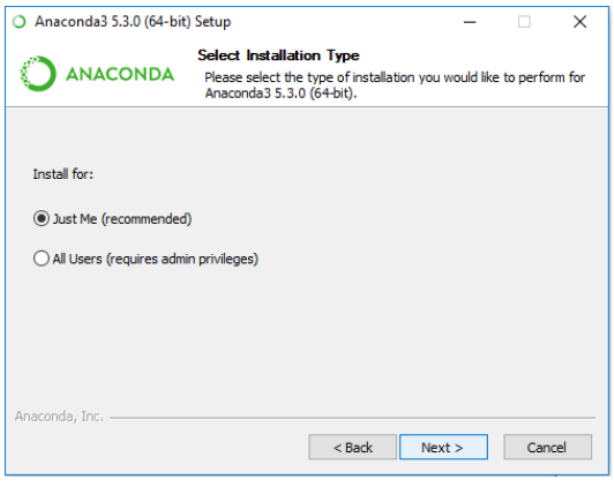


Figura 30: Tipo de Instalación (Fuente: [23])

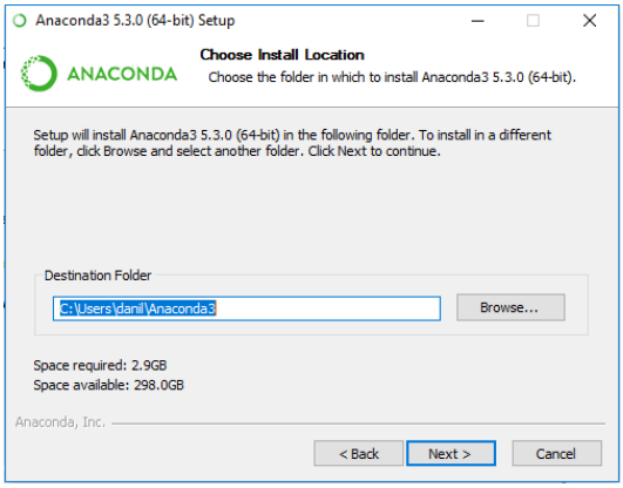


Figura 31: Localización de instalación (Fuente: [23])

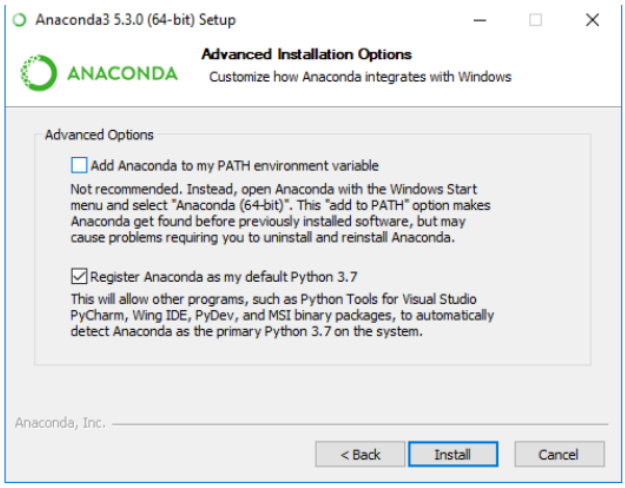


Figura 32: PATH (Fuente: [23])

6. Tras presionar el botón “Instalar”, comenzará el proceso de instalación. Durante el mismo se preguntará si se quiere instalar Visual Studio Code, un editor de código de fuente desarrollado por Microsoft para Windows. Se marcará la casilla “Saltar”, ya que en principio no es necesario y también puede instalarse a posteriori.

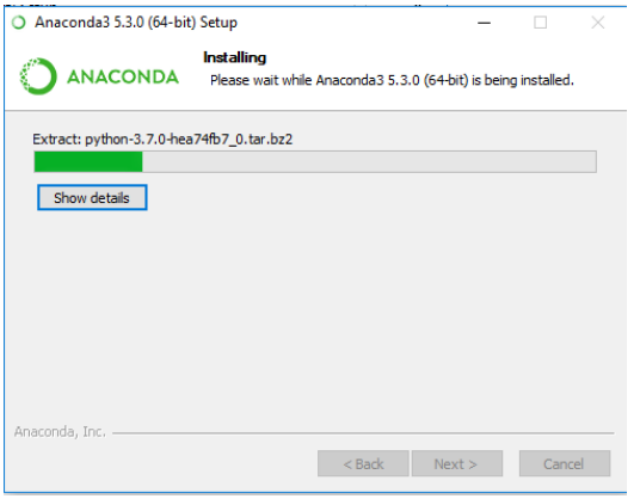


Figura 33: Instalación Anaconda (Fuente: [23])

7. Cuando finaliza la instalación, se puede acceder a *Anaconda Navigator*. Generalmente hay dos pantallas importantes: *Home* y *Enviroments*. En *Home* aparecen las aplicaciones que vienen instaladas por defecto en el *workspace* actual de Anaconda en Root o raíz. En este espacio aparece *Jupyter Notebooks*, que era el objetivo principal de la instalación y con el que ha sido llevado a cabo el proyecto. Por otro lado, en Enviroments se tienen todos los paquetes que han sido instalados de raíz.

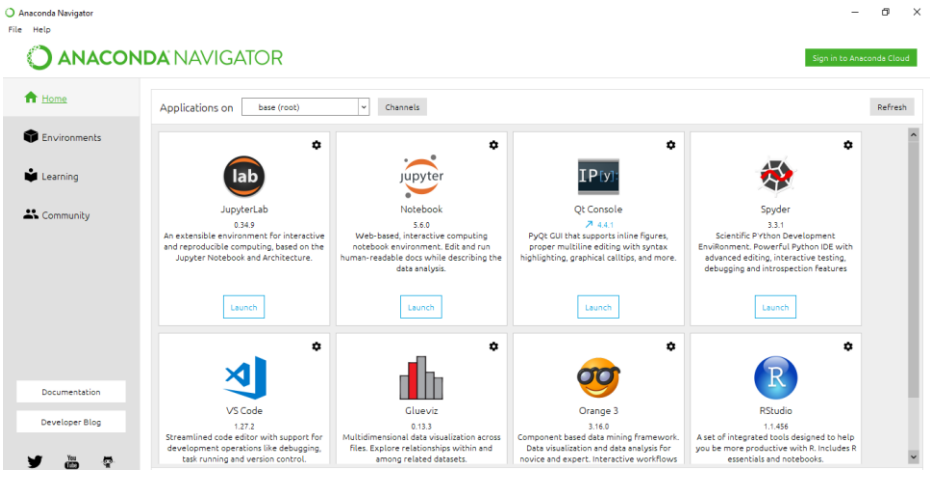


Figura 34: Home Anaconda

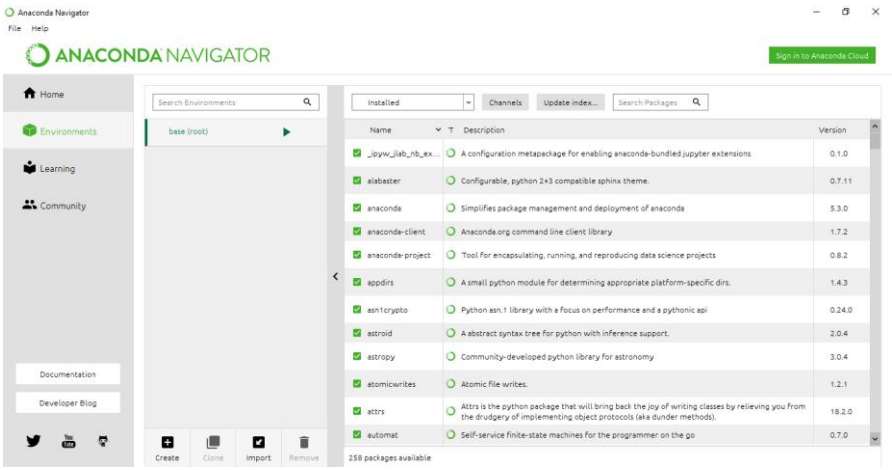


Figura 35: Enviroments Anaconda

3.6 Teoría de Mecanismos

3.6.1 Terminología

Utilizando la definición empleada por Alejo Avello en su libro “Teoría de Máquinas” [7], una máquina es una combinación de cuerpos dispuestos de tal forma que producen un trabajo. Actualmente connota la capacidad para transmitir niveles de fuerza/momento considerables, como ocurre con el motor de un automóvil, por ejemplo. Sin embargo, cuando la fuerza o momento involucrados son pequeños, la función principal del dispositivo pasa a ser transmitir o modificar el movimiento. En este caso, en lugar de hablar de máquinas se habla de mecanismos, como puede ser un reloj. Por otro lado, la frontera entre máquinas y mecanismos es difusa, lo que hace que se utilicen ambos términos de manera intercambiable.

Los mecanismos se componen por elementos con posibilidad de movimiento, a excepción de uno, conocido como elemento fijo. Los elementos se componen por partículas materiales con desplazamiento relativo entre ellas cuando el elemento se encuentra bajo la acción de fuerzas exteriores. No obstante, estos desplazamientos suelen ser tan pequeños que pueden ser despreciados sin cometer un error significativo, lo que lleva a que se considere los elementos como sólidos rígidos. Los elementos se clasifican en función del número de conexiones que tengan con otros elementos en binarios, ternarios, etc. Por ejemplo, un elemento con dos conexiones se llamará binario.

Las uniones entre elementos, de nombre “pares cinemáticos”, permiten algunos movimientos relativos entre elementos e impiden otros.

Un ejemplo sería la bisagra de una puerta: es un par cinemático que permite la rotación de la puerta respecto al eje vertical e impide el resto de movimientos. Los pares, al igual que los elementos se clasifican en binarios, ternarios, etc., dependiendo del número de elementos que confluyan en el par.

Un conjunto de elementos móviles unidos mediante pares cinemáticos se denomina cadena cinemática. Según su definición, en una cadena cinemática no hay elemento fijo. La cadena cinemática es una generalización del concepto de mecanismo, por lo que un mecanismo puede definirse como una cadena cinemática en la que uno de sus elementos se ha hecho fijo. Las cadenas cinemáticas pueden ser abiertas o cerradas.

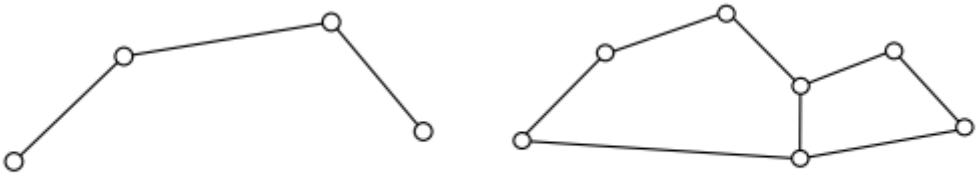


Figura 1.1: Cadena abierta. Figura 1.2: Cadena cerrada.

Figura 36: Cadena abierta y cerrada (Fuente: [7])

Hay dos conceptos análogos para cadenas cinemáticas y mecanismos. La movilidad (M) corresponde a la cadena cinemática y hace referencia al número de parámetros que es necesario utilizar para definir por completo su posición. Para los mecanismos aparece el concepto de grado de libertad (G). Como se ha dicho anteriormente, un mecanismo es una cadena cinemática en la cual un elemento se ha hecho fijo. Esta fijación de un elemento es equivalente a restarle 3 grados de libertad a la movilidad de la cadena para el caso de trabajar en el plano o restarle 6 en el caso del espacio, debido a que un elemento tiene tres grados de libertad en el plano y seis en el espacio.

Atendiendo a su movimiento, se denomina manivela a un elemento si da vueltas completas respecto a un eje fijo, balancín si oscila respecto de un eje fijo y biela si tiene un movimiento general.

Los pares cinemáticos se denominan de clase I, II, III, etc., dependiendo del número de grados de libertad que permitan en el movimiento relativo entre los dos elementos que une. Para el caso de un par de clase I, solo se permite un grado de libertad, para la clase II se permiten dos grados de libertad y así sucesivamente.

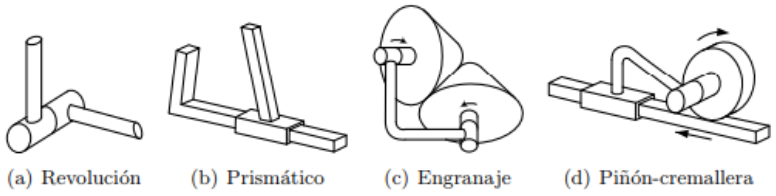


Figura 37: Pares de clase I (Fuente: [7])

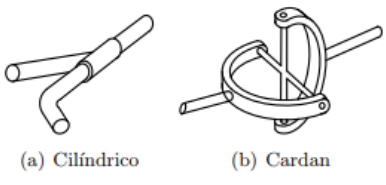


Figura 1.10: Pares de clase II.

Figura 1.11: Par Esférico (Clase III)

Figura 38: Pares de clase II y III (Fuente: [7])

El cálculo del número de grados de libertad de un mecanismo se realiza aplicando la siguiente expresión matemática:

$$G = 3 \cdot (n - 1) - 2 \cdot PI - PII \qquad (3.1)$$

Siendo:

G → Número de grados de libertad.

n → Número de barras.

PI → Cantidad de pares binarios de un grado de libertad.

PII → Cantidad de pares binarios de dos grados de libertad.

3.6.2 Análisis cinemático

Hay diversos modos de resolver problemas de reducido tamaño, dada su comodidad y facilidad de interpretación. No obstante, en problemas complejos o de un tamaño más considerable estos métodos se quedan cortos y pasan a ser sustituidos por métodos numéricos.

Para desarrollar un método matemático que pueda ser programado fácilmente es necesario crear en primer lugar un modelo matemático simple y eficiente del mecanismo. Crear este

modelo implica transformar conceptos como par cinemático o elemento en un conjunto de datos numéricos dispuestos en forma de matriz o vector. Esto es el proceso de modelización. Este procedimiento puede realizarse con diferentes tipos de coordenadas, como pueden ser las independientes, dependientes, relativas dependientes, de punto de referencia o naturales.

Las coordenadas naturales definen de forma absoluta la posición de cada elemento y comúnmente se sitúan en los pares. Con este tipo de coordenadas, las ecuaciones de restricción pueden ser de sólido rígido y de par cinemático, pudiéndose dar el segundo caso solo en determinados pares.

Las ecuaciones de sólido rígido tienen una deducción directa. Si tenemos una barra de longitud $L_{12,}$ debe imponerse la condición de que el punto 1 permanezca a distancia constante del punto 2.

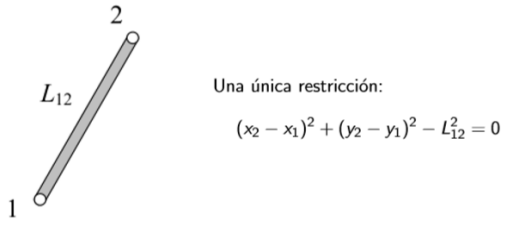


Figura 39: Condición de sólido rígido (Fuente: [24])

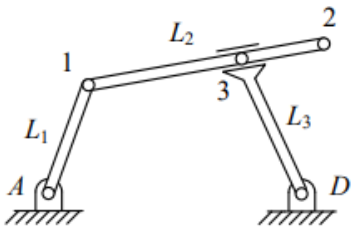


Figura 40: Ejemplo mecanismo (Fuente: [7])

De par cinemático pueden darse dos ecuaciones, y surgen de las limitaciones al movimiento que impone la deslizadera rígida.

Para el ejemplo de la figura, se tiene en primer lugar que el segmento 12 debe permanecer formando ángulo constante con el segmento 3D, lo que es equivalente a decir que su producto escalar debe permanecer constante. En segundo lugar, el punto 3 debe desplazarse sobre el segmento 12, lo que significa que el producto vectorial entre 12 y 13 debe ser nulo. De esta manera se obtendrían las ecuaciones:

$$(x_2 - x_1)(x_3 - x_D) + (y_2 - y_1)(y_3 - y_D) - k = 0$$
$$(x_3 - x_1)(y_2 - y_1) - (y_3 - y_1)(x_2 - x_1) = 0$$

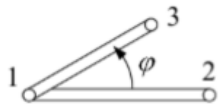
Figura 41: Ecuaciones de restricción para deslizadera

Las normas a seguir para modelar de forma correcta un mecanismo en coordenadas naturales son:

a) En cada articulación debe situarse un punto.
b) En los pares prismáticos deben existir como mínimo tres puntos alineados. Dos de ellos sirven para definir el eje y el tercero para la deslizadera.
c) Cada sólido debe tener al menos dos puntos, ya que en caso contrario no se podría determinar la orientación del cuerpo.
d) También pueden utilizarse tantos puntos adicionales como se considere necesario.

Habitualmente, también se definen las posiciones de los elementos mediante coordenadas mixtas: naturales y relativas. Estas son más utilizadas cuando hay que añadir una restricción que defina el ángulo entre dos eslabones.

Con el estudio cinemático de un mecanismo se pretende conocer su movimiento independientemente de las fuerzas fluctuantes. Los problemas cinemáticos son geométricos y están orientados al análisis del movimiento en términos de posición, velocidad y aceleración.

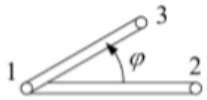


Se añade ϕ al vector $\mathbf{q}$.
Usando el producto escalar: $\rightarrow\vec{\mathbf{12}} \cdot \vec{\mathbf{13}} = L_{12}L_{13}\cos\phi$.
Usando el producto vectorial: $\rightarrow\vec{\mathbf{12}} \times \vec{\mathbf{13}} = L_{12}L_{13}\sin\phi$.

Podemos añadir una de las dos restricciones a $\mathbf{\Phi(q)}$, pero mejor ambas (evitar singularidades):

$$\begin{aligned}(x_2 - x_1)(x_3 - x_1) + (y_2 - y_1)(y_3 - y_1) &= L_{12}L_{13}\cos\phi \\ (x_2 - x_1)(y_3 - y_1) - (y_2 - y_1)(x_3 - x_1) &= L_{12}L_{13}\sin\phi\end{aligned}$$

Figura 42: Restricción de ángulo parte 1 (Fuente: [24])



Cuando el ángulo ϕ (orientación *absoluta*) es relativo a tierra (12 es tierra), las ecuaciones se simplifican:

$$\begin{aligned} L_{12}(x_3 - x_1) + \cancelto{0}{(y_2 - y_1)}(y_3 - y_1) &= L_{12}L_{13}\cos\phi \\ L_{12}(y_3 - y_1) - \cancelto{0}{(y_2 - y_1)}(x_3 - x_1) &= L_{12}L_{13}\sin\phi \end{aligned}$$

$$\downarrow$$

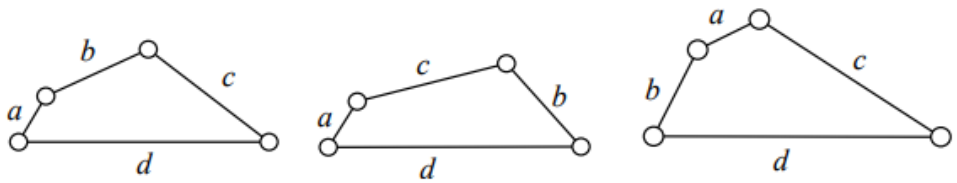
$$\begin{aligned} x_3 - x_1 &= L_{13}\cos\phi \\ y_3 - y_1 &= L_{13}\sin\phi \end{aligned}$$

Figura 43: Restricción de ángulo parte II (Fuente: [24])

3.6.3 Ley de Grashoff

Para este apartado también se utilizará de referencia el libro de Alejo Avello [7]. Comienza explicando que el cuadrilátero articulado es un mecanismo muy utilizado debido a su sencillez de construcción, versatilidad y facilidad de diseño. Se puede comportar como una doble manivela, manivela-balancín y doble balancín. En el doble manivela, las dos barras articuladas al elemento fijo actúan como manivelas, lo que quiere decir que dan revoluciones completas. En el caso de manivela-balancín, un elemento da revoluciones completas y otro oscila entre dos posiciones extremas. Para el caso de un cuadrilátero de doble balancín, ambos elementos oscilan entre posiciones extremas.

Teniendo cuatro longitudes tales que **a<b<c<d** hay tres configuraciones posibles de cadenas cinemáticas distintas, las cuales se llamarán **I**, **II** y **III**.

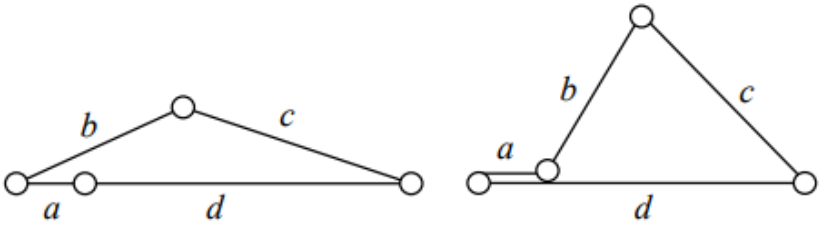


(a) Configuración I (b) Configuración II (c) Configuración III

Figura 44: Configuraciones posibles (Fuente: [7])

Para estudiar las condiciones geométricas que deben darse para que una barra sea manivela o balancín, hay que saber cuándo una barra puede dar vueltas completas con respecto a

otra. Si se demuestra que una barra puede dar vueltas completas con respecto a otra en una de las tres configuraciones de la figura, también podrá hacerlo en las otras dos.



(a) Barras extendidas (b) Barras opuestas

Figura 45: Posiciones límite (Fuente: [7])

Para que la barra **a** pueda dar vueltas completas con respecto a la barra **d**, deben poder alcanzase las dos posiciones representadas en la figura anterior, en las cuales **a** y **d** aparecen alineadas. Estas configuraciones se conocen como posiciones límite.

Se puede escribir:

$$b + c > a + d \qquad (3.2)$$

$$d - a > c - b \qquad (3.3)$$

La primera ecuación expresa una propiedad de los triángulos que establece que la suma de las longitudes de dos de sus lados es mayor que la del tercero. Por su parte, la segunda ecuación expresa la propiedad que establece que la diferencia de las longitudes de dos lados es menor que la del tercero.

La segunda ecuación también se puede escribir como:

$$d + b > c + a \qquad (3.4)$$

Para obtener estas expresiones se parte de la hipótesis de que **a** puede dar vueltas completas con respecto a **d**. Se podría seguir el mismo procedimiento con el resto de combinaciones entre barras para obtener una tabla de desigualdades como la siguiente:

Barras	*Desigualdad* I	*Desigualdad* II
a y d	$b+c>a+d\ (iii)$	$b+d>a+c\ (i)$
a y c	$b+d>a+c\ (i)$	$b+c>a+d\ (iii)$
a y b	$c+d>a+b\ (i)$	$b+c>a+d\ (iii)$
b y d	$a+c>b+d\ (ii)$	$a+d>b+c\ (iv)$
b y c	$a+d>b+c\ (iv)$	$a+c>b+d\ (ii)$
c y d	$a+b>c+d\ (ii)$	$a+d>b+c\ (iv)$

Figura 46: Desigualdades (Fuente: [7])

En la tabla se aprecia que solo hay cuatro tipos de desigualdades diferentes: **i**, **ii**, **iii** y **iv**. Las de tipo **i** se cumplen automáticamente, como por ejemplo, que la suma de las dos barras más largas es mayor que la suma de las dos barras más cortas. Las de tipo **ii** son imposibles, como que la suma de las dos barras más cortas es mayor que la suma de las dos más largas. Por último, **iii** y **iv** son opuesta la una a la otra y pueden cumplirse o no. La desigualdad **iii** es conocida como desigualdad de Grashoff.

De la tabla se pueden sacar las siguientes conclusiones:

a) La única barra que puede dar vueltas completas con respecto a las demás es la pequeña. Esto se puede verificar comprobando en la tabla que cuando la barra **a** no aparece en la primera columna siempre aparece una condición imposible.

b) Si la barra pequeña puede dar vueltas completas con respecto de otra barra, también podrá hacerlo con respecto al resto. Es decir, para que la barra **a** pueda dar vueltas completas tiene que cumplir la desigualdad de Grashoff. Una vez se tenga esto, se satisfarán también las condiciones necesarias para que de vueltas completas con respecto a **b**, **c** y **d**.

Si se cumple la desigualdad de Grashoff, habrá varios movimientos posibles para el cuadrilátero:

1. Doble manivela si el elemento **a** es fijo. Las barras contiguas al fijo dan vueltas completas, por lo que son manivelas.
2. Manivela-balancín si **a** es contiguo al elemento fijo.
3. Doble balancín si el elemento **a** es opuesto al fijo.

En caso de no cumplirse la desigualdad de Grashoff, el cuadrilátero será de doble balancín.

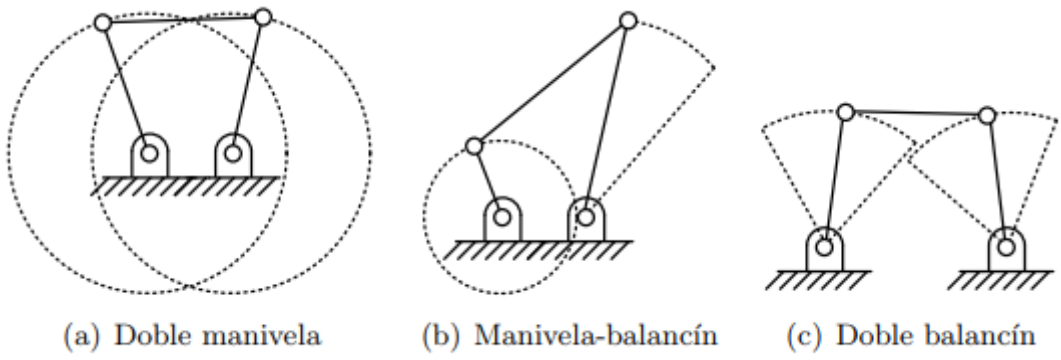


Figura 47: Grashoff (Fuente: [7])

3.6.4 Problema Posición

El problema posición o problema de montaje, consiste en calcular la posición **q** de todos los elementos del mecanismo a partir de la posición de los grados de libertad **z**.

Las **n** incógnitas del vector **q** están ligadas por **m** ecuaciones de restricción, que se agrupan en el vector de restricciones **ϕ** de dimensión **mx1**.

Seguidamente, hay que calcular la matriz jacobiana $\boldsymbol{\phi_q}$. Esta matriz de dimensiones **mxn** está compuesta por las derivadas parciales de las ecuaciones de restricción con respecto al vector de coordenadas dependientes.

Por ejemplo, partiendo de un vector **q** tal que:

$$q = \begin{pmatrix} X_1 \\ Y_1 \\ X_2 \end{pmatrix} \qquad (3.5)$$

y un vector **ϕ** tal que:

$$\varphi(\mathrm{q}, \mathrm{t}) = \begin{pmatrix} \varphi_1 \\ \varphi_2 \\ \varphi_3 \end{pmatrix} \qquad (3.6)$$

El jacobiano se construiría de la forma:

$$\varphi_{\mathrm{q}}(1,1) = \text{Derivada de } \varphi_1 \text{ respecto a } \mathrm{X}_1$$

$$\varphi_{\mathrm{q}}(1,2) = \text{Derivada de } \varphi_1 \text{ respecto a } \mathrm{Y}_1$$

$$\varphi_{\mathrm{q}}(1,3) = \text{Derivada de } \varphi_1 \text{ respecto a } \mathrm{X}_2$$

Habría que repetir este proceso para construir la matriz elemento a elemento.

Una vez se tiene **ϕ** y $\boldsymbol{\phi_q}$, ya se puede obtener el valor de las coordenadas dependientes del vector **q.** Para ello se parte de la ecuación:

$$\varphi\left(\mathrm{q} + \Delta_{\mathrm{q}}\right) = \varphi(\mathrm{q}) + \varphi_{\mathrm{q}} \cdot \Delta_{\mathrm{q}} = 0 \qquad (3.7)$$

De donde se despeja:

$$\varphi_{\mathrm{q}} \cdot \Delta_{\mathrm{q}} = -\varphi \qquad (3.8)$$

Esta ecuación se convertiría en:

$$Ax = b \qquad (3.9)$$

Donde **A** es $\boldsymbol{\phi_q}$ ampliado con las filas extra y **b** es **–ϕ** ampliado**.** Habría que despejar **x**, pero no se puede operar con matrices dividiendo de esta manera. Por lo tanto, es necesario multiplicar a ambos lados de la igualdad por la izquierda por $\mathbf{A^{-1}}$, o lo que es lo mismo, por la inversa del jacobiano.

$$\varphi_q{}^{-1} \cdot \varphi_q \cdot \Delta_q = \varphi_q{}^{-1} \cdot -\varphi \quad (3.10)$$

Al multiplicar el jacobiano por su inversa toma el valor la unidad, por lo que quedaría:

$$\Delta_q = \varphi_q{}^{-1} \cdot -\varphi \quad (3.11)$$

Y para terminar, el nuevo valor de **q** sería:

$$q = q + \Delta_q \quad (3.12)$$

Hay que repetir este proceso hasta que el vector **ϕ** se aproxime a 0, lo que indicaría la validez del vector **q** calculado. Sin embargo, hay datos iniciales para los que el mecanismo no converge. Por ejemplo, si se tiene un mecanismo cuatro barras y se establece que la posición del primer apoyo es $\mathbf{X_A = 0, Y_A = 0}$ y la del segundo $\mathbf{X_B = 50\ Y_B = 0}$, al decir que las longitudes de las barras son, por ejemplo $\mathbf{L_1 = 1, L_2 = 2}$ y $\mathbf{L_3 = 3}$ el problema no tendría solución ya que tal posición no existe. Por ello, es necesario imponer un límite de iteraciones, como por ejemplo **100**, y si llega a dicho límite se puede determinar que el mecanismo no converge.

Para poder realizar las operaciones indicadas con matrices, se necesita que las dimensiones sean correctas. Esto significa que a **ϕ** y a $\boldsymbol{\phi_q}$ habrá que añadirle tantas filas como grados de libertad tenga el mecanismo. Para ello, si por ejemplo se tiene solo un grado de libertad habrá que asignar un valor conocido a una de las variables dependientes. Esto se traduce en que para la fila que añadamos a **ϕ** su valor será **0**, ya que su valor permanece invariable. Por otro lado, para la matriz $\boldsymbol{\phi_q}$ se tendría que la variable conocida tendría valor 1 y el resto **0**.

Por último, se puede medir el error en el cálculo de **q** obteniendo el módulo del vector $\boldsymbol{\Delta_q}$, teniendo que ser lo más próximo a 0. También se puede saber si el mecanismo convergerá calculando el rango de la matriz jacobiana. Si su rango es igual al número de coordenadas dependientes, convergerá.

3.6.5 Problema de Velocidad

El problema velocidad consiste en determinar las velocidades de todas las variables del mecanismo una vez se sabe su posición **q** y la velocidad de los grados de libertad.

Se parte de la igualdad:

$$\varphi_q = 0 \quad (3.13)$$

Derivando se obtiene:

$$\varphi_q q_p + \varphi_t = 0 \quad (3.14)$$

Siendo $\mathbf{q_p}$ el vector velocidad, $\boldsymbol{\phi_q}$ el jacobiano y $\boldsymbol{\phi_t}$ la derivada parcial de las ecuaciones de restricción respecto al tiempo. Para las ecuaciones de sólido rígido el valor de esta derivada es **0**. Solo tendría un valor no nulo la correspondiente al ángulo, que en ese caso tendría la velocidad que se le indique.

De este modo, la expresión quedaría:

$$\varphi_q q_p = -\varphi_t \qquad (3.15)$$

Este sistema de ecuaciones tiene infinitas soluciones y por tanto hay que ampliar añadiendo un dato conocido de velocidad, lo que se hace añadiendo una fila a la matriz de coeficientes del lado izquierdo y un dato a la columna del lado derecho de la ecuación por cada gado de libertad.

De esta forma se llega a un sistema de ecuaciones lineal matricial de la forma:

$$Ax = b \qquad (3.16)$$

Habría que multiplicar en ambas partes de la igualdad por la **A** invertida en el lado izquierdo, del mismo modo que se hizo en el problema de posición. De esta manera quedaría:

$$x = A^{-1}b \qquad (3.17)$$

3.6.6 Problema Aceleración

El problema aceleración trata de determinar las aceleraciones de todas las variables del mecanismo conociendo la posición **q**, la velocidad $\mathbf{q_p}$ y las aceleraciones de los grados de libertad.

Se parte de la ecuación que se obtiene tras derivar la ecuación inicial para el problema de velocidad, es decir:

$$\varphi_q q_p + \varphi_t = 0 \qquad (3.14)$$

Se deriva por segunda vez:

$$\varphi_{qp} q_p + \varphi_q q_{pp} + \varphi_t = 0 \qquad (3.18)$$

Se despeja $\boldsymbol{\phi_q q_{pp}}$:

$$\varphi_q q_{pp} = -\varphi_{qp} q_p - \varphi_t \qquad (3.19)$$

Siendo $\boldsymbol{\phi_q}$ el jacobiano, $\mathbf{q_{pp}}$ el vector aceleración, $\mathbf{q_p}$ el vector velocidad, $\boldsymbol{\phi_{qp}}$ la derivada del jacobiano respecto al tiempo y $\boldsymbol{\phi_t}$ la derivada de las ecuaciones de restricción con respecto al tiempo, cuyo valor es nulo en la mayoría de los casos. Es decir, se tendría:

$$\varphi_q q_{pp} = -\varphi_{qp} q_p \qquad (3.20)$$

Del mismo modo que en el problema velocidad, llamando **b** al conjunto formado por $\boldsymbol{\phi_{qp}}\mathbf{q_{pp}}$ se llega a un sistema de ecuaciones lineal matricial:

$$Ax = b \qquad (3.21)$$

y despejando la **x**:

$$x = A^{-1} b \qquad (3.22)$$

Capítulo 4:

Notebooks

4. Notebooks

En este capítulo se describirán los Notebooks desarrollados en este TFG. Para cada uno de los cuatro mecanismos se han realizado tres Notebooks. En el primero se resuelve el problema posición y se dibuja el mecanismo en la posición **q**. En el segundo se deducen velocidad y aceleración ($\mathbf{q_p}$ y $\mathbf{q_{pp}}$) y se representan las gráficas de las velocidades y aceleraciones de cada coordenada de cada punto móvil. Por último, en el tercero se realiza una animación. En la tabla inferior se indica el número de Notebooks y qué se desarrolla en cada uno.

Para la explicación se insertarán algunas partes de los Notebooks, que se encontrarán completos en los anexos finales y pueden encontrarse en el repositorio de github https://github.com/ingmec-ual/jupyter-teoria-mecanismos.

En los anexos finales las celdas de texto aparecen en código LaTeX, ya que es la manera en la que se han escrito las ecuaciones y matrices con el fin de obtener en el Notebook un resultado con buena apariencia.

Notebook	Mecanismo	Problema Posición	Problemas Velocidad y Aceleración	Animación
1	Cuatro Barras	X		
2			X	
3				X
1	Cinco Barras	X		
2			X	
3				X
1	Biela-Manivela	X		
2			X	
3				X
1	Biela-Manivela Invertida	X		
2			X	
3				X

Tabla 1: Contenido de cada Notebook

4.1 Cuatro Barras

El mecanismo Cuatro Barras (4B) está formado por tres barras móviles y una fija, que se unen con nudos articulados. Las barras móviles se unen a la fija mediante pivotes.

Ejemplos reales de este tipo de mecanismo serían por ejemplo:

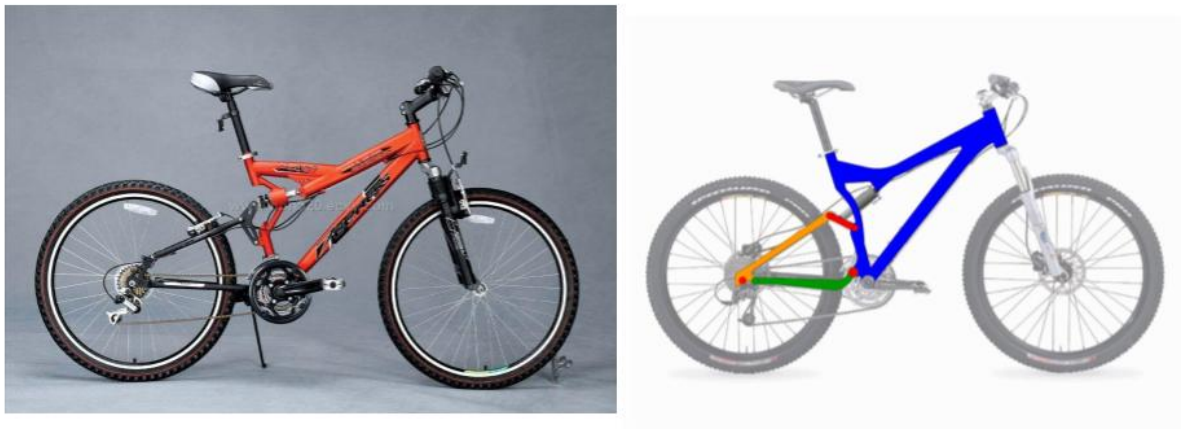


Figura 48: Suspensión trasera de una MTB (Fuente: [25])

Figura 49: Elíptica (Fuente: [25])

Figura 50: Limpiaparabrisas (Fuente: [25])

4.1.1 Problema Posición

En primer lugar se modeló el mecanismo para que el alumno tuviese una idea de su estructura, así como de la posición de cada punto. Esto se realizó en una celda de texto.

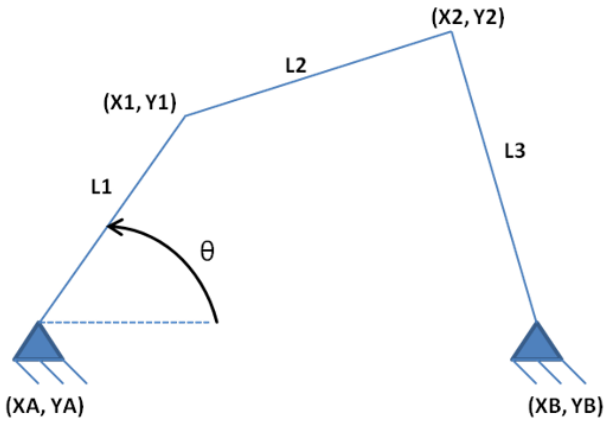


Figura 51: Modelado mecanismo Cuatro Barras

El siguiente paso es calcular los grados de libertad aplicando la ecuación explicada en el apartado de Teoría de Mecanismos. En este caso se tendrían cuatro barras y cuatro pares binarios de un grado de libertad. Resolviendo la ecuación, se obtendría que el mecanismo tiene un solo grado de libertad.

PASO 2: GRADOS DE LIBERTAD

Los grados de libertad se calculan:

$$G = 3 \cdot (n - 1) - 2 \cdot PI - PII$$

Siendo:

PI -> Numero de pares binarios de un grado de libertad

PII -> Número de pares binarios de dos grados de libertad. En este caso tendríamos:

$$n = 4$$
$$P_I = 4$$
$$P_{II} = 0$$

Por lo tanto:

$$G = 3 \cdot (4 - 1) - 2 \cdot 4 - 0 = 1$$

Figura 52: Grados de libertad mecanismo Cuatro Barras

El tercer paso sería definir el vector de variables dependientes **q**. En él se incluyen las coordenadas que varían con el tiempo y una correspondiente al grado de libertad. Para el caso del mecanismo cuatro barras estaría compuesto por cinco coordenadas: $\mathbf{X_1}$, $\mathbf{Y_1}$, $\mathbf{X_2}$, $\mathbf{Y_2}$ y $\boldsymbol{\theta}$.

PASO 3: DEFINICIÓN DEL VECTOR q

El vector $\mathbf{q}$ de dimensión $N \times 1$ contiene las coordenadas dependientes del mecanismo. Es decir, contiene las coordenadas que no se mantienen fijas ya que varían con el tiempo.

Lo hemos modelado empleando las 5 coordenadas:

$$\mathbf{q} = \begin{bmatrix} x_1 \\ y_1 \\ x_2 \\ y_2 \\ \theta \end{bmatrix}$$

Figura 53: Vector q mecanismo Cuatro Barras

Seguidamente, se ha añadido una celda de texto para explicar las primeras líneas del código, ya que en ellas se llaman una serie de librerías necesarias para la implementación del programa. Tras ella, se inserta la primera celda de código del programa, donde se importan las librerías mencionadas.

PASO 4: IMPLEMENTACIÓN EN PYTHON

Al igual que en otros entornos de programación, necesitamos añadir librerías que contengan las funciones que vamos a utilizar. Esto es necesario hacerlo al principio del código. Las que vamos a usar son las siguientes:

1. numpy -> Sirve para trabajar con arrays y matrices, ofreciendo una interfaz similar a los comandos en MATLAB.
2. math -> La utilizaremos para usar funciones matemáticas.
3. pprint -> "pretty print", su función es ayudar a depurar el código.
4. matplotlib.pyplot -> Es necesaria para dibujar gráficas.

Figura 54: Explicación de las librerías que se van a utilizar

```
#PASO 4

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

El quinto paso es la lectura de datos. Primero se ha explicado qué parámetros son los que se requieren para comenzar la resolución del problema y se ha resaltado el hecho de hay que elegir una variable independiente entre las componentes del vector **q** dado que se tiene un grado de libertad.

En este caso se ha escogido el ángulo, lo que hace que también sea un dato de partida. De este modo, se tiene que los datos necesarios al inicio serán: **L_1**, **L_2**, **L_3**, **X_A**, **X_B**, **Y_A**, **Y_B** y **Θ.** Tras la celda explicativa se procede a la implementación del código.

En la celda donde se desarrolla el código correspondiente a este apartado, se definió un *dictionary* vacío. Esto sirve para tener un array vacío donde se irán añadiendo los datos que se lean a continuación. Una vez se han leído todos los datos, se define una posición **q** inicial de forma aleatoria, que servirá para realizar la primera iteración en el proceso de resolución del problema posición.

Se van a introducir los siguientes datos de prueba:

$$L_1 = 1$$

$$L_2 = 2$$

$$L_3 = 2.5$$

$$\theta(inicial) = 0$$

$$X_B = 3$$

El código para la lectura de datos y definición de la posición inicial es:

```
#PASO 5

print ('MECANISMO DE CUATRO BARRAS')
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["L3"] = float (input ('Introduce longitud L3:'))
meca["theta"] = float (input ('Introduce angulo inicial theta:')) #En
rad
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))
```

El mecanismo cuatro barras el del tipo cuadrilátero articulado, razón por la cual habrá que evaluar si se cumple la Ley de Grashoff. El sexto paso se dedica a esto, en primer lugar desarrollando la teoría para que el alumno comprenda el por qué de tenerlo en cuenta y seguidamente implementando el código.

La relevancia de comprobar que se cumpla radica en la animación que se realizará posteriormente, ya que si no se cumple la desigualdad, la simulación no se ejecutará correctamente. El objetivo es que cumpliéndolo o no, resuelva los problemas posición, velocidad y aceleración, pero que en caso de no cumplir aparezca un mensaje en el que se indique. En este último supuesto, también se ejecutará la simulación para que se vea gráficamente la consecuencia del incumplimiento.

En la celda de código donde se hace la comprobación se hacen dos funciones. Primero se ordenan las longitudes de mayor a menor y después se comprueba que se cumpla la ley. En caso de no cumplirse, aparecerá el texto: “No cumple la desigualdad de Grashoff, por lo que la simulación no se ejecutará correctamente”.

```
# PASO 6

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.")
```

En séptimo lugar se construye la matriz de restricciones. Al igual que se ha estado haciendo en los pasos anteriores, hay una celda explicativa antes de la celda de código. En este caso, se explica el procedimiento general a seguir para componer esta matriz, así como las restricciones que se aplicarían al caso en cuestión. Para el cuatro barras se necesitan tres ecuaciones de sólido rígido, que se corresponden con las tres barras que lo componen, y una cuarta que haga referencia al ángulo Θ.

La matriz de restricciones para este mecanismo sería:

1. $Si \quad \cos(\theta) < \frac{1}{\sqrt{2}}$

$$\Phi = \begin{bmatrix} (X_1 - X_A)^2 + (Y_1 - Y_A)^2 - L_1^2 \\ (X_2 - X_1)^2 + (Y_2 - Y_1)^2 - L_2^2 \\ (X_B - X_2)^2 + (Y_B - Y_2)^2 - L_3^2 \\ X_1 - L_1 \cos(\theta) \end{bmatrix}$$

2. $Si \quad \cos(\theta) > \frac{1}{\sqrt{2}}$

$$\Phi = \begin{bmatrix} (X_1 - X_A)^2 + (Y_1 - Y_A)^2 - L_1^2 \\ (X_2 - X_1)^2 + (Y_2 - Y_1)^2 - L_2^2 \\ (X_B - X_2)^2 + (Y_B - Y_2)^2 - L_3^2 \\ Y_1 - L_1 \sin(\theta) \end{bmatrix}$$

Figura 55: Matriz de restricciones mecanismo Cuatro Barras

El código con el cual se diseña la matriz de restricciones es:

```
#PASO 7

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] = (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
```

Se prosigue con la construcción de la matriz jacobiana. Como ya se explicó anteriormente, esta matriz se compone por las derivadas parciales de las ecuaciones de restricción con respecto al vector de coordenadas dependientes. Este procedimiento es lo que se explica en la primera celda correspondiente a este apartado, mientras que en la segunda se escribe el código.

1. $Si \quad \cos(\theta) < \frac{1}{\sqrt{2}}$

$$\mathbf{\Phi_q} = \begin{bmatrix} 2X_1 & 2Y_1 & 0 & 0 & 0 \\ -2(X_2 - X_1) & -2(Y_2 - Y_1) & 2(X_2 - X_1) & 2(Y_2 - Y_1) & 0 \\ 0 & 0 & -2(X_B - X_2) & 2Y_2 & 0 \\ 1 & 0 & 0 & 0 & L1\sin(\theta) \end{bmatrix}$$

1. $Si \quad \cos(\theta) > \frac{1}{\sqrt{2}}$

$$\mathbf{\Phi_q} = \begin{bmatrix} 2X_1 & 2Y_1 & 0 & 0 & 0 \\ -2(X_2 - X_1) & -2(Y_2 - Y_1) & 2(X_2 - X_1) & 2(Y_2 - Y_1) & 0 \\ 0 & 0 & -2(X_B - X_2) & 2Y_2 & 0 \\ 0 & 1 & 0 & 0 & -L1\cos(\theta) \end{bmatrix}$$

Figura 56: Matriz jacobiana mecanismo Cuatro Barras

El código para diseñar la matriz jacobiana es:

```
#PASO 8


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
```

Una vez se tienen las matrices de restricción y jacobiana, se procede a la resolución del problema posición. Se ha seguido el mismo proceder que para el resto del Notebook, es decir, primero una celda de texto explicativa y después una de código.

Para realizar el proceso iterativo por el que se resuelve el problema posición, se crea un bucle que se repetirá mientras que el error sea mayor que la tolerancia designada y no se llegue al número máximo de iteraciones permitido. Antes de entrar en el bucle, habrá que dar unos valores iniciales al error y a la tolerancia, siendo el error 10 y la tolerancia 1^{-10}. Además, habrá que inicializar el contador **i** a 0 y definir una matriz de ceros llamada **deltaQ** de dimensiones **nx1**, es decir, las mismas que el vector **q.** Se da el valor de la **q** supuesta inicialmente al vector **q** antes de entrar en el bucle, para que su valor vaya cambiando con cada iteración.

Dentro del bucle, el proceso a seguir es dar a cada coordenada el valor guardado en su correspondiente posición del vector **q**, para con ellas recalcular las matrices de restricción y jacobiana. Se escriben ambas matrices y se calcula el rango de la matriz jacobiana, manteniendo un margen de 1^{-5}.

Estas dos partes del bucle sirven para controlar la exactitud del proceso iterativo, ya que como se explicó anteriormente, los valores de la matriz de restricciones deben ir aproximándose a cero y el rango del jacobiano debe ser igual al número de filas del vector de coordenadas dependientes.

Para continuar se calcula el incremento de **q** con una función propia de la librería Numpy, que se sumará al vector **q** ya conocido. Se calcula el valor de error haciendo el módulo del vector $\boldsymbol{\Delta_q}$ y se incrementa el contador de iteraciones. Para terminar, se incrementa el número del contador y se escribe el número del error. Este proceso se repetirá hasta que no se cumpla una de las dos condiciones impuestas en el bucle. Además, si en el proceso iterativo se supera el máximo de iteraciones indicado el mecanismo no convergerá, es decir, no podrá alcanzar la posición establecida. Para este caso se ha añadido una excepción para que se imprima en pantalla el mensaje "No se puede alcanzar la posición".

En este paso aparece una ventana de resultados, donde se puede seguir el proceso iterativo que se ha diseñado, donde se indica el valor del vector **q**, de $\boldsymbol{\phi}$ y $\boldsymbol{\phi_q}$, el rango de la matriz, el error de la iteración en la que se encuentra y el número de iteraciones totales.

```
#PASO 9
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)
```

Los resultados que arroja la ventana de salida para los datos de prueba propuestos se desarrollan en las siguientes páginas.

```
q=
array([[0.1],
       [1. ],
       [1. ],
       [0.2],
       [0. ]])
Phi=
array([[ 0.01],
       [-2.55],
       [-2.21],
       [ 1.  ],
       [ 0.  ]])
jacob=
array([[ 0.2,  2. ,  0. ,  0. ,  0. ],
       [-1.8,  1.6,  1.8, -1.6,  0. ],
       [ 0. ,  0. , -4. ,  0.4,  0. ],
       [ 0. ,  1. ,  0. ,  0. , -1. ],
       [ 0. ,  0. ,  0. ,  0. ,  1. ]])
rango=5

error iter1=
19.19195767662633
```

```
q=
array([[ 10.05      ],
       [  0.        ],
       [ -1.17605634],
       [-16.03556338],
       [  0.        ]])
Phi=
array([[100.0025    ],
       [379.16363383],
       [268.32873946],
       [  0.        ],
       [  0.        ]])
jacob=
array([[ 20.1       ,   0.        ,   0.        ,   0.        ,
           0.        ],
       [ 22.45211268,  32.07112676, -22.45211268, -32.07112676,
           0.        ],
       [  0.        ,   0.        ,  -8.35211268, -32.07112676,
           0.        ],
       [  0.        ,   1.        ,   0.        ,   0.        ,
          -1.        ],
       [  0.        ,   0.        ,   0.        ,   0.        ,
           1.        ]])
rango=5

error iter2=
9.748195126631266


q=
array([[ 5.07475124e+00],
       [-9.28043661e-17],
       [-1.23775501e+00],
       [-7.65281912e+00],
       [ 0.00000000e+00]])
Phi=
array([[ 2.47531002e+01],
       [ 9.44133757e+01],
       [ 7.02742080e+01],
       [-9.28043661e-17],
       [ 0.00000000e+00]])
jacob=
array([[ 1.01495025e+01, -1.85608732e-16,  0.00000000e+00,
         0.00000000e+00,  0.00000000e+00],
       [ 1.26250125e+01,  1.53056382e+01, -1.26250125e+01,
        -1.53056382e+01,  0.00000000e+00],
       [ 0.00000000e+00,  0.00000000e+00, -8.47551002e+00,
        -1.53056382e+01,  0.00000000e+00],
       [ 0.00000000e+00,  1.00000000e+00,  0.00000000e+00,
         0.00000000e+00, -1.00000000e+00],
       [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         0.00000000e+00,  1.00000000e+00]])
rango=5

error iter3=
6.207811152264088
```

```
q=
array([[ 2.63590262e+00],
       [ 7.73393444e-16],
       [-2.84067650e+00],
       [-2.17380649e+00],
       [ 0.00000000e+00]])
Phi=
array([[5.94798262e+00],
       [3.07183536e+01],
       [3.25889367e+01],
       [7.73393444e-16],
       [0.00000000e+00]])
jacob=
array([[ 5.27180524e+00,  1.54678689e-15,  0.00000000e+00,
         0.00000000e+00,  0.00000000e+00],
       [ 1.09531582e+01,  4.34761299e+00, -1.09531582e+01,
        -4.34761299e+00,  0.00000000e+00],
       [ 0.00000000e+00,  0.00000000e+00, -1.16813530e+01,
        -4.34761299e+00,  0.00000000e+00],
       [ 0.00000000e+00,  1.00000000e+00,  0.00000000e+00,
         0.00000000e+00, -1.00000000e+00],
       [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         0.00000000e+00,  1.00000000e+00]])
rango=5

error iter4=
49.07572536662256

q=
array([[ 1.50763965e+00],
       [ 7.56463733e-15],
       [ 1.66989119e+01],
       [-4.71777877e+01],
       [ 0.00000000e+00]])
Phi=
array([[1.27297732e+00],
       [2.45251841e+03],
       [2.40715384e+03],
       [7.56463733e-15],
       [0.00000000e+00]])
jacob=
array([[ 3.01527930e+00,  1.51292747e-14,  0.00000000e+00,
         0.00000000e+00,  0.00000000e+00],
       [-3.03825445e+01,  9.43555755e+01,  3.03825445e+01,
        -9.43555755e+01,  0.00000000e+00],
       [ 0.00000000e+00,  0.00000000e+00,  2.73978238e+01,
        -9.43555755e+01,  0.00000000e+00],
       [ 0.00000000e+00,  1.00000000e+00,  0.00000000e+00,
         0.00000000e+00, -1.00000000e+00],
       [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         0.00000000e+00,  1.00000000e+00]])
rango=5

error iter5=
27.826714440264766
```

```
q=
array([[ 1.08546406e+00],
       [-1.54079305e-15],
       [-2.79749660e+00],
       [-2.73274009e+01],
       [ 0.00000000e+00]])
Phi=
array([[ 1.78232229e-01],
       [ 7.57864221e+02],
       [ 7.74147804e+02],
       [-1.54079305e-15],
       [ 0.00000000e+00]])
jacob=
array([[ 2.17092812e+00, -3.08158611e-15,  0.00000000e+00,
         0.00000000e+00,  0.00000000e+00],
       [ 7.76592133e+00,  5.46548017e+01, -7.76592133e+00,
        -5.46548017e+01,  0.00000000e+00],
       [ 0.00000000e+00,  0.00000000e+00, -1.15949932e+01,
        -5.46548017e+01,  0.00000000e+00],
       [ 0.00000000e+00,  1.00000000e+00,  0.00000000e+00,
         0.00000000e+00, -1.00000000e+00],
       [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         0.00000000e+00,  1.00000000e+00]])
rango=5

error iter6=
13.945738845678857

q=
array([[ 1.00336451e+00],
       [-3.78141889e-16],
       [ 1.62163225e+00],
       [-1.41006026e+01],
       [ 0.00000000e+00]])
Phi=
array([[ 6.74033666e-03],
       [ 1.95209249e+02],
       [ 1.94476892e+02],
       [-3.78141889e-16],
       [ 0.00000000e+00]])
jacob=
array([[ 2.00672902e+00, -7.56283778e-16,  0.00000000e+00,
         0.00000000e+00,  0.00000000e+00],
       [-1.23653548e+00,  2.82012052e+01,  1.23653548e+00,
        -2.82012052e+01,  0.00000000e+00],
       [ 0.00000000e+00,  0.00000000e+00, -2.75673550e+00,
        -2.82012052e+01,  0.00000000e+00],
       [ 0.00000000e+00,  1.00000000e+00,  0.00000000e+00,
         0.00000000e+00, -1.00000000e+00],
       [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         0.00000000e+00,  1.00000000e+00]])
rango=5

error iter7=
6.916538585815401
```

```
q=
array([[ 1.00000564e+00],
       [ 5.83419759e-16],
       [ 1.43719430e+00],
       [-7.18652441e+00],
       [ 0.00000000e+00]])
Phi=
array([[1.12819902e-05],
       [4.78372670e+01],
       [4.78384947e+01],
       [5.83419759e-16],
       [0.00000000e+00]])
jacob=
array([[ 2.00001128e+00,  1.16683952e-15,  0.00000000e+00,
         0.00000000e+00,  0.00000000e+00],
       [-8.74377323e-01,  1.43730488e+01,  8.74377323e-01,
        -1.43730488e+01,  0.00000000e+00],
       [ 0.00000000e+00,  0.00000000e+00, -3.12561140e+00,
        -1.43730488e+01,  0.00000000e+00],
       [ 0.00000000e+00,  1.00000000e+00,  0.00000000e+00,
         0.00000000e+00, -1.00000000e+00],
       [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         0.00000000e+00,  1.00000000e+00]])
rango=5

error iter8=
3.328280592449615

q=
array([[ 1.00000000e+00],
       [ 4.06484118e-16],
       [ 1.43750000e+00],
       [-3.85824383e+00],
       [ 0.00000000e+00]])
Phi=
array([[3.18207682e-11],
       [1.10774517e+01],
       [1.10774517e+01],
       [4.06484118e-16],
       [0.00000000e+00]])
jacob=
array([[ 2.00000000e+00,  8.12968236e-16,  0.00000000e+00,
         0.00000000e+00,  0.00000000e+00],
       [-8.75000002e-01,  7.71648766e+00,  8.75000002e-01,
        -7.71648766e+00,  0.00000000e+00],
       [ 0.00000000e+00,  0.00000000e+00, -3.12500000e+00,
        -7.71648766e+00,  0.00000000e+00],
       [ 0.00000000e+00,  1.00000000e+00,  0.00000000e+00,
         0.00000000e+00, -1.00000000e+00],
       [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         0.00000000e+00,  1.00000000e+00]])
rango=5

error iter9=
1.435556200143426
```

```
q=
array([[ 1.00000000e+00],
       [-1.82585357e-16],
       [ 1.43750000e+00],
       [-2.42268763e+00],
       [ 0.00000000e+00]])
Phi=
array([[ 0.00000000e+00],
       [ 2.06082160e+00],
       [ 2.06082160e+00],
       [-1.82585357e-16],
       [ 0.00000000e+00]])
jacob=
array([[ 2.00000000e+00, -3.65170714e-16,  0.00000000e+00,
         0.00000000e+00,  0.00000000e+00],
       [-8.75000000e-01,  4.84537526e+00,  8.75000000e-01,
        -4.84537526e+00,  0.00000000e+00],
       [ 0.00000000e+00,  0.00000000e+00, -3.12500000e+00,
        -4.84537526e+00,  0.00000000e+00],
       [ 0.00000000e+00,  1.00000000e+00,  0.00000000e+00,
         0.00000000e+00, -1.00000000e+00],
       [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         0.00000000e+00,  1.00000000e+00]])
rango=5

error iter10=
0.42531723405813104
```

```
q=
array([[ 1.0000000e+00],
       [ 3.7379877e-17],
       [ 1.4375000e+00],
       [-1.9973704e+00],
       [ 0.0000000e+00]])
Phi=
array([[0.0000000e+00],
       [1.8089475e-01],
       [1.8089475e-01],
       [3.7379877e-17],
       [0.0000000e+00]])
jacob=
array([[ 2.00000000e+00,  7.47597539e-17,  0.00000000e+00,
         0.00000000e+00,  0.00000000e+00],
       [-8.75000000e-01,  3.99474079e+00,  8.75000000e-01,
        -3.99474079e+00,  0.00000000e+00],
       [ 0.00000000e+00,  0.00000000e+00, -3.12500000e+00,
        -3.99474079e+00,  0.00000000e+00],
       [ 0.00000000e+00,  1.00000000e+00,  0.00000000e+00,
         0.00000000e+00, -1.00000000e+00],
       [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         0.00000000e+00,  1.00000000e+00]])
rango=5

error iter11=
0.04528322586829045
q=
array([[ 1.00000000e+00],
       [ 1.06994449e-17],
       [ 1.43750000e+00],
       [-1.95208717e+00],
       [ 0.00000000e+00]])
Phi=
array([[0.00000000e+00],
       [2.05057055e-03],
       [2.05057055e-03],
       [1.06994449e-17],
       [0.00000000e+00]])
jacob=
array([[ 2.00000000e+00,  2.13988898e-17,  0.00000000e+00,
         0.00000000e+00,  0.00000000e+00],
       [-8.75000000e-01,  3.90417434e+00,  8.75000000e-01,
        -3.90417434e+00,  0.00000000e+00],
       [ 0.00000000e+00,  0.00000000e+00, -3.12500000e+00,
        -3.90417434e+00,  0.00000000e+00],
       [ 0.00000000e+00,  1.00000000e+00,  0.00000000e+00,
         0.00000000e+00, -1.00000000e+00],
       [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         0.00000000e+00,  1.00000000e+00]])
rango=5

error iter12=
0.0005252251477831435
```

```
q=
array([[ 1.00000000e+00],
       [ 3.56374430e-20],
       [ 1.43750000e+00],
       [-1.95156195e+00],
       [ 0.00000000e+00]])
Phi=
array([[0.00000000e+00],
       [2.75861455e-07],
       [2.75861455e-07],
       [3.56374430e-20],
       [0.00000000e+00]])
jacob=
array([[ 2.00000000e+00,  7.12748859e-20,  0.00000000e+00,
         0.00000000e+00,  0.00000000e+00],
       [-8.75000000e-01,  3.90312389e+00,  8.75000000e-01,
        -3.90312389e+00,  0.00000000e+00],
       [ 0.00000000e+00,  0.00000000e+00, -3.12500000e+00,
        -3.90312389e+00,  0.00000000e+00],
       [ 0.00000000e+00,  1.00000000e+00,  0.00000000e+00,
         0.00000000e+00, -1.00000000e+00],
       [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         0.00000000e+00,  1.00000000e+00]])
rango=5

error iter13=
7.067709434954129e-08

q=
array([[ 1.00000000e+00],
       [ 1.24070869e-23],
       [ 1.43750000e+00],
       [-1.95156187e+00],
       [ 0.00000000e+00]])
Phi=
array([[0.00000000e+00],
       [5.32907052e-15],
       [5.32907052e-15],
       [1.24070869e-23],
       [0.00000000e+00]])
jacob=
array([[ 2.00000000e+00,  2.48141739e-23,  0.00000000e+00,
         0.00000000e+00,  0.00000000e+00],
       [-8.75000000e-01,  3.90312375e+00,  8.75000000e-01,
        -3.90312375e+00,  0.00000000e+00],
       [ 0.00000000e+00,  0.00000000e+00, -3.12500000e+00,
        -3.90312375e+00,  0.00000000e+00],
       [ 0.00000000e+00,  1.00000000e+00,  0.00000000e+00,
         0.00000000e+00, -1.00000000e+00],
       [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         0.00000000e+00,  1.00000000e+00]])
rango=5

error iter14=
1.3653347480295311e-15
num iters:14
```

Al analizar el resultado obtenido, se puede ver fácilmente que en las primeras iteraciones los valores de **ϕ** dista notoriamente de **0** y que el error es demasiado grande. Conforme van sucediéndose las iteraciones, la matriz se acerca cada vez más al valor nulo, del mismo modo que hace el error. Además, en todo momento el rango de la matriz jacobiana el rango es igual al número de coordenadas dependientes de **q**.

Para finalizar, en este Notebook se representa gráficamente la posición obtenida. Para ello se llama al problema posición para obtener el valor final de **q**, y seguidamente se extraen los puntos móviles del mecanismo. Tras esto, se utilizan funciones de la librería Matplotlib con las que se indica que los ejes X e Y deben ser iguales y se dibuja cada barra de manera individual. Además, con un fin estético, se ha representado con un punto grueso las articulaciones fijas. Por último, se utiliza la función de la librería mencionada que representa la gráfica definida.

```
#PASO 10


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')


    plt.plot ([meca["XA"], X1], [meca["YA"], Y1])     #[pos inicial
(x1,x2), pos final (y1,y2)]
    plt.plot ([X1, X2], [Y1, Y2])
    plt.plot ([X2, meca["XB"]], [Y2, meca ["YB"]])

    plt.plot(meca["XA"], meca["YA"], 'bo')
    plt.plot(meca["XB"], meca["YB"], 'go')

    plt.show()#block=False)
    return

dibuja_mecanismo(q,meca)
```

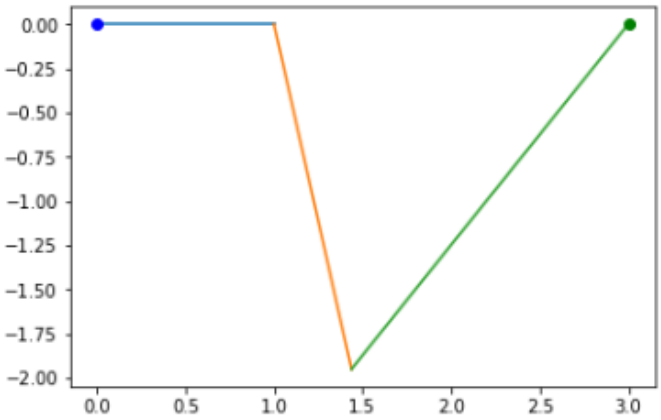


Figura 57: Dibujo mecanismo Cuatro Barras

4.1.2 Problemas Velocidad y Aceleración

En el segundo Notebook se resuelven los problemas velocidad y aceleración. Además, se representa en forma de gráfica cómo varían estos parámetros para cada coordenada del vector **q**.

Para comenzar, se inserta el código desarrollado en el Notebook anterior, a excepción de la función que dibuja el mecanismo. El objetivo es calcular la matriz jacobiana y el vector de posición.

Una vez calculados, se puede proceder a la resolución del problema velocidad. El proceso se ha explicado anteriormente y aparece detallado en un cuadro de texto, con la finalidad de que el alumno comprenda todos los pasos llevados a cabo. Para codificar este cálculo, es necesario crear una matriz de ceros correspondiente al vector velocidad para posteriormente calcular sus valores. Esto se hace dándole a la variable conocida un valor, que en este caso es la unidad, para seguidamente calcular el resto de parámetros del vector $\mathbf{q_p}$ a partir de él, gracias a la función *resuelve_prob_velocidad*. En ella se utiliza una función contenida en la librería Numpy con la cual se puede realizar el cálculo que se detalló en el apartado teórico.

```
#PASO 2

def resuelve_prob_velocidad(q,qp,meca):
    qp = np.linalg.solve(jacob_Phiq(q,meca),qp)
    return qp
qp = np.zeros ((5,1))
qp[4]=1

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

De esta forma se obtiene que el vector velocidad es:

```
qp=
array([[-9.12892351e-32],
       [ 1.00000000e+00],
       [-9.75780937e-01],
       [ 7.81250000e-01],
       [ 1.00000000e+00]])
```

A continuación se resuelve el problema aceleración. En la celda de código se ha implementado el procedimiento visto anteriormente, según el cual para obtener el vector aceleración es necesario obtener el producto de $\partial_{\Phi q} \cdot q_p$, al cual se ha llamado **b**, para posteriormente multiplicar por el lado izquierdo por la inversa del jacobiano.

Se creará una matriz compuesta por ceros para el vector aceleración q_{pp} del mismo modo que se hizo para el problema velocidad, donde posteriormente se irán añadiendo valores. Tras este inicio, se llama a la función *resuelve_prob_posicion,* en la cual se extraen las variables de posiciones y velocidades y se realiza el cálculo del producto de la derivada del jacobiano con respecto al tiempo y el vector velocidad.

Para terminar, se volverá a hacer uso de la función *linalg.solve* para realizar la operación final. Con ello se tendrá el vector aceleración.

1. $Si \ \cos(\theta) < \frac{1}{\sqrt{2}}$

$$\mathbf{b} = \begin{bmatrix} 2\dot{X}_1^2 + 2\dot{Y}_1^2 \\ -2\dot{X}_1(\dot{X}_2 - \dot{X}_1) - 2\dot{Y}_1(\dot{Y}_2 - \dot{Y}_2) + 2\dot{X}_2(\dot{X}_2 - \dot{X}_1) - 2\dot{Y}_2 \cdot (\dot{Y}_2 - \dot{Y}_1) \\ 2\dot{X}_2^2 + 2\dot{Y}_2^2 \\ \dot{\theta}^2 \cdot L_1 \cdot \cos(\theta) \\ 1 \end{bmatrix}$$

1. $Si \ \cos(\theta) > \frac{1}{\sqrt{2}}$

$$\mathbf{b} = \begin{bmatrix} 2\dot{X}_1^2 + 2\dot{Y}_1^2 \\ -2\dot{X}_1(\dot{X}_2 - \dot{X}_1) - 2\dot{Y}_1(\dot{Y}_2 - \dot{Y}_2) + 2\dot{X}_2(\dot{X}_2 - \dot{X}_1) - 2\dot{Y}_2(\dot{Y}_2 - \dot{Y}_1) \\ 2\dot{X}_2^2 + 2\dot{Y}_2^2 \\ \dot{\theta}^2 L_1 \sin(\theta) \\ 1 \end{bmatrix}$$

Figura 58: Matriz b mecanismo Cuatro Barras

```
#PASO 3
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))

    b[4] = 1 #Aceleracion conocida
    qpp = np.linalg.solve(-jacob_Phiq(q,meca),b)
    #print ("qpp=")
    #pprint.pprint(qpp)

    return qpp

qpp=np.zeros((5,1))
qpp=resuelve_prob_aceleracion(q,qp,qpp,meca)
qpp
```

Teniendo como resultado:

```
qpp=
array([[-1. ],
       [-1. ],
       [ 1.03828094],
       [-0.03064928],
       [-1. ]])
```

Una vez calculadas las velocidades y aceleraciones del mecanismo, se pueden graficar para cada coordenada. Estas gráficas se han hecho para el caso de una revolución completa, definiendo un array para el ángulo que vaya de **0** a **2π** revoluciones dando 50 valores.

Se crea una matriz de ceros para cada coordenada que contenga 50 posiciones, a la cual se irán añadiendo valores conforme vaya aumentando el ángulo. Se inicia el contador a cero y a continuación, se crea un bucle *for* que vaya incrementando el contador de uno en uno hasta llegar al máximo indicado en el array definido inicialmente.

Dentro del bucle, se asigna a la última variable del vector posición, es decir, la correspondiente al ángulo, el valor del ángulo **t**, que irá aumentando conforme se vayan sucediendo las iteraciones. Conociendo ese valor, se obtienen el resto de valores para el problema posición. Una vez resuelto, se crea una matriz de ceros para el vector velocidad $\mathbf{q_p}$ y se le asigna un valor de velocidad a la componente correspondiente al dato de velocidad conocido. Teniendo esa componente del vector velocidad, se llama al problema velocidad para obtener el resto de elementos.

Cuando se han obtenido todas las componentes del vector $\mathbf{q_p}$, se añaden esos valores a las matrices de ceros creadas inicialmente para guardar la velocidad de cada coordenada. Tras ello se incrementa el contador.

Una vez finaliza el bucle, se representan las cuatro gráficas utilizando funciones de la librería Matplotlib.

El código y los resultados obtenidos para el ejemplo propuesto son los siguientes:

```
#PASO 4: GRÁFICAS DE VELOCIDADES
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))
        #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)
```

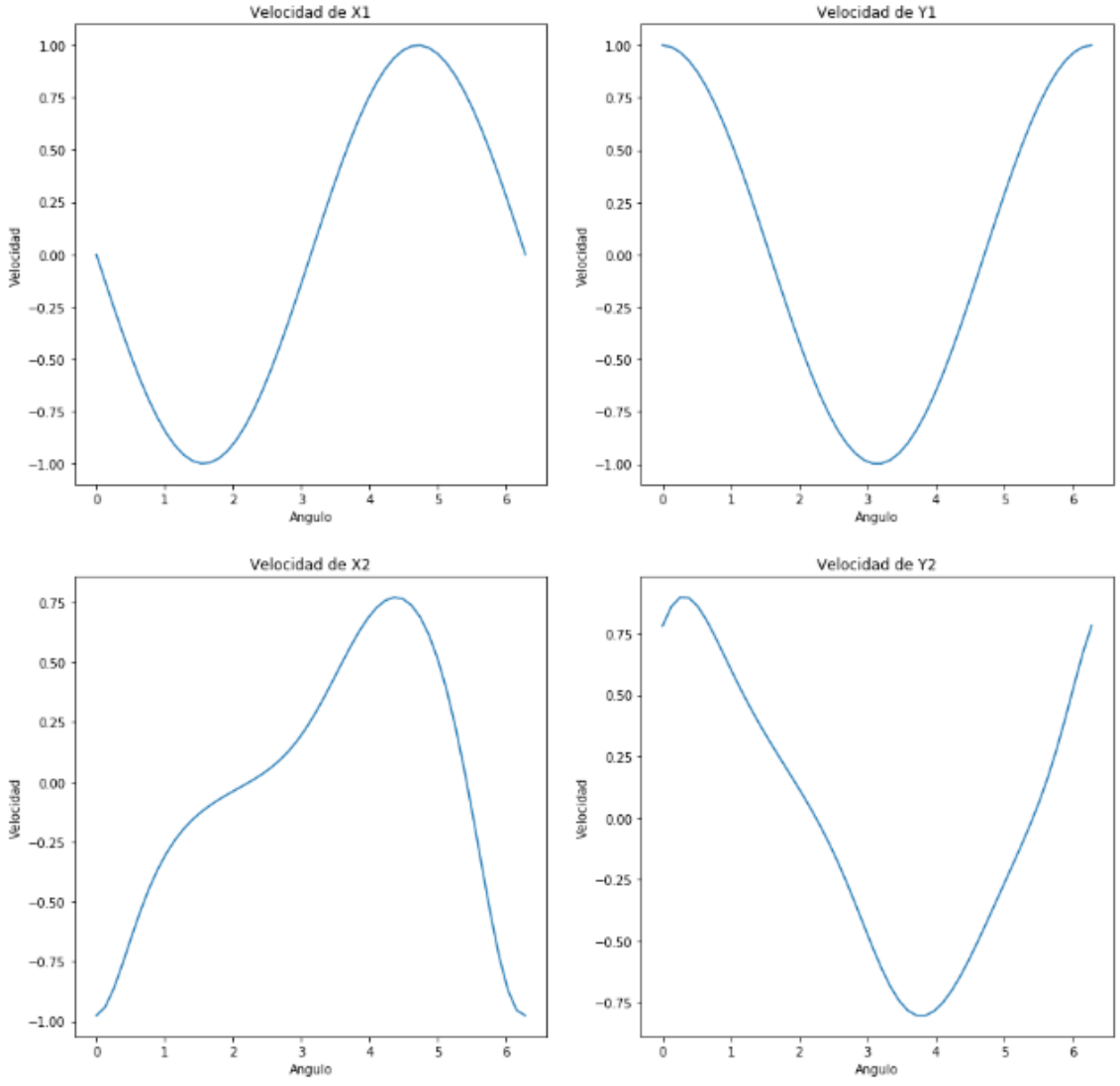


Figura 59: Gráficas velocidades mecanismo Cuatro Barras

Para graficar las aceleraciones se ha realizado un proceso muy similar al de las velocidades. La principal modificación es que una vez calculados todos los valores del vector velocidad, se crea una matriz de ceros para el vector aceleración $\mathbf{q_{pp}}$. Seguidamente se le da un valor a la componente correspondiente al dato de aceleración conocido y a partir de él se obtienen el resto de valores llamando al problema aceleración. El código y los resultados son:

```
#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))



    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 ('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)
```

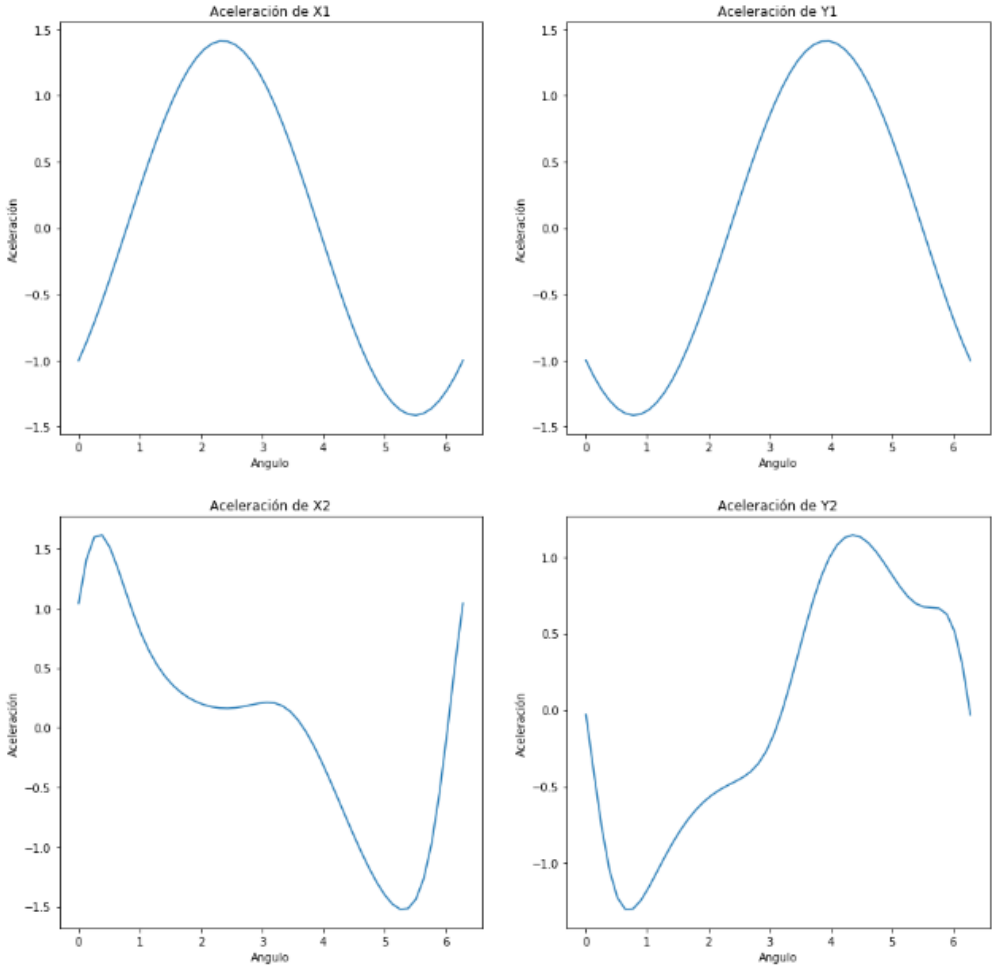


Figura 60: Gráficas aceleraciones mecanismo Cuatro Barras

4.1.3 Animación

Para hacer la animación, en primer lugar hay que volver a copiar el código donde se resuelve el problema posición. Tras esto, se puede proceder a hacer el código que creará la simulación. Para ello es necesario importar la función *animation* de la librería Matplotlib y la función *HTML* de *Ipython.Display*.

Tras la importación de librerías, se definen los límites de los ejes y se guarda el valor calculado de **q** con el nombre **last_q.** Una vez terminados los pasos preliminares, se define una función inicial, con la cual se inicializa la animación, para a continuación crear la función *animate*, que será la que genere la animación propiamente dicha.

En esta última función se define la variable **last_q** como una variable global para que la reconozca, ya que fue mencionada antes de la función. La función que se va a utilizar realmente es un bucle que va recalculando el valor de la posición y lo representa en la

misma pantalla. Por lo tanto, se le asigna el valor **last_q** a **q** para que la **q** inicial sea la última calculada.

Una vez calculado este valor, se resuelve el problema posición para obtener el resto de elementos que conforman el vector **q.** Una vez modificados los valores, se le asigna el valor **q** a la variable **last_q**, para que pase a tener el nuevo valor. A continuación, se extraen las coordenadas y se definen las coordenadas de los ejes **x** e **y**.

Por último se llama a la función *FuncAnimation*, a la cual se le pasarán una serie de parámetros. El primero será *fig,* que se creó anteriormente. Después va la función que realiza la animación, a la cual se le ha dado el nombre de *animate*. A continuación, se indica dónde se inicia la función, que en este caso sería en *init*. Seguidamente, en *fargs* se indican los argumentos restantes que se pasan a la función *anima,* siendo en este caso *q* y *meca*. Tras esta función se le pasa el número de *frames* que se va a usar, es decir, el parámetro *i* que se le pasa a la función *animate.* Después aparece el parámetro *interval*, que indica el intervalo de *frames* en milisegundo. Por último se ha puesto *blit=True*, que indica que la imagen no se recalcula entera sino que solo se actualizan las partes nuevas obtenidas en cada nuevo *frame*, lo cual permite que la animación se ejecute a mayor velocidad.

```
%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())
```

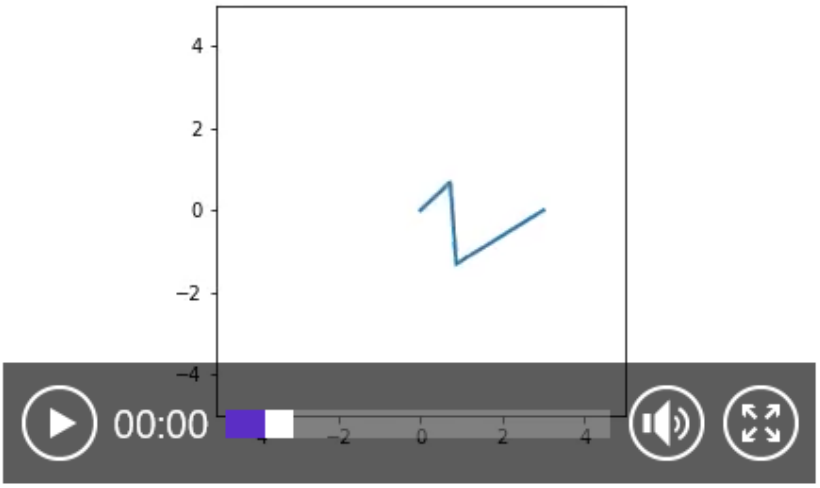


Figura 61: Animación mecanismo Cuatro Barras

4.2 Cinco Barras

El mecanismo Cinco Barras (5B) se compone por cuatro barras articuladas y una fija. Ejemplos reales de este mecanismo son los siguientes:

Figura 62: Robot SCARA de doble brazo (Fuente: 25])

4.2.1 Problema Posición

El Notebook correspondiente al problema de posición del mecanismo cinco barras sigue la misma estructura que el del cuatro barras. Por lo tanto, también se comienza modelando el mecanismo, como se representa en la figura siguiente.

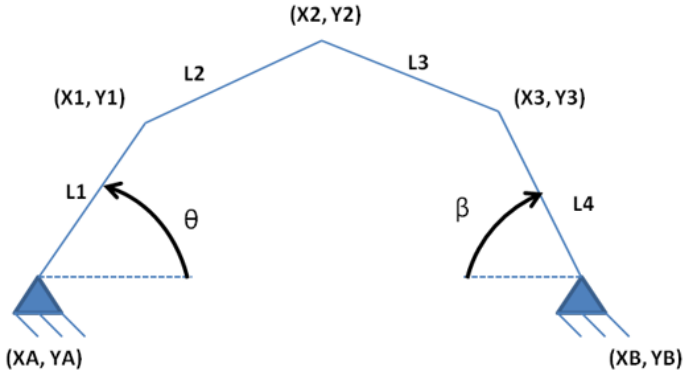


Figura 63: Modelado mecanismo Cinco Barras

A continuación es necesario calcular los grados de libertad y definir el vector **q** *[Figura 66].* Para este mecanismo habrá que añadir las coordenadas del tercer punto móvil y además, al tener dos grados de libertad se tendrá que escoger también como coordenada dependiente el ángulo que forma la cuarta barra con la fija.

PASO 2: GRADOS DE LIBERTAD

Los grados de libertad se calculan:

$$G = 3 \cdot (n - 1) - 2 \cdot PI - PII$$

Siendo:

PI -> Numero de pares binarios de un grado de libertad

PII -> Número de pares binarios de dos grados de libertad. En este caso tendríamos:

$$n = 5$$
$$P_I = 5$$
$$P_{II} = 0$$

Por lo tanto:

$$G = 3 \cdot (5 - 1) - 2 \cdot 5 - 0 = 2$$

PASO 3: DEFINICIÓN DEL VECTOR q

El vector $\mathbf{q}$ de dimensión $N \times 1$ contiene las coordenadas dependientes del mecanismo. Es decir, contiene las coordenadas que no se mantienen fijas ya que varían con el tiempo.

Lo hemos modelado enpleando las 8 coordenadas:

$$\mathbf{q} = \begin{bmatrix} x_1 \\ y_1 \\ x_2 \\ y_2 \\ x_3 \\ y_3 \\ \theta \\ \beta \end{bmatrix}$$

Figura 64: GDL y vector q mecanismo Cinco Barras

Se continúa con la lectura de datos. Serán los mismos que se necesitaban para el Cuatro Barras, aunque añadiendo la longitud de la cuarta barra y el segundo ángulo. Antes de este paso también se ha explicado la necesidad de incorporar las librerías y se ha hecho la llamada.

No obstante, no se adjuntará la imagen en este apartado porque es exactamente igual que en el mecanismo anterior, del mismo modo que ocurrirá para los dos siguientes mecanismos.

Los datos de prueba utilizados han sido:

$$L_1 = 2$$

$$L_2 = 3.5$$

$$L_3 = 4$$

$$L_4 = 1$$

$$\theta = 0.3$$

$$\beta = 0.9$$

$$X_B = 2$$

Una vez leídos los datos, se procede a construir la matriz de restricciones. La celda de texto explicativa es idéntica a la que aparecía en el mecanismo Cuatro Barras, ya que el tipo de restricciones a emplear son las mismas. Solo varía el final, donde aparece la matriz construida.

Para nuestro caso tendríamos:

$$Si\ \cos(\theta) < \frac{1}{\sqrt{2}} \rightarrow\ \Phi(3) = (X_1 - X_A) - L_1 \cdot \cos(\theta)$$

$$Si\ \cos(\theta) > \frac{1}{\sqrt{2}} \rightarrow\ \Phi(3) = (Y_1 - Y_A) - L_1 \cdot \sin(\theta)$$

$$Si\ \cos(\beta) < \frac{1}{\sqrt{2}} \rightarrow\ \Phi(4) = (X_3 - X_B) - L_4 \cdot \cos(\beta)$$

$$Si\ \cos(\beta) > \frac{1}{\sqrt{2}} \rightarrow\ \Phi(4) = (Y_3 - Y_B) - L_4 \cdot \sin(\beta)$$

La matriz quedaría:

$$\mathbf{\Phi} = \begin{bmatrix} (X_1 - X_A)^2 + (Y_1 - Y_A)^2 - L_1^2 \\ (X_2 - X_1)^2 + (Y_2 - Y_1)^2 - L_2^2 \\ (X_3 - X_2)^2 + (Y_3 - Y_2)^2 - L_3^2 \\ (X_B - X_3)^2 + (Y_B - Y_3)^2 - L_4^2 \\ \Phi(3) \\ \Phi(4) \end{bmatrix}$$

Figura 65: Matriz de restricciones mecanismo Cinco Barras

```
# PASO 6


def Phi (q,meca):
    #Inicializa a cero Phi
    Phi = np.zeros((8,1))

    #Extraer coordenadas
    X1 = q[0]
    Y1 = q[1]
    X2 = q[2]
    Y2 = q[3]
    X3 = q[4]
    Y3 = q[5]
    theta = q[6]
    beta = q[7]

    Phi[0] = X1**2 + Y1**2 - meca["L1"]**2
    Phi[1] = (X2-X1)**2 + (Y2-Y1)**2 -meca["L2"]**2
    Phi[2] = (X3-X2)**2 + (Y3-Y2)**2 - meca["L3"]**2
    Phi[3] = (meca["XB"] - X3)**2 + (meca["YB"] - Y3)**2 - meca["L4"]**2

    if (abs(math.cos(theta)) < 0.05 ):
```

```
        Phi[4] = X1-meca["L1"]*math.cos(theta)
    else:
        Phi[4] = Y1-meca["L1"]*math.sin(theta)

    if (abs(math.cos(beta)) < 0.05 ):
        Phi[5] = (X3-meca["XB"])-meca["L4"]*math.cos(beta)
    else:
        Phi[5] = Y3-meca["L4"]*math.sin(beta)

    return Phi
```

El siguiente paso es la construcción de la matriz jacobiana.

1. $Si \ \cos(\theta) < \frac{1}{\sqrt{2}}$

   $\Phi_q(4,0) = 1 // \ \Phi_q(4,6) = L_1 \cdot \sin(\theta)$

2. $Si \ \cos(\theta) > \frac{1}{\sqrt{2}}$

   $\Phi_q(4,1) = 1 // \ \Phi_q(4,6) = -L_1 \cdot \cos(\theta)$

1. $Si \ \cos(\beta) < \frac{1}{\sqrt{2}}$

   $\Phi_q(5,4) = 1 // \ \Phi_q(5,7) = L_4 \cdot \sin(\beta)$

2. $Si \ \cos(\beta) > \frac{1}{\sqrt{2}}$

   $\Phi_q(5,5) = 1 // \ \Phi_q(5,7) = -L_4 \cdot \cos(\beta)$

La matriz jacobiana sería:

$$\boldsymbol{\Phi_q} = \begin{bmatrix} 2X_1 & 2Y_1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -2(X_2 - X_1) & -2(Y_2 - Y_1) & 2(X_2 - X_1) & 2(Y_2 - Y_1) & 0 & 0 & 0 & 0 \\ 0 & 0 & -2(X_3 - X_2) & -2(Y_3 - Y_2) & 2(X_3 - X_2) & 2(Y_3 - Y_2) & 0 & 0 \\ 0 & 0 & 0 & 0 & -2(X_B - X_3) & -2(Y_B - Y_3) & 0 & 0 \\ \Phi_q(4,0) & \Phi_q(4,1) & 0 & 0 & 0 & 0 & \Phi_q(4,6) & \Phi_q(4,7) \\ 0 & 0 & 0 & 0 & \Phi_q(5,4) & \Phi_q(5,5) & \Phi_q(5,6) & \Phi_q(5,7) \end{bmatrix}$$

Figura 66: Matriz jacobiana mecanismo Cinco Barras

```
def jacob_Phiq(q,meca):
    #Inicializa a cero la matriz jacobiana
    Jacob = np.zeros((8,8))

    #Extraer coordenadas

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

```
    theta = q[6]
    beta = q[7]

    #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*(X3-X2)
    Jacob[2,3] = -2*(Y3-Y2)
    Jacob[2,4] = 2*(X3-X2)
    Jacob[2,5] = 2*(Y3-Y2)
    Jacob[3,4] = -2*(meca["XB"]-X3)
    Jacob[3,5] = -2*(meca["YB"]-Y3)

    if (abs(math.cos(theta)) < 0.05 ):
        Jacob[4,6] = meca["L1"]*math.sin(theta)
        Jacob[4,0] = 1
    else:
        Jacob[4,6] = -meca["L1"]*math.cos(theta)
        Jacob[4,1] = 1
    if (abs(math.cos(beta)) < 0.05 ):
        Jacob[5,7] = meca["L4"]*math.sin(beta)
        Jacob[5,4] = 1
    else:
        Jacob[5,7] = -meca["L4"]*math.cos(beta)
        Jacob[5,5] = 1

    Jacob[6,6] = 1
    Jacob[7,7] = 1
    return Jacob
```

Con todo esto ya se puede proceder a resolver el problema de posición. El método a seguir será el mismo para todos los mecanismos. Para el que se está tratando concretamente, habrá que tener en cuenta que al tener dos grados de libertad y haber escogido los ángulos, en la matriz de restricciones extendida habrá que asignarle el valor cero a sus parámetros correspondientes.

En la matriz jacobiana extendida las dos filas a añadir serán las siguientes:

$$\varphi_q(6) = (0, 0, 0, 0, 0, 0, 1, 0)$$

$$\varphi_q(7) = (0, 0, 0, 0, 0, 0, 0, 1)$$

Que sean los términos seis y siete pese a tener ocho variables dependientes se debe simplemente a cuestiones de nomenclatura, ya que en el lenguaje Python se nombra los elementos comenzando por el número cero. Esto quiere decir que el término seis será realmente el siete y el elemento siete será el ocho.

```
# PASO 8


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 ((8,1))
    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]
        X3 = q[4]
        Y3 = q[5]
        theta = q[6]
        beta = q[7]

        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)
```

Dado que en el mecanismo anterior aparecen los resultados de todas las iteraciones para ilustrar el resultado que aporta el proceso iterativo, en el Cinco Barras solo se va a indicar la **q** final obtenida con el fin de reducir la extensión del apartado. El Notebook completo ejecutado se encontrará en los anexos finales para poder ver en detalle el proceso. Por lo tanto, el vector posición será:

```
q=
array([[ 1.91067298],
       [ 0.59104041],
       [-0.9068351 ],
       [ 2.66749129],
       [ 2.62160997],
       [ 0.78332691],
       [ 0.3 ],
       [ 0.9 ]])
```

Por último se hace el dibujo del mecanismo para la posición calculada.

```
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]
    beta = q[7]

    plt.axis('equal')

    plt.plot ([meca["XA"], X1], [meca["YA"], Y1])     #[pos inicial (x1,x2), pos
final (y1,y2)]
    plt.plot ([X1, X2], [Y1, Y2])
    plt.plot ([X2, X3], [Y2, Y3])
    plt.plot([X3,meca["XB"]], [Y3, meca["YB"]])

    plt.plot(meca["XA"], meca["YA"], 'bo')
    plt.plot(meca["XB"], meca["YB"], 'go')
    plt.show()#block=False)
    return

dibuja_mecanismo(q,meca)
```

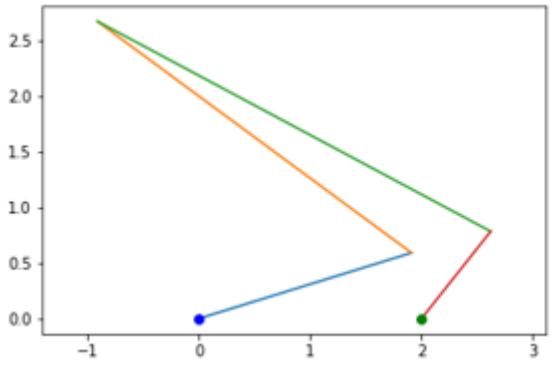


Figura 67: Dibujo mecanismo Cinco Barras

4.2.2 Problemas Velocidad y Aceleración

El proceso para calcular la velocidad es exactamente igual que en el mecanismo Cuatro Barras, permaneciendo completamente invariables las celdas de texto explicativas. Lo único que varía en este punto es la celda de código, ya que al tener dos grados de libertad hay que dar como dato las velocidades y aceleraciones correspondientes a los dos ángulos conocidos.

```
# PASO 2

def resuelve_prob_velocidad(q,qp, meca):

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

    return qp


qp = np.zeros ((8,1))
    #Velocidad del gdl.
qp[6]=0.1
qp[7] =0
qp=resuelve_prob_velocidad (q,qp, meca)

qp
```

Como resultado se tiene que para los datos de prueba el vector velocidad es:

```
qp=

array([[-3.82134596],
       [ 0.1910673  ],
       [ 0.21667426 ],
       [ 2.2100774  ],
       [ 0.         ],
       [ 0.         ],
       [ 0.1        ],
       [ 0.         ]])
```

Para el problema aceleración se debe considerar que habrá diferentes posibles valores para la derivada del jacobiano respecto al tiempo, ya que cada ecuación correspondiente a uno de los dos ángulos puede tomar dos valores en función de su amplitud.

Por otro lado, dado que la matriz **b** se obtiene necesariamente con la derivada del jacobiano, también tendrá diferentes configuraciones posibles.

$$\mathbf{b} = \begin{bmatrix} 2\dot{X}_1^2 & 2\dot{Y}_1^2 & 0 & 0 & 0 & 0 & 0 & 0 \\ -2\dot{X}_1(\dot{X}_2 - \dot{X}_1) & -2\dot{Y}_1(\dot{Y}_2 - \dot{Y}_1) & 2\dot{X}_2(\dot{X}_2 - \dot{X}_1) & 2\dot{Y}_2(\dot{Y}_2 - \dot{Y}_1) & 0 & 0 & 0 & 0 \\ 0 & 0 & -2\dot{X}_2(\dot{X}_3 - \dot{X}_2) & -2\dot{Y}_2(\dot{Y}_3 - \dot{Y}_2) & 2\dot{X}_3(\dot{X}_3 - \dot{X}_2) & 2\dot{Y}_3(\dot{Y}_3 - \dot{Y}_2) & 0 & 0 \\ 0 & 0 & 0 & 0 & 2\dot{X}_3^2 & 2\dot{Y}_3^2 & 0 & 0 \\ b(4) & & & & & & & \\ b(5) & & & & & & & \\ 1 & & & & & & & \\ 1 & & & & & & & \end{bmatrix}$$

1. $Si \;\; \cos(\theta) < 0.95 \rightarrow b(4) = \dot{\theta}^2 \cdot L_1 \cdot \cos(\theta)$
2. $Si \;\; \cos(\theta) > 0.95 \rightarrow b(4) = \dot{\theta}^2 \cdot L_1 \cdot \sin(\theta)$
3. $Si \;\; \cos(\beta) < 0.95 \rightarrow b(5) = \dot{\beta}^2 \cdot L_4 \cdot \cos(\beta)$
4. $Si \;\; \cos(\beta) > 0.95 \rightarrow b(5) = \dot{\beta}^2 \cdot L_4 \cdot \sin(\beta)$

Figura 68:Matriz b mecanismo Cinco Barras

```
#PASO 3


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]
    beta = q[7]
    #Extraer las velocidades
    X1q = qp[0]
    Y1q = qp[1]
    X2q = qp[2]
    Y2q = qp[3]
    X3q = qp[4]
    Y3q = qp[5]
    thetaq = qp[6]
    betaq = qp[7]
    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*(X3q-X2q) - 2*Y2q*(Y3q-Y2q) + 2*X3q*(X3q-X2q) + 2*Y3q*(Y3q-Y2q)
    b[3] = 2*X3q**2 + 2*Y3q**2
```

```
    if (abs(math.cos(theta)) < (math.sqrt(2)/2) ):
        b[4] = thetaq**2 * (meca["L1"] * math.cos(theta))
    else:
        b[4] =  thetaq**2 * (meca["L1"] * math.sin(theta))

    if (abs(math.cos(beta)) < (math.sqrt(2)/2) ):
        b[5] = betaq**2 * (meca["L4"] * math.cos(beta))
    else:
        b[5] =  betaq**2 * (meca["L4"] * math.sin(beta))

    b[6] = 1 #Aceleracion conocida
    b[7] = 1
    qpp = np.linalg.solve(-jacob_Phiq(q,meca),b)



    return qpp

qpp=np.zeros ((8,1))
qpp = resuelve_prob_aceleracion(q,qp, qpp,meca)
qpp
```

El vector aceleración obtenido es:

```
qpp=
array([[-1.84487168e+02],
       [ 1.90476257e+00],
       [ 1.51174210e+01],
       [ 1.13030395e+02],
       [-7.83326910e-01],
       [ 1.56665382e-01],
       [ 1.00000000e+00],
       [ 1.00000000e+00]])
```

Para graficar las velocidades, el código utilizado es muy similar al del mecanismo Cuatro Barras. Hay que tener en cuenta que al tener dos grados de libertad en lugar de uno, se le debe pasar al problema posición un valor para cada uno de ellas, del mismo modo que habrá que hacer para el problema velocidad.

Además, como es evidente, se obtendrán como resultado seis gráficas en lugar de cuatro, ya que son 3 puntos móviles, por lo que habrá que añadirlas en el *subplot*. Esta misma consideración debe hacerse para las gráficas de las aceleraciones. Los códigos se desarrollan en las siguientes páginas.

```
#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[7] = t
        q = resuelve_prob_posicion (q,meca)

        qp = np.zeros((8,1))
        qp[6] = -1  #inicializar qp en 0 con qp[6] = 1 rad/s
        qp[7] = -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])
        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)
```

```
#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[7] = t
        q = resuelve_prob_posicion (q,meca)

        qp = np.zeros((8,1))
        qp[6] = 1  #inicializar qp en 0 con qp[6] = 1 rad/s
        qp[7] = 2
        qp = resuelve_prob_velocidad(q, qp, meca)

        qpp = np.zeros((8,1))
        qpp[6] = 1  #inicializar qp en 0 con qpp[4] = 1 rad/s**2
        qpp[7] = 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)
```

Los resultados obtenidos son:

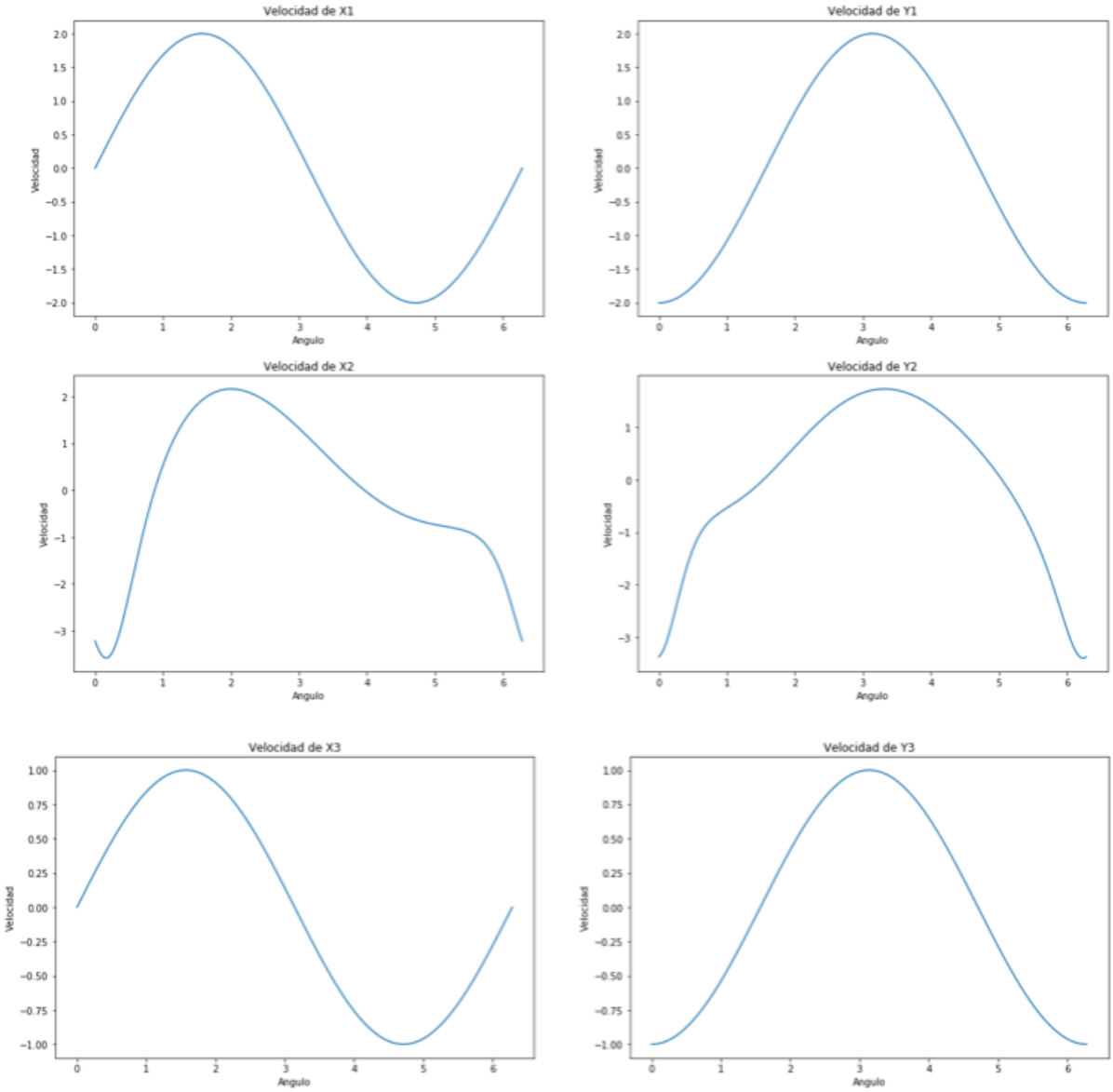


Figura 69: Gráficas velocidades mecanismo Cinco Barras

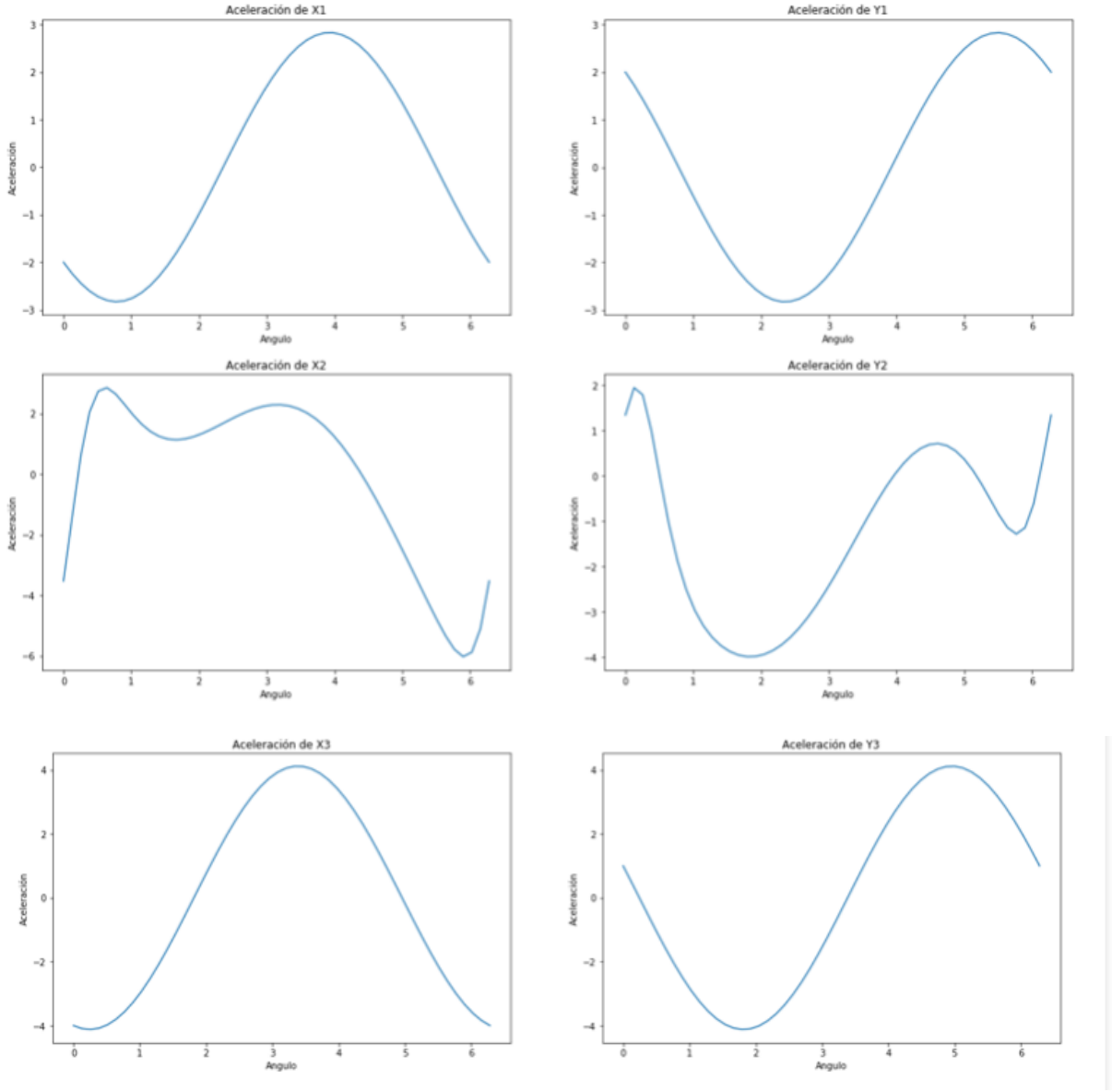


Figura 70: Aceleraciones mecanismo Cinco Barras

4.2.3 Animación

Para realizar la simulación del mecanismo Cinco Barras hay que tener en cuenta que hay posiciones que no podrán alcanzarse, ya que si se sitúa el segundo apoyo fijo a una distancia demasiado grande en relación con las longitudes de las otras cuatro barras, nunca podrá llegar. En la animación al darse esa situación realiza un movimiento anormal. Este problema se soluciona añadiendo una excepción del mismo modo que se hizo con el mecanismo anterior.

```
%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(( -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[6] = i*omega
    q[7] = i*omega*2

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

    # 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]
    beta = q[7]
```

```
    x=[meca["XA"], X1, X2, X3, meca["XB"]]
    y=[meca["YA"], Y1, Y2, Y3, 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())
```

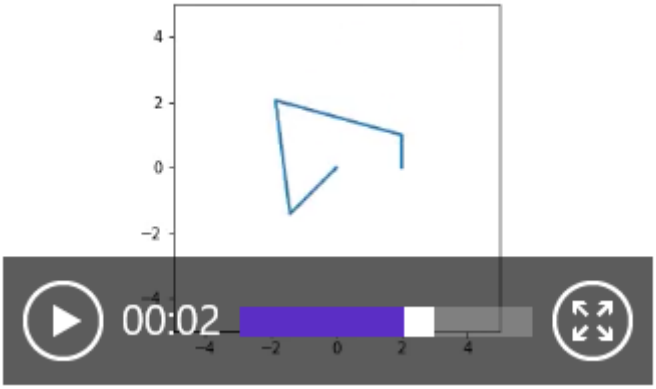


Figura 71: Animación mecanismo Cinco Barras

4.3 Biela-Manivela

El mecanismo Biela-Manivela (BM) está compuesto por una manivela y una barra llamada biela, la cual se encuentra articulada por un extremo con dicha manivela y por el otro con un elemento que describe un movimiento aleatorio.

La manivela, al girar la rueda, transmite el movimiento circular a la biela, experimentando esta un movimiento de vaivén. Este movimiento es reversible, lo que significa que también puede transformarse el movimiento de vaivén en uno de rotación.

Este mecanismo fue de vital importancia en el desarrollo de la locomotora de vapor. Actualmente tienen gran utilidad por ejemplo en motores de combustión interna, teniendo también otros usos, como por ejemplo, en máquinas de coser.

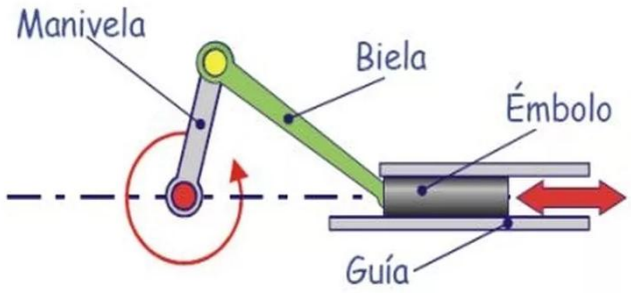


Figura 74: Mecanismo Biela-Manivela (Fuente: [28])

Figura 72: Motor de combustión interna (Fuente: [29])

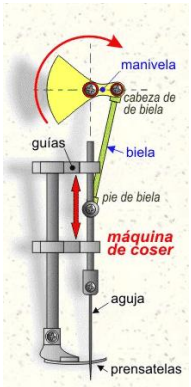


Figura 73: Máquina de coser (Fuente: [30])

4.3.1 Problema Posición

Siguiendo el mismo procedimiento que en el resto de mecanismos, primero se ha realizado una representación del mecanismo indicando los parámetros.

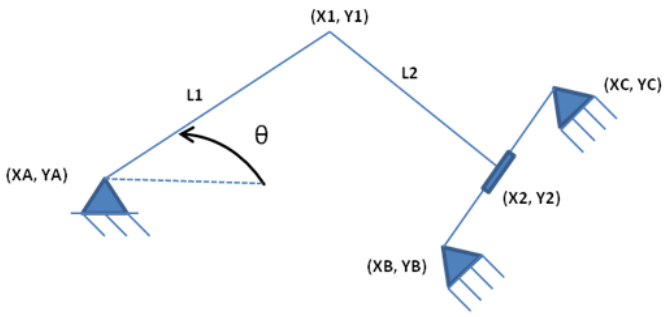


Figura 75: Mecanismo Biela-Manivela

Seguidamente se han calculado los grados de libertad y se ha definido el vector **q**.

PASO 2: GRADOS DE LIBERTAD

Los grados de libertad se calculan:

$$G = 3 \cdot (n - 1) - 2 \cdot PI - PII$$

Siendo:

PI -> Numero de pares binarios de un grado de libertad

PII -> Número de pares binarios de dos grados de libertad. En este caso tendríamos:

$$n = 4$$
$$P_I = 4$$
$$P_{II} = 0$$

Por lo tanto:

$$G = 3 \cdot (4 - 1) - 2 \cdot 4 - 0 = 1$$

PASO 3: DEFINICIÓN DEL VECTOR q

El vector $\mathbf{q}$ de dimensión $N \times 1$ contiene las coordenadas dependientes del mecanismo. Es decir, contiene las coordenadas que no se mantienen fijas ya que varían con el tiempo.

Lo hemos modelizado enpleando las 5 coordenadas:

$$\mathbf{q} = \begin{bmatrix} x_1 \\ y_1 \\ x_2 \\ y_2 \\ \theta \end{bmatrix}$$

Figura 76: GDL y vector q mecanismo Biela-Manivela

Tras estos pasos se importan las librerías y se realiza la lectura de datos, así como la definición de una posición inicial. Se necesitará conocer la longitud de la biela y la manivela, el ángulo de la manivela respecto a la horizontal y las coordenadas de los puntos fijos sobre los que se desplaza la deslizadera.

Los datos de prueba han sido:

$$L_1 = 1$$

$$L_2 = 2$$

$$\Theta = 0.5$$

$$X_B = 2$$

$$X_C = 3$$

$$Y_B = 0$$

$$Y_B = 0$$

Al definir la matriz **ϕ** se utiliza una restricción de sólido rígido para cada barra, otra ecuación para el ángulo y por último una para la deslizadera. Esta condición se basa en imponer que los extremos de la recta donde se desliza el émbolo y el punto de unión de la biela con este segmento deben estar alineados. El resultado es:

1. $Si \quad \cos(\theta) < \frac{1}{\sqrt{2}}$

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

2. $Si \quad \cos(\theta) > \frac{1}{\sqrt{2}}$

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

Figura 77: Matriz de restricciones mecanismo Biela-Manivela

Y su código:

```
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
```

Siguiendo con el procedimiento habitual, se deduce la matriz jacobiana y se implementa el código, para posteriormente resolver el problema posición.

1. $Si \ \cos(\theta) < \frac{1}{\sqrt{2}}$

$$\mathbf{\Phi_q} = \begin{bmatrix} 2X_1 & 2Y_1 & 0 & 0 & 0 \\ -2(X_2 - X_1) & -2(Y_2 - Y_1) & 2(X_2 - X_1) & 2(Y_2 - Y_1) & 0 \\ 0 & 0 & Y_C - Y_B & X_B - X_C & 0 \\ 1 & 0 & 0 & 0 & L1\sin(\theta) \end{bmatrix}$$

1. $Si \ \cos(\theta) > \frac{1}{\sqrt{2}}$

$$\mathbf{\Phi_q} = \begin{bmatrix} 2X_1 & 2Y_1 & 0 & 0 & 0 \\ -2(X_2 - X_1) & -2(Y_2 - Y_1) & 2(X_2 - X_1) & 2(Y_2 - Y_1) & 0 \\ 0 & 0 & Y_C - Y_B & X_B - X_C & 0 \\ 0 & 1 & 0 & 0 & -L1\cos(\theta) \end{bmatrix}$$

Figura 78: Matriz jacobiana mecanismo Biela-Manivela

```
#PASO 7


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["XB"] - meca["XC"]

    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
```

```
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)
```

Dado que ya se ha visto cómo son presentados los resultados en este proceso iterativo, par este caso solo se van a insertar los resultados obtenidos para la última iteración.

```
q=
array([[0.87758256],
       [0.47942554],
       [2.81927027],
       [0. ],
       [0.5 ]])
```

```
Phi=
array([[ 9.06563713e-12],
       [-8.88178420e-16],
       [ 0.00000000e+00],
       [ 0.00000000e+00],
       [ 0.00000000e+00]])
```

```
rango=5

error iter8= 7.304419941333233e-12

num iters:8
```

Para finalizar, se representa el mecanismo. Para indicar en qué eje se desplaza el émbolo, se ha dibujado con trazo discontinuo.

```
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])     #[pos inicial (x1,x2), pos
final (y1,y2)]
    plt.plot ([X1, X2], [Y1, Y2])
    plt.plot ([meca["XB"],meca["XC"]], [meca["YB"],meca["YC"]], linestyle='dashed')

    plt.plot(meca["XA"], meca["YA"], 'bo')
    plt.plot(meca["XB"], meca["YB"], 'go')
    plt.plot(meca["XC"], meca["YC"], 'go')
    plt.show()#block=False)
    return

dibuja_mecanismo(q,meca)
```

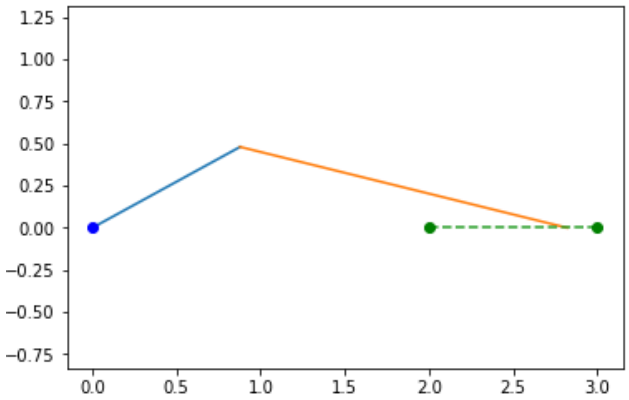


Figura 79: Dibujo mecanismo Biela-Manivela

4.3.2 Problemas Velocidad y Aceleración

La parte de este Notebook correspondiente al problema velocidad es exactamente igual que para el Cuatro Barras, ya que tienen las mismas variables dependientes. Debido a este motivo, solo se indicará el resultado obtenido para los datos de prueba y se procederá a la resolución del problema aceleración.

```
qp=
array([[ 0.54630249],
       [-1. ],
       [ 0.79321425],
       [ 0. ],
       [ 1. ]])
```

En el problema aceleración, la **b** obtenida es:

1. $Si \quad \cos(\theta) < \frac{1}{\sqrt{2}}$

$$\mathbf{b} = \begin{bmatrix} 2\dot{X}_1^2 + 2\dot{Y}_1^2 \\ -2\dot{X}_1(\dot{X}_2 - \dot{X}_1) \quad -2\dot{Y}_1(\dot{Y}_2 - \dot{Y}_1) \quad 2\dot{X}_2(\dot{X}_2 - \dot{X}_1) \quad 2\dot{Y}_2(\dot{Y}_2 - \dot{Y}_1) \quad 0 \\ 2\dot{X}_2^2 + 2\dot{Y}_2^2 \\ \dot{\theta}^2 L_1 \cos(\theta) \\ 1 \end{bmatrix}$$

2. $Si \quad \cos(\theta) > \frac{1}{\sqrt{2}}$

$$\mathbf{b} = \begin{bmatrix} 2\dot{X}_1^2 + 2\dot{Y}_1^2 \\ -2\dot{X}_1(\dot{X}_2 - \dot{X}_1) \quad -2\dot{Y}_1(\dot{Y}_2 - \dot{Y}_1) \quad 2\dot{X}_2(\dot{X}_2 - \dot{X}_1) \quad 2\dot{Y}_2(\dot{Y}_2 - \dot{Y}_1) \quad 0 \\ 2\dot{X}_2^2 + 2\dot{Y}_2^2 \\ \dot{\theta}^2 L_1 \sin(\theta) \\ 1 \end{bmatrix}$$

Figura 80: Matriz b mecanismo Biela-Manivela

El código para la resolución del problema aceleración es:

```
#PASO 3


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))

    b[4] = 1 #Aceleracion conocida
    qpp = np.linalg.solve(-jacob_Phiq(q,meca),b)



    return qpp


qpp=np.zeros((5,1))
qpp=resuelve_prob_aceleracion(q,qp,qpp,meca)
qpp
```

Apareciendo como resultado en la ventana de salida:

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

Y por último, se grafican las velocidades y aceleraciones de las coordenadas dependientes. El código por el cual se obtienen, al igual que ocurrió con el problema velocidad, es exactamente igual que en el primer mecanismo de este proyecto. Por esta razón solo se insertarán las gráficas.

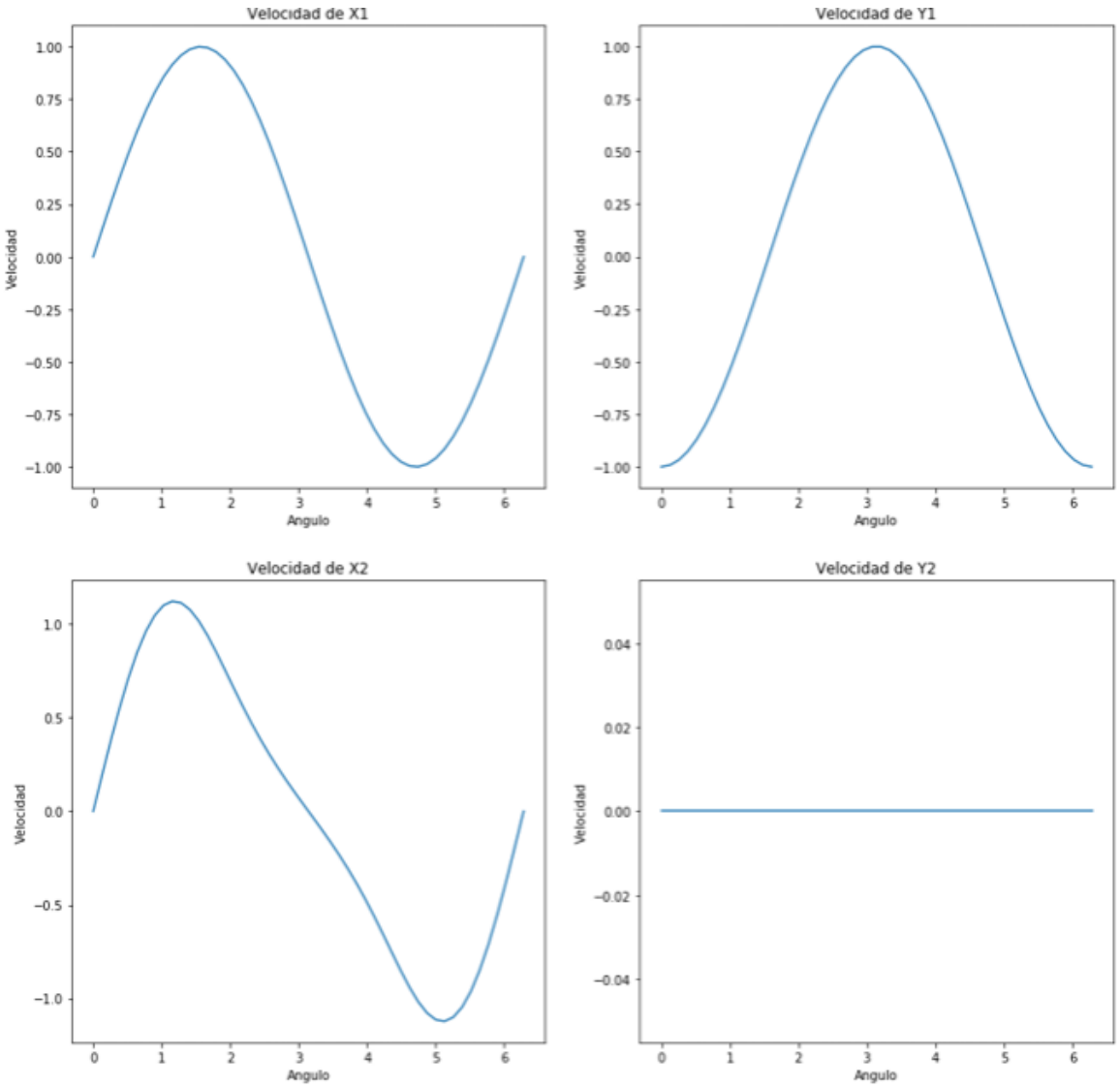


Figura 81: Gráficas velocidad mecanismo Biela-Manivela

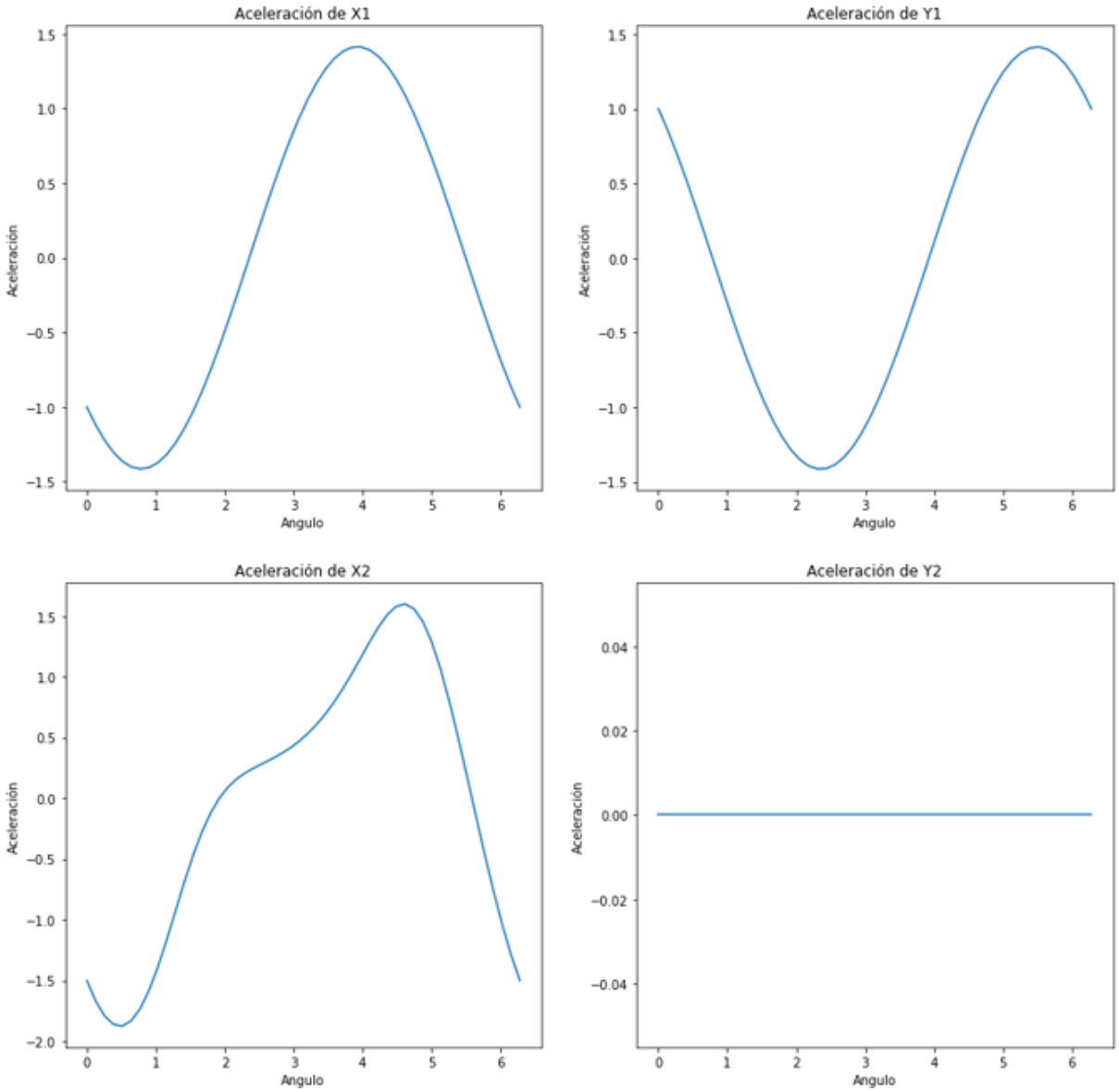


Figura 82: Gráficas aceleración mecanismo Biela-Manivela

4.3.3 Animación

En este Notebook también se ha añadido una excepción para el caso de superar el número de iteraciones posibles. La animación se ha hecho de la forma:

```
%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"], 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())
```

Siendo el resultado:

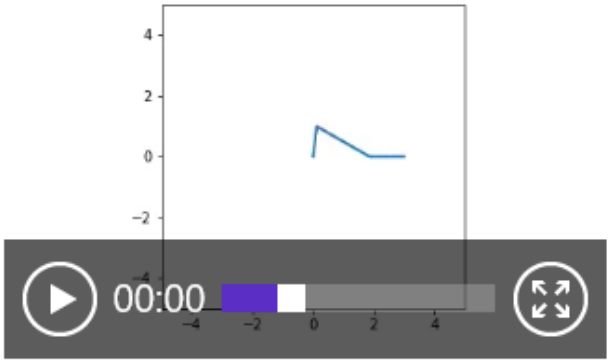


Figura 83: Animación mecanismo Biela-Manivela

4.4 Biela-Manivela Invertida

El mecanismo conocido como Biela-Manivela Invertida (BMI) se diferencia del tradicional Biela-Manivela en que el eje donde se desplaza el émbolo no está delimitado por dos apoyos fijos, sino por un punto móvil y un extremo libre. La deslizadera se une un apoyo fijo a través de una barra, lo que implica que en este mecanismo se tienen tres barras en lugar de dos.

Este mecanismo a su vez forma parte de un mecanismo de retorno rápido, como por ejemplo en una limadora o cepilladora. También está presente en los sistemas con cilindros, como podría ser un remolque.

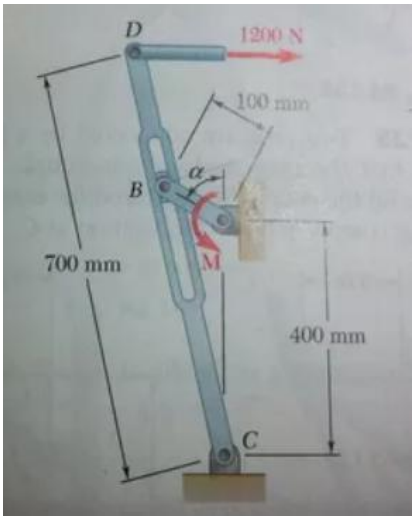


Figura 84: Mecanismo BMI [Fuente: [31]]

Figura 85: Limadora [Fuente: [32]]

4.4.1 Problema Posición

El modelado del mecanismo es el siguiente:

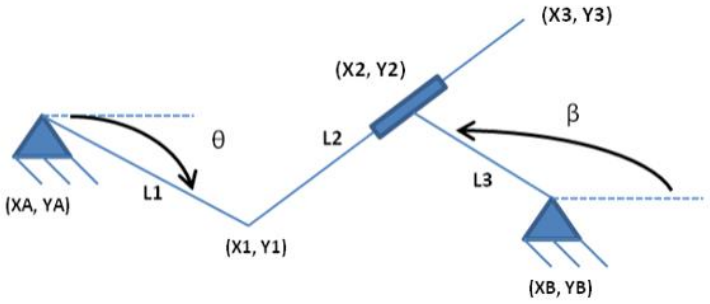


Figura 86: Modelado mecanismo Biela-Manivela Invertida

Hay que seguir el mismo procedimiento que se llevó a cabo en el resto de mecanismos de este proyecto. Esto significa que en primer lugar es necesario calcular el número de grados de libertad del mecanismo y definir el vector de coordenadas dependientes, añadiendo además tantas variables como grados de libertad tenga el mecanismo.

PASO 2: GRADOS DE LIBERTAD

Los grados de libertad se calculan:

$$G = 3 \cdot (n-1) - 2 \cdot PI - PII$$

Siendo:

PI -> Numero de pares binarios de un grado de libertad

PII -> Número de pares binarios de dos grados de libertad. En este caso tendríamos:

$$n = 5$$
$$P_I = 5$$
$$P_{II} = 0$$

Por lo tanto:

$$G = 3 \cdot (5-1) - 2 \cdot 5 - 0 = 2$$

PASO 3: DEFINICIÓN DEL VECTOR q

El vector $\mathbf{q}$ de dimensión $N \times 1$ contiene las coordenadas dependientes del mecanismo. Es decir, contiene las coordenadas que no se mantienen fijas ya que varían con el tiempo.

Lo hemos modelizado enpleando las 8 coordenadas:

$$\mathbf{q} = \begin{bmatrix} x_1 \\ y_1 \\ x_2 \\ y_2 \\ x_3 \\ y_3 \\ \theta \\ \beta \end{bmatrix}$$

Figura 87: GDL y vector q mecanismo Biela-Manivela Invertida

Los parámetros que definen al mecanismo y por tanto se pedirán por teclado son las longitudes de las tres barras, las coordenadas de los puntos fijos **A** y **B** y por último los ángulos que forman las barras **1** y **3** respecto a la horizontal.

Como datos de prueba se han tomado:

$$L_1 = 2$$

$$L_2 = 3$$

$$L_3 = 1$$

$$\Theta = 0.5$$

$$\beta = 0.8$$

$$X_B = 2$$

$$Y_B = 0$$

Hay que aplicar una restricción de sólido rígido para cada una de las barras y una ecuación para la deslizadera, al igual que en el Biela-Manivela. También habrá que añadir una ecuación para cada uno de los ángulos, ya que a diferencia del mecanismo anterior, el actual tiene dos grados de libertad.

Para nuestro caso tendríamos para el ángulo θ:

$$Si \;\; \cos(\theta) < 0.95 \rightarrow (X_1 - X_A) - L_1 \cdot \cos(\theta)$$

$$Si \;\; \cos(\theta) > 0.95 \rightarrow (Y_1 - Y_A) - L_1 \cdot \sin(\theta)$$

Para el ángulo β:

$$Si \;\; \cos(\beta) < 0.95 \rightarrow (X_3 - X_B) - L_3 \cdot \cos(\beta)$$

$$Si \;\; \cos(\beta) > 0.95 \rightarrow (Y_3 - Y_B) - L_3 \cdot \sin(\beta)$$

La matriz quedaría:

$$\mathbf{\Phi} = \begin{bmatrix} (X_1 - X_A)^2 + (Y_1 - Y_A)^2 - L_1^2 \\ (X_2 - X_1)^2 + (Y_2 - Y_1)^2 - L_2^2 \\ (X_B - X_3)^2 + (Y_B - Y_3)^2 - L_3^2 \\ (X_3 - X_1)(Y_2 - Y_1) - (X_2 - X_1)(Y_3 - Y_1) \\ \Phi(4) \\ \Phi(5) \end{bmatrix}$$

Figura 88: Matriz de restricciones mecanismo Biela-Manivela Invertida

```
#PASO 6

def Phi (q,meca):
    #Inicializa a cero Phi
    Phi = np.zeros((8,1))

    #Extraer coordenadas
    X1 = q[0]
    Y1 = q[1]
    X2 = q[2]
    Y2 = q[3]
    X3 = q[4]
    Y3 = q[5]
    theta = q[6]
    beta = q[7]


    Phi[0] = X1**2 + Y1**2 - meca["L1"]**2


    Phi[1] = (X2-X1)**2 + (Y2-Y1)**2 - meca["L2"]**2
    Phi[2] = (meca["XB"]-X3)**2 + (meca["YB"]-Y3)**2 - meca["L3"]**2
    Phi[3] = (X3-X1)*(Y2-Y1) - (X2-X1)*(Y3-Y1)

    if (abs(math.cos(theta)) < 0.95):
        Phi[4] = X1-meca["L1"]*math.cos(theta)
    else:
        Phi[4] = Y1-meca["L1"]*math.sin(theta)


    if (abs(math.cos(beta)) < 0.95):
        Phi[5] = (X3-meca["XB"])-meca["L3"]*math.cos(beta)
    else:
        Phi[5] = (Y3-meca["YB"])-meca["L3"]*math.sin(beta)


    return Phi
```

El siguiente paso es calcular el jacobiano:

1. $Si \ \cos(\theta) < 0.95$

   $\Phi_q(4,0) = 1 \ \Phi_q(4,6) = L_1 \cdot \sin(\theta)$

2. $Si \ \cos(\theta) > 0.95$

   $\Phi_q(4,1) = 1 \ \Phi_q(4,6) = -L_1 \cdot \cos(\theta)$

1. $Si \ \cos(\beta) < 0.95$

   $\Phi_q(5,4) = 1 \ \Phi_q(5,7) = L_3 \cdot \sin(\beta)$

2. $Si \ \cos(\beta) > \frac{1}{\sqrt{2}}$

   $\Phi_q(5,5) = 1 \ \Phi_q(5,7) = -L_3 \cdot \cos(\beta)$

La matriz jacobiana sería:

$$\mathbf{\Phi_q} = \begin{bmatrix} 2X_1 & 2Y_1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -2(X_2 - X_1) & -2(Y_2 - Y_1) & 2(X_2 - X_1) & 2(Y_2 - Y_1) & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -2(X_B - X_3) & -2(Y_B - Y_3) & 0 & 0 \\ -2(Y_2 - Y_1) + (Y_3 - Y_1) & (X_2 - X_1) - (X_3 - X_1) & -Y_3 - Y_1 & X_3 - X_1 & Y_2 - Y_1 & -(X_2 - X_1) & 0 & 0 \\ \Phi_q(4,0) & \Phi_q(4,1) & 0 & 0 & 0 & 0 & \Phi_q(4,6) & \Phi_q(4,7) \\ 0 & 0 & 0 & 0 & \Phi_q(5,4) & \Phi_q(5,5) & \Phi_q(5,6) & \Phi_q(5,7) \end{bmatrix}$$

Figura 89: Matriz jacobiana mecanismo Biela-Manivela Invertida

```
# PASO 7


def jacob_Phiq(q,meca):
    #Inicializa a cero la matriz jacobiana
    Jacob = np.zeros((8,8))

    #Extraer coordenadas

    X1 = q[0]
    Y1 = q[1]
    X2 = q[2]
    Y2 = q[3]
    X3 = q[4]
    Y3 = q[5]
    theta = q[6]
    beta = q[7]

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

    if (abs(math.cos(theta)) < 0.95 ):
        Jacob[4,6] = meca["L1"]*math.sin(theta)
        Jacob[4,0] = 1
    else:
        Jacob[4,6] = -meca["L1"]*math.cos(theta)
        Jacob[4,1] = 1



    if (abs(math.cos(beta)) < 0.95 ):
        Jacob[5,7] = meca["L3"]*math.sin(beta)
        Jacob[5,4] = 1
    else:
        Jacob[5,7] = -meca["L3"]*math.cos(beta)
        Jacob[5,5] = 1



    Jacob[6,6] = 1
    Jacob[7,7] = 1


    return Jacob
```

Al tener el mismo vector **q** que el mecanismo Cinco Barras, el problema posición se resuelve de la misma manera. Por esta razón no se va a insertar esa parte del Notebook y solo se indicará el resultado final.

```
q=                                          Phi=
array([[1.75516512],                         array([[ 0.00000000e+00],
       [0.95885108],                                [ 1.77635684e-15],
       [4.66110172],                                [-1.11022302e-16],
       [0.21351053],                                [ 1.11022302e-16],
       [2.69670671],                                [ 0.00000000e+00],
       [0.71735609],                                [-1.11022302e-16],
            [0.5 ],                                 [ 0.00000000e+00],
           [0.8 ]])                                 [ 0.00000000e+00]])

                                  rango=8
                    error iter11= 3.395998511097105e-16
```

Y para finalizar, se dibuja el mecanismo:

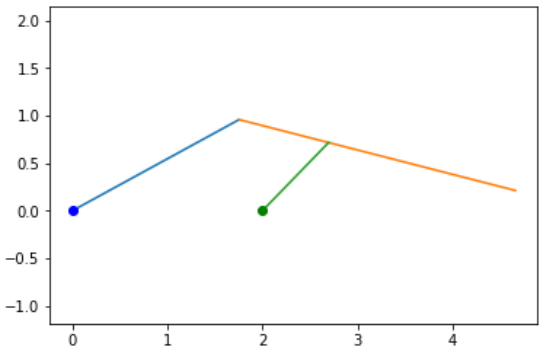


Figura 90: Dibujo mecanismo Biela-Manivela Invertida

4.4.2 Problemas Velocidad y Aceleración

Del mismo modo que ocurrió con el Biela-Manivela y el Cuatro Barras, el mecanismo Biela-Manivela Invertida tiene la misma resolución para el problema velocidad que el Cinco Barras. Por esta razón se va a proceder a dar los detalles del problema aceleración:

$$\mathbf{b} = \begin{bmatrix} 2\dot{X}_1^2 & 2\dot{Y}_1^2 & 0 & 0 & 0 & 0 & 0 & 0 \\ -2\dot{X}_1(\dot{X}_2 - \dot{X}_1) & -2\dot{Y}_1(\dot{Y}_2 - \dot{Y}_1) & 2\dot{X}_2(\dot{X}_2 - \dot{X}_1) & 2\dot{Y}_2(\dot{Y}_2 - \dot{Y}_1) & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2\dot{X}_3^2 & 2\dot{Y}_3^2 & 0 & 0 \\ -\dot{X}_1((\dot{Y}_2 - \dot{Y}_1) + (\dot{Y}_3 - \dot{Y}_1)) & \dot{Y}_1((\dot{X}_2 - \dot{X}_1) - (\dot{X}_3 - \dot{X}_1)) & \dot{X}_2(-\dot{Y}_3 - \dot{Y}_1) & \dot{Y}_2(\dot{X}_3 - \dot{X}_1) & \dot{X}_3(\dot{Y}_2 - \dot{Y}_1) & \dot{Y}_3(-(\dot{X}_2 - \dot{X}_1)) & 0 & 0 \\ b(4) \\ b(5) \\ 1 \\ 1 \end{bmatrix}$$

1. $Si \quad \cos(\theta) < 0.95 \rightarrow b(4) = \dot{\theta}^2 \cdot L_1 \cdot \cos(\theta)$
2. $Si \quad \cos(\theta) > 0.95 \rightarrow b(4) = \dot{\theta}^2 \cdot L_1 \cdot \sin(\theta)$
3. $Si \quad \cos(\beta) < 0.95 \rightarrow b(5) = \dot{\beta}^2 \cdot L_3 \cdot \cos(\beta)$
4. $Si \quad \cos(\beta) > 0.95 \rightarrow b(5) = \dot{\beta}^2 \cdot L_3 \cdot \sin(\beta)$

Figura 91: Matriz b mecanismo Biela-Manivela Invertida

```
#PASO 3


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]
    beta = q[7]
    #Extraer las velocidades
    X1q = qp[0]
    Y1q = qp[1]
    X2q = qp[2]
    Y2q = qp[3]
    X3q = qp[4]
    Y3q = qp[5]
    thetaq = qp[6]
    betaq = qp[7]
    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*X3q**2 + 2*Y3q**2
    b[3] = -2*X1q*((Y2q-Y1q)+(Y3q-Y1q)) + Y1q*((X2q-X1q)-(X3q-X1q)) + X2q*(-Y3q-
Y1q) + Y2q*(X3q-X1q) + X3q*(Y2q-Y1q) -Y3q*(X2q-X1q)


    if (abs(math.cos(theta)) < 0.95 ):
        b[4] = thetaq**2 * (meca["L1"] * math.cos(theta))
    else:
        b[4] = thetaq**2 * (meca["L1"] * math.sin(theta))


    if (abs(math.cos(beta)) < 0.95 ):
        b[5] = betaq**2 * (meca["L3"] * math.cos(beta))
    else:
        b[5] = betaq**2 * (meca["L3"] * math.sin(beta))
    b[6] = 1 #Aceleracion conocida
    b[7] = 1
```

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



    return qpp



qpp=np.zeros ((8,1))
qpp = resuelve_prob_aceleracion (q,qp,qpp,meca)
qpp
```

Los resultados obtenidos para los problemas velocidad y aceleración han sido:

```
qp=
array([[-0.95885108],
       [ 1.75516512],
       [-1.31819903],
       [ 0.35413793],
       [-1.43471218],
       [ 1.39341342],
       [ 1. ],
       [ 2. ]])
```

```
qpp=
array([[-0.79631405 ],
       [-2.7140162  ],
       [-2.73535687 ],
       [-9.3771278  ],
       [-2.06947075 ],
       [-3.56613107 ],
       [-1. ],
       [-1. ]])
```

El código con el que se generan las gráficas es igual que en el mecanismo Cinco Barras, motivo por el cual en las páginas siguientes se insertarán directamente las gráficas de las velocidades *(Figura [92])* y aceleraciones *(Figura [93])*.

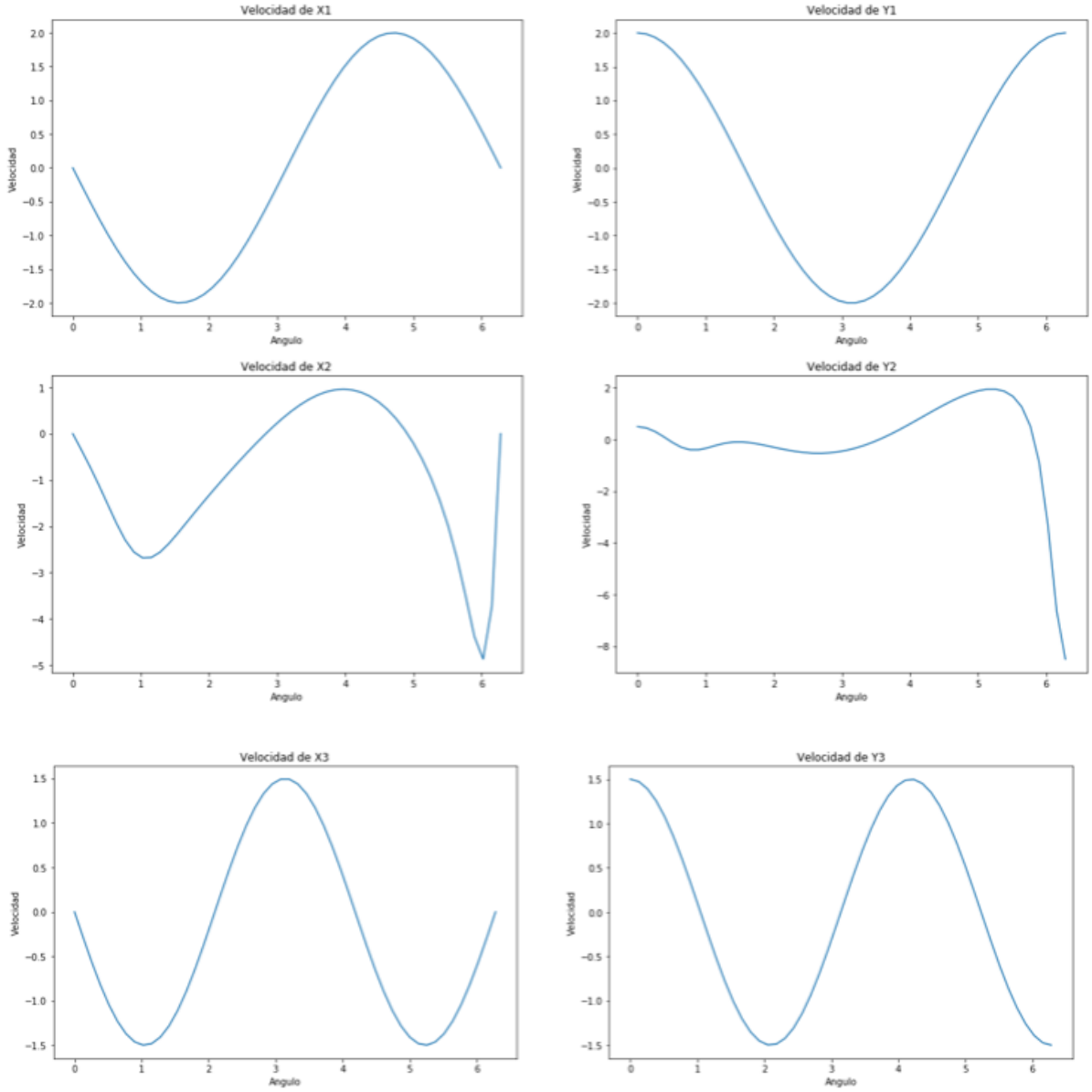


Figura 92: Gráficas velocidades mecanismo Biela-Manivela Invertida

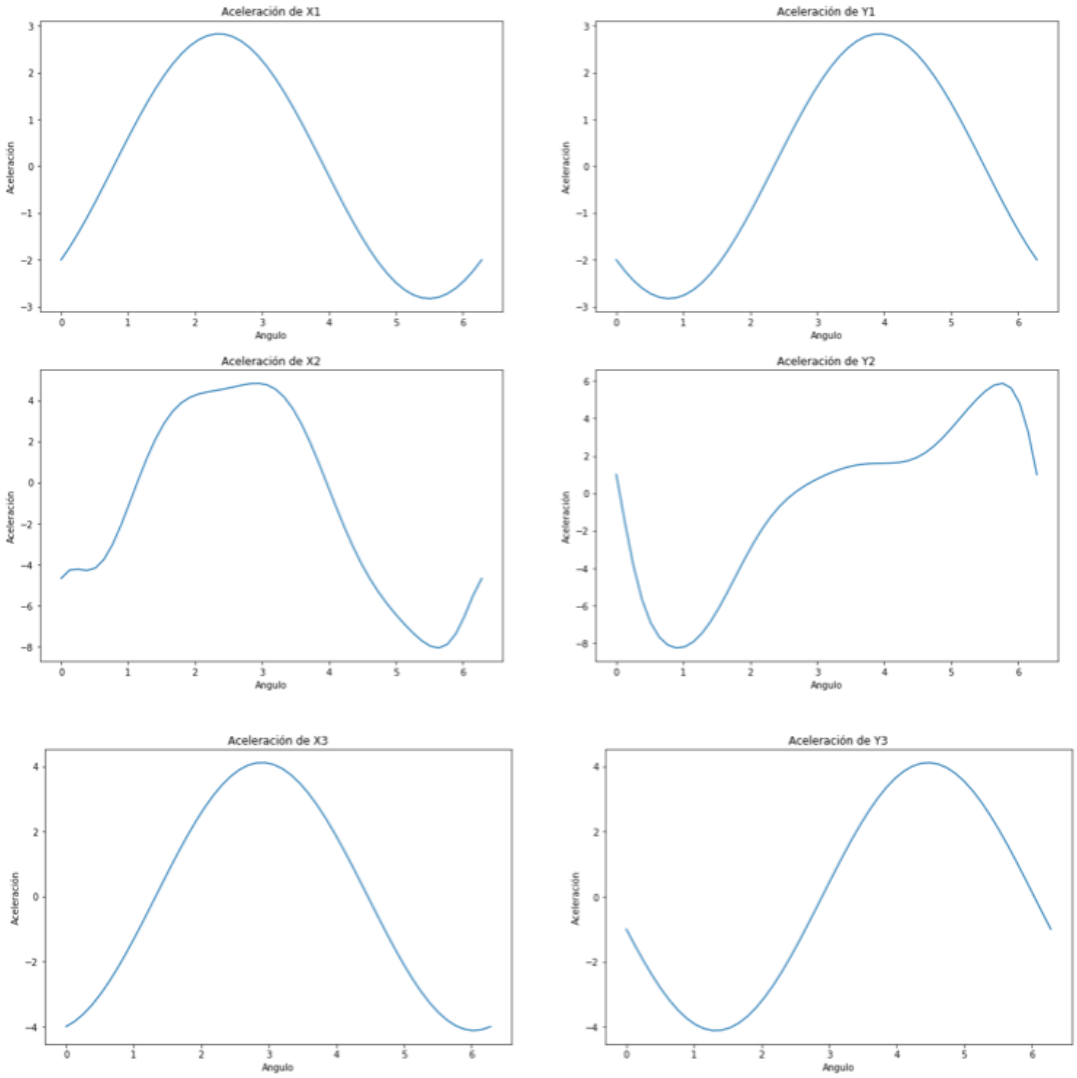


Figura 93: Gráficas aceleraciones mecanismo Biela-Manivela Invertida

4.4.3 Animación

Se va a insertar directamente la animación y la representación del mecanismo en función del ángulo, ya que el código es idéntico al del mecanismo Cinco Barras.

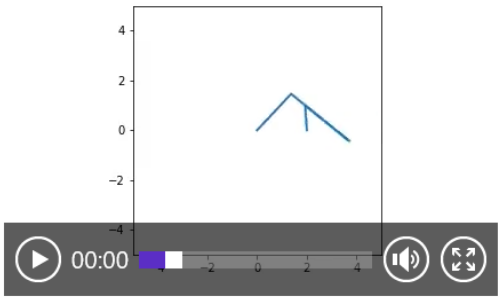


Figura 94: Animación mecanismo Biela-Manivela Invertida

Capítulo 5:

Conclusiones y Trabajos futuros

5. Conclusiones y Trabajos futuros

5.1. Conclusiones

El objetivo inicial era diseñar e implementar un material de apoyo interactivo para la asignatura Teoría de Mecanismos, desarrollando una serie de programas que resolviesen por métodos cinemáticos los problemas de posición, velocidad y aceleración de varios de los mecanismos más comunes a la par que explicasen lo que ocurre en cada paso. Estos programas o *Notebooks Jupyter* tenían que mejorar la experiencia docente en prácticas y servir como material de estudio autónomo para el alumno.

Además, para aprovechar la particularidad de Jupyter Notebook de poder ser ejecutado desde un servidor remoto, se planteó el objetivo de ejecutar los programas creados desde un teléfono móvil.

Respecto al objetivo de desarrollar un material de estudio autónomo para el alumno, se puede afirmar que los *Notebooks* implementados son de gran utilidad para el estudio del análisis cinemático por métodos numéricos en Teoría de Mecanismos. Esto puede asegurarse por diferentes razones. Una es la estructura de los *Notebooks,* puesto que al seguirse un orden determinado para cada tipo de problema facilita al alumno la asimilación del contenido, además de percibir con claridad las diferencias entre los diferentes mecanismos. Otro motivo es la intercalación de celdas de texto con teoría y las de código, ya que de esta manera puede comprender el significado de la salida de cada celda. Por último, las celdas de texto en sí son imprescindibles para considerar que los *Notebooks* son un material de estudio autónomo. Esto se debe a que en ellas se aúnan explicaciones de diferentes fuentes de una manera en la que es mucho más fácil de entender el procedimiento, estando ilustrado en caso de ser necesario, y explicado el proceso de simplificación de las ecuaciones sin omitir ningún paso.

Por otro lado, el objetivo de que mejore la experiencia docente en prácticas no puede asegurarse que se haya cumplido. El motivo de no poder afirmarlo es sencillamente que la asignatura finalizó antes de haber completado el proyecto, motivo por el cual se debe esperar a otro curso para hacer la prueba.

El último objetivo, según el cual se pretendía poder ejecutar los programas desarrollados desde un teléfono móvil, ha sido realizado.

5.2. Trabajos Futuros

Para el futuro se ha propuesto el objetivo de utilizar Jupyter Notebook en diferentes computadores simultáneamente con solo un servidor utilizando JupyterHub [33], que se trata de un concentrador multiusuario que genera, administra y distribuye varias instancias del servidor portátil Jupyter para un solo usuario.

JupyterHub tiene varias distribuciones que se centran en dos casos. Uno es para una pequeña cantidad de usuarios y un solo servidor y la otra es para permitir más de 100 usuarios utilizando una cantidad variable de servidores en la nube.

Bibliografía

Bibliografía

[1] J.L. Blanco Claraco, et al, "Memoria final de acciones del proyecto de grupo docente curso 2014/2015", Universidad de Almería, Almería, Proy, 2015.

[2] J.L Torres Moreno, et al, "MecaSENS 3D: Creación de mecanismos para un laboratorio multipuesto empleando impresión 3D", Universidad de Almeria, Almería, Proy, 2015.

[3] J.L. Torres Moreno, et al, "MecaSENS 3D: Creación de mecanismos para un laboratorio multipuesto empleando impresión 3D. Balance del Segundo año", Universidad de Almería, Almería, Proy, 2016.

[4] J.L Torres Moreno; A. Giménez Fernández; J.L Blanco Claraco; F.J Garrido Jiménez, "MecaServer: Servidor de contenido interactivo y gestión de un laboratorio remoto de Ingeniería Mecánica", Universidad de Almería, Almería, Proy, 2016.

[5] Torres Moreno, "torresmoreno/MecaServer", GitHub, 2016. [Online]. Available: https://github.com/torresmoreno/MecaServer. [Accessed: 13- Apr- 2019].

[6] "Guía Docente", apuntes de clase de Teoría de Mecanismos, Departamento de Ingeniería, Universidad de Almería, 2018.

[7] A. Avello Iturriagagoitia, *Teoría de máquinas*. 2nd.ed. Navarra: Universidad de Navarra, 2014.

[8] J.L. Guzmán, et al, “Laboratorio Virtual para la Enseñanza de Control Climático de Invernaderos”, *Revista Iberoamericana de Automática e Informática Industrial,* vol. 2, no. 2, pp. 82-92, Septiembre 2010.

[9] "Easy Java Simulations Wiki | Main / EJS Home Page", Um.es, 2009. [Online]. Available: https://www.um.es/fem/EjsWiki/Main/HomePage. [Accessed: 05- Jul- 2018].

[10] W. Christian and F. Esquembre, "Easy Java/Javascript Simulations Manual", Um.es, 2015. [Online]. Available: https://www.um.es/fem/EjsWiki/uploads/Download/EjsS_Manual.pdf. [Accessed: 05- Jul- 2018].

[11] "Coursera | Online Courses From Top Universities. Join for Free", Coursera. [Online]. Available: https://www.coursera.org/learn/aprendiendo-programar-python/home/welcome. [Accessed: 23- Sep- 2018].

[12] "Quickstart tutorial — NumPy v1.15 Manual", Docs.scipy.org. [Online]. Available: https://docs.scipy.org/doc/numpy-1.15.1/user/quickstart.html. [Accessed: 27- Sep- 2018].

[13] P. Recuero, "Python para todos (3): ScyPy, NumPy, Pandas ...¿Qué librerías necesitamos? - Think Big", Think Big, 2018. [Online]. Available: https://empresas.blogthinkbig.com/python-todos-3-librerias/. [Accessed: 28- Sep- 2018].

[14] "Introducción a la librería NumPy de Python - Parte 1 - Ligdi González", Ligdi González, 2018. [Online]. Available: http://ligdigonzalez.com/introduccion-a-numpy-python-1/. [Accessed: 15- Jan- 2019].

[15] "Introducción a la librería NumPy de Python - Parte 2 - Ligdi González", Ligdi González, 2018. [Online]. Available: http://ligdigonzalez.com/introduccion-a-numpy-python-2/. [Accessed: 15- Jan- 2019].

[16] "Introducción a la Librería Matplotlib de Python – Parte 1 - Ligdi González", Ligdi González, 2018. [Online]. Available: http://ligdigonzalez.com/libreria-pandas-de-matplotlib-tutorial/. [Accessed: 05- Mar- 2019].

[17] "Matplotlib: Python plotting — Matplotlib 3.0.3 documentation", Matplotlib.org. [Online]. Available: https://matplotlib.org/. [Accessed: 06- Mar- 2019].

[18] "Matplotlib", Es.wikipedia.org. [Online]. Available: https://es.wikipedia.org/wiki/Matplotlib. [Accessed: 06- Mar- 2019].

[19] K. Correoso, "Creando una animación con matplotlib y ffmpeg | Pybonacci", Pybonacci.org, 2012. [Online]. Available: https://www.pybonacci.org/2012/12/16/creando-una-animacion-con-matplotlib-y-ffmpeg/. [Accessed: 09- Mar- 2019].

[20] L. Tiao, "Embedding Matplotlib Animations in Jupyter as Interactive JavaScript W", Louis Tiao, 2018. [Online]. Available: http://louistiao.me/posts/notebooks/embedding-matplotlib-animations-in-jupyter-as-interactive-javascript-widgets/. [Accessed: 10- Mar- 2019].

[21] "Anaconda (distribución de Python)", Es.wikipedia.org. [Online]. Available: https://es.wikipedia.org/wiki/Anaconda_%28distribuci%C3%B3n_de_Python%29. [Accessed: 10- Feb- 2019].

[22] "Home - Anaconda", Anaconda. [Online]. Available: https://www.anaconda.com/. [Accessed: 10- Feb- 2019].

[23] D. Bayas, "Instalación de la plataforma Anaconda en Windows 10", Ciencia de datos para inexpertos, 2018. [Online]. Available:

https://danilobayas.wordpress.com/2018/11/09/instalacion-de-la-plataforma-anaconda-en-windows/. [Accessed: 05- Mar- 2019].

[24] "Tema 4: Análisis cinemático por métodos numéricos", Apuntes de clase de Teoría de Mecanismos, Departamento de Ingeniería, Universidad de Almería, 2018.

[25] " Práctica 1: Ejercicios temas 1 y 2", Apuntes de clase de Teoría de Mecanismos, Universidad de Almería, 2018.

[26] h. iPython/Jupyter Notebook and Pandas and A. Sobolev, "iPython/Jupyter Notebook and Pandas, how to plot multiple graphs in a for loop?", Stack Overflow, 2018. [Online]. Available: https://stackoverflow.com/questions/29532894/ipython-jupyter-notebook-and-pandas-how-to-plot-multiple-graphs-in-a-for-loop/29533687#29533687. [Accessed: 22- May- 2019].

[27] I. Mac?, "Ipython Notebook: where is jupyter_notebook_config.py in Mac?", Stack Overflow, 2017. [Online]. Available: https://stackoverflow.com/questions/32625939/ipython-notebook-where-is-jupyter-notebook-config-py-in-mac/32626312#32626312. [Accessed: 18- May- 2019].

[28] "En rectilíneo alternativo | Tecnología", Nabitec.wordpress.com, 2014. [Online]. Available: https://nabitec.wordpress.com/category/elementos-transformadores-del-movimiento/circular-continuo/en-rectilineo-alternativo/. [Accessed: 15- May- 2019].

[29] V. Gomez, "Motor de combustión interna – SolidWorks", Vigg, 2017. [Online]. Available: https://victorgomezgarcia.wordpress.com/2017/04/02/motor-de-combustion-interna-solidworks/. [Accessed: 20- May- 2019].

[30] "Mecanismo biela-manivela", Concurso.cnice.mec.es, 2005. [Online]. Available: http://concurso.cnice.mec.es/cnice2006/material107/mecanismos/mec_biela-manivela.htm. [Accessed: 20- May- 2019].

[31] "Chegg.com", Chegg.com. [Online]. Available: https://www.chegg.com/homework-help/questions-and-answers/whitworth-mechanism-shown-used-produce-quick-return-motion-point-d-block-b-pinned-crank-ab-q3198630. [Accessed: 20- May- 2019].

[32] "Cómo funciona una LIMADORA mecánica? Partes, tipos y usos", Comofunciona.co.com, 2016. [Online]. Available: http://comofunciona.co.com/la-limadora-mecanica/. [Accessed: 20- May- 2019].

[33] "JupyterHub — JupyterHub 1.0.1dev documentation", Jupyterhub.readthedocs.io. [Online]. Available: https://jupyterhub.readthedocs.io/en/latest/. [Accessed: 15- May- 2019].

Anexos

Procedimiento para poder utilizar Jupyter Notebook en el teléfono móvil.

Para utilizar Jupyter Notebbok en el teléfono móvil, es necesario que esté conectado a la misma red *Wifi* que el computador en el que se hayan ejecutado Anaconda.

El procedimiento a seguir es sencillo. En primer lugar, hay que abrir en el buscador de Windows “anaconda prompt”.

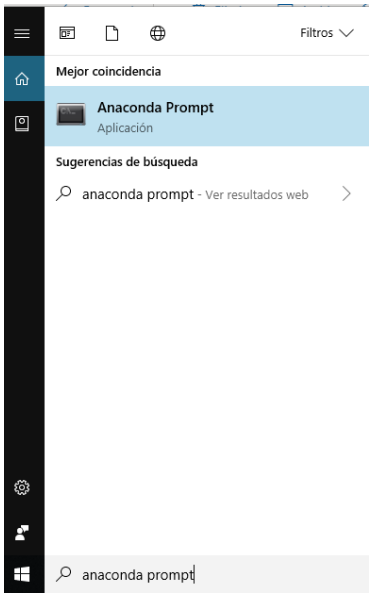


Figura 95: Búsqueda de anaconda prompt

Una vez inicializado, ejecutar el comando “Jupyter Notebook –generate—config” y se indica que se quiere sobrescribir en caso de que se pregunte.

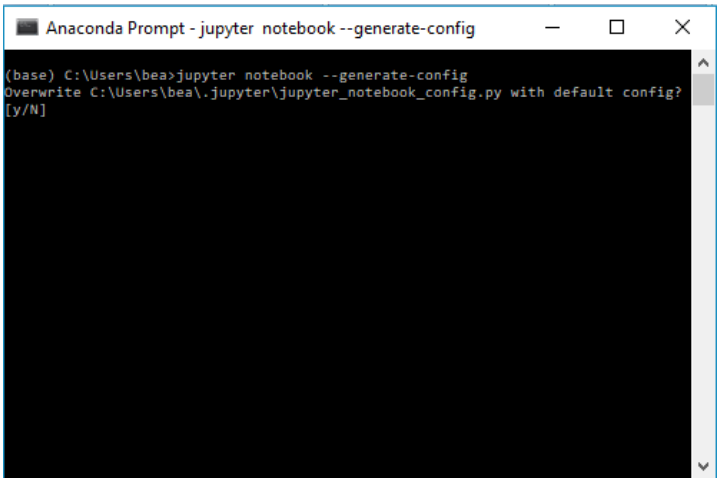


Figura 96: Jupyter Notebook –generate-config

Por otro lado, se sigue la siguiente ruta: *Disco local>>Usuarios>>bea>>.jupyter* y se abre el archivo "Jupyter_notebook_config" con el programa Notepad++.

Una vez se tiene el archivo indicado abierto, hay que realizar una serie de modificaciones. La primera es descomentar la línea 68 (quitándole el símbolo #).

```
66  #
67  #  This can be set to false to prevent changing password from the UI/A
68  #c.NotebookApp.allow_password_change = True
```

Figura 97: Línea 68 original

```
66  #
67  #  This can be set to false to prevent changing password from the UI/A
68  c.NotebookApp.allow_password_change = True
```

Figura 98: Línea 68 modificada

El segundo es descomentar la línea 82 y cambiar el lado derecho de la igualdad por *True*.

```
79  #
80  #  Local IP addresses (such as 127.0.0.1 and ::1) are allowed as local,
81  #  with hostnames configured in local_hostnames.
82  #c.NotebookApp.allow_remote_access = False
83
84  ## Whether to allow the user to run the notebook as root.
85  #c.NotebookApp.allow_root = False
```

Figura 99: Línea 82 original

```
79  #
80  #  Local IP addresses (such as 127.0.0.1 and ::1) are allowed as local,
81  #  with hostnames configured in local_hostnames.
82  c.NotebookApp.allow_remote_access = True
83
84  ## Whether to allow the user to run the notebook as root.
85  #c.NotebookApp.allow_root = False
```

Figura 100: Línea 82 modificada

Por último, habrá que descomentar la línea 200 y sustituir "localhost" por "*".

```
198
199   ## The IP address the notebook server will listen on.
200   #c.NotebookApp.ip = 'localhost'
201
```

Figura 101: Línea 200 original

```
198
199   ## The IP address the notebook server will listen on.
200   c.NotebookApp.ip = '*'
201
```

Figura 102: Línea 200 modificada

Una vez se ha modificado el documento, se vuelve a abrir anaconda prompt y se cambia la contraseña. Para hacerlo habrá que escribir el comando "Jupyter Notebook password" e insertar la nueva contraseña. Hay que tener en cuenta que al escribirla no aparece en pantalla, aunque al introducirla se pide una confirmación.

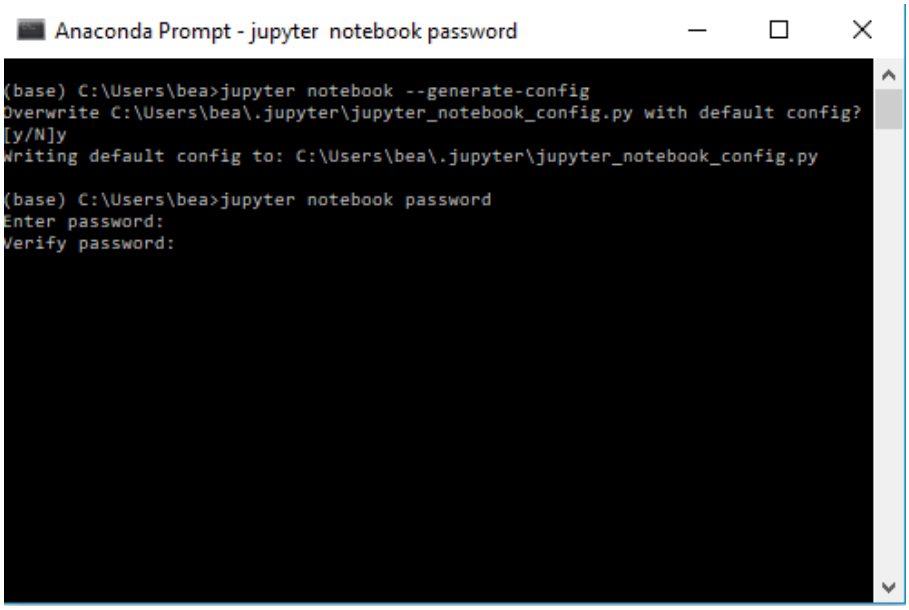


Figura 103: Cambio de contraseña

Por último, se ejecuta Jupyter Notebook simplemente escribiéndolo en Anaconda Prompt.

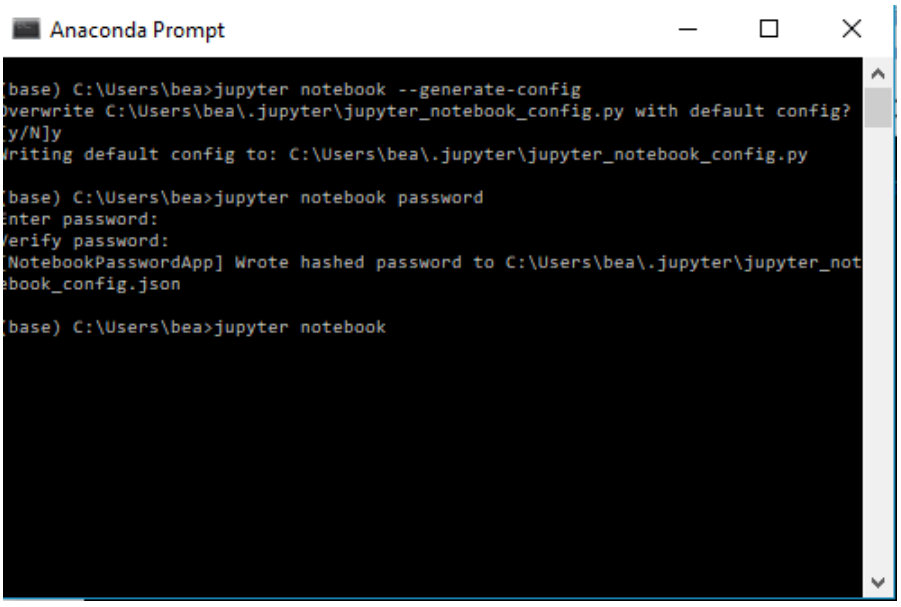


Figura 104: Ejecución de Jupyter Notebook desde Anaconda prompt

Tras este paso se abrirá el programa en el navegador de internet y pedirá la contraseña que se había indicado previamente.

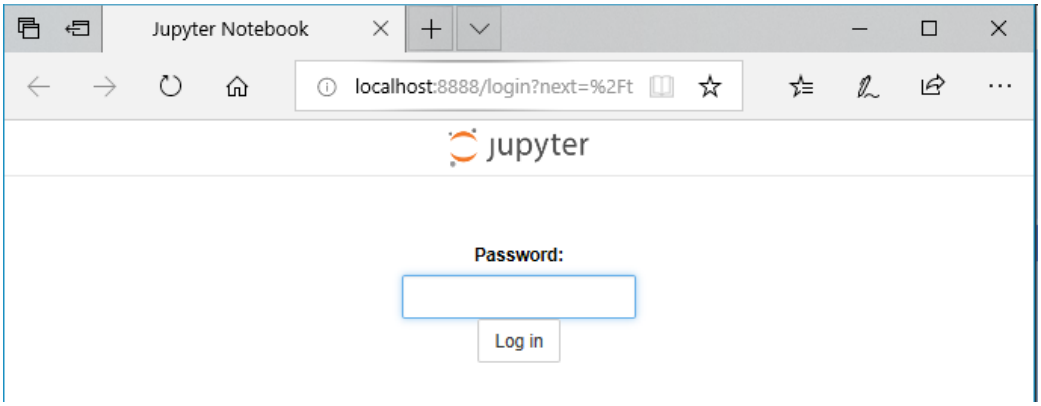


Figura 105: Petición de contraseña

Al introducir la contraseña, que en este caso es "*Notebook*", aparece la pantalla principal de Jupyter Notebook.

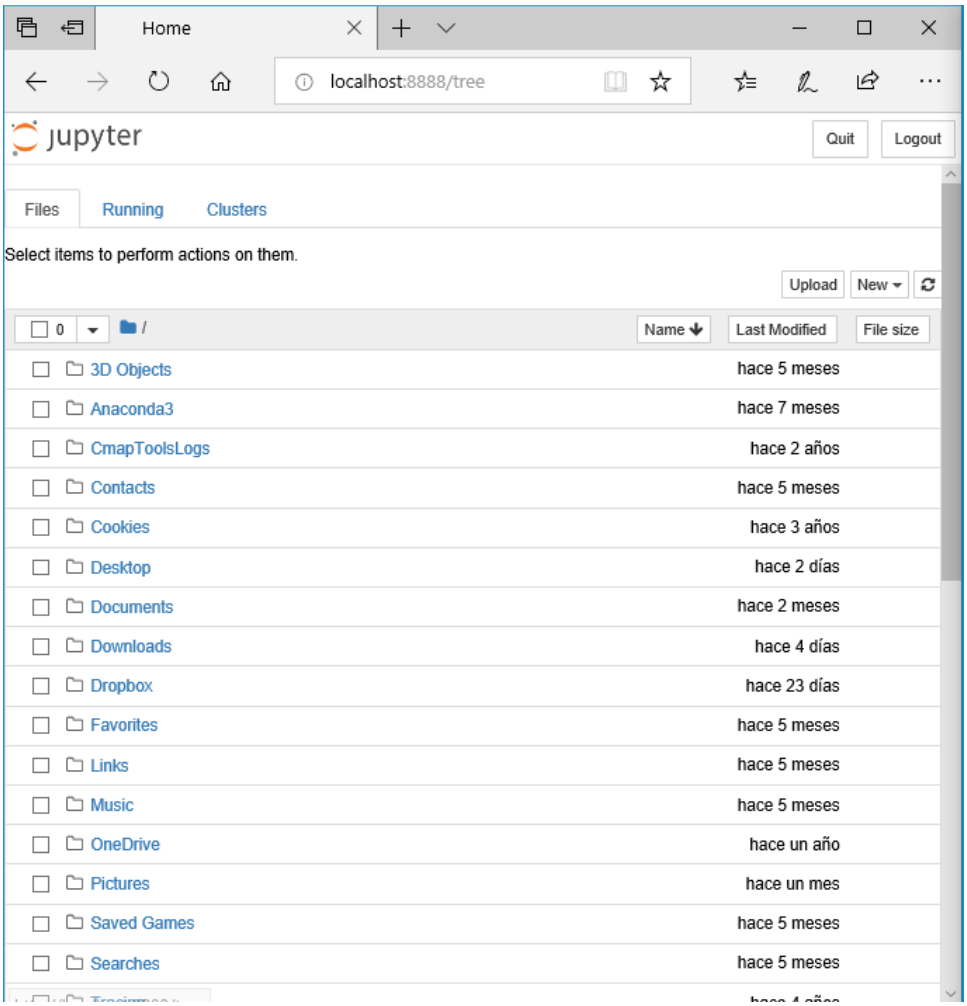


Figura 106: Home Jupyter Notebook

Ahora ya pueden simularse los programas diseñados en el teléfono móvil. No se debe olvidar que el dispositivo debe estar conectado a la misma red *Wifi* que el ordenador. Lo último que debe ser tenido en cuenta es que se debe cambiar "localhost" de la dirección URL por la dirección ip. Esta dirección se puede conocer a través de la consola del sistema escribiendo el comando "ipconfig". Aparecen varios códigos, pero el que se necesita es la dirección IPv4.

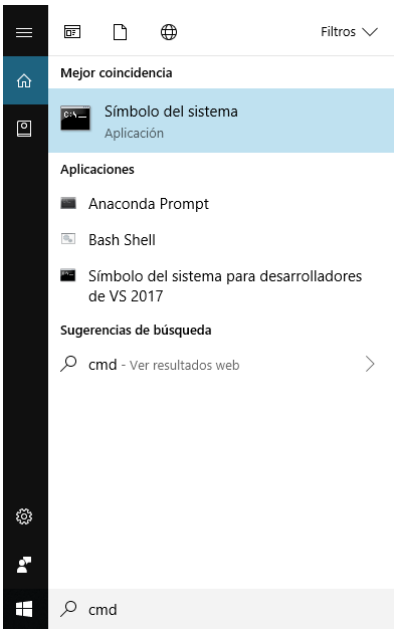


Figura 107: Consola de Windows

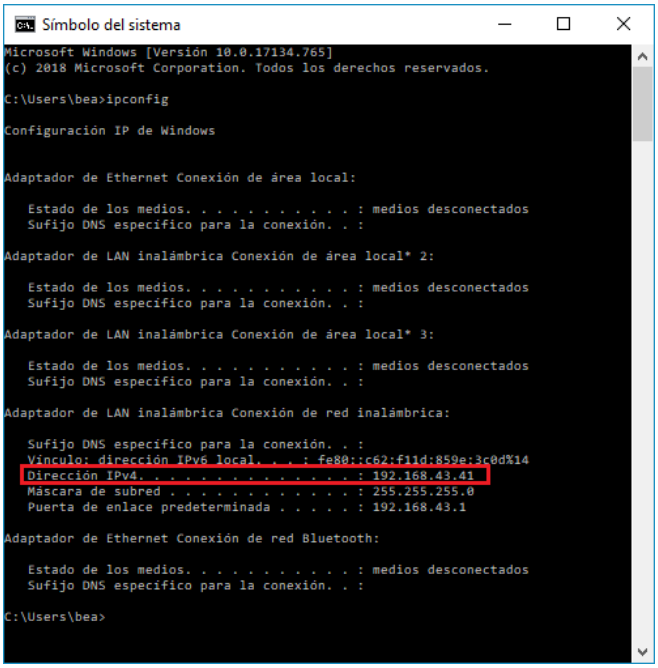


Figura 108: Dirección IP

En este caso, la dirección que habría que introducir en el navegador del teléfono móvil es: 192.168.43.41:8888/tree.

Habiendo seguido el procedimiento se podrá ejecutar cualquier *Notebook* en el teléfono del mismo modo que si fuese un computador. Cabe destacar que los pasos donde se realizan cambios en *anaconda prompt* o en archivos del sistema solo se

hacen una vez en cada computador. Es decir, que una vez realizados ya solo será necesario introducir la URL en el móvil y funcionará.

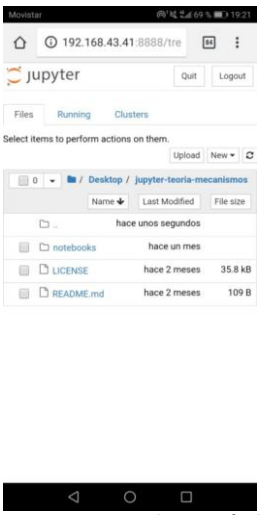


Figura 109: Desktop móvil

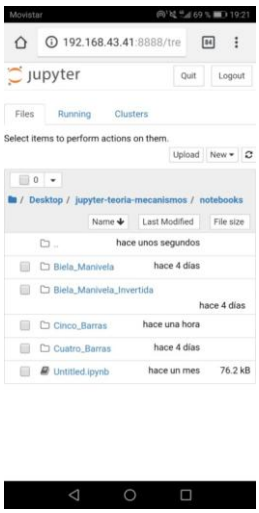


Figura 110: Notebooks móvil

Figura 111: 4B móvil

Figura 112: BMI móvil

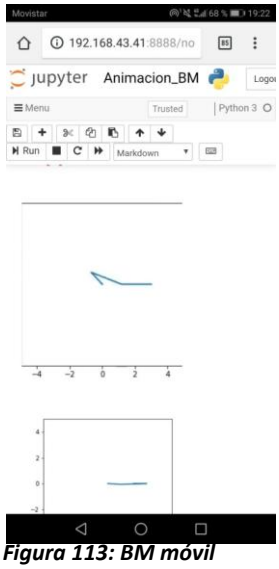


Figura 113: BM móvil

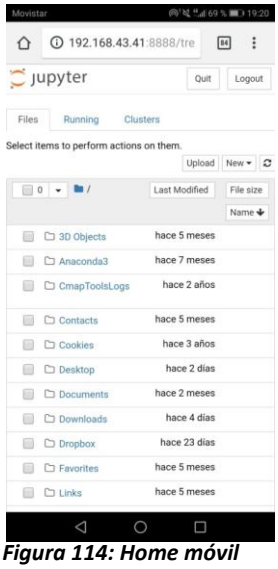


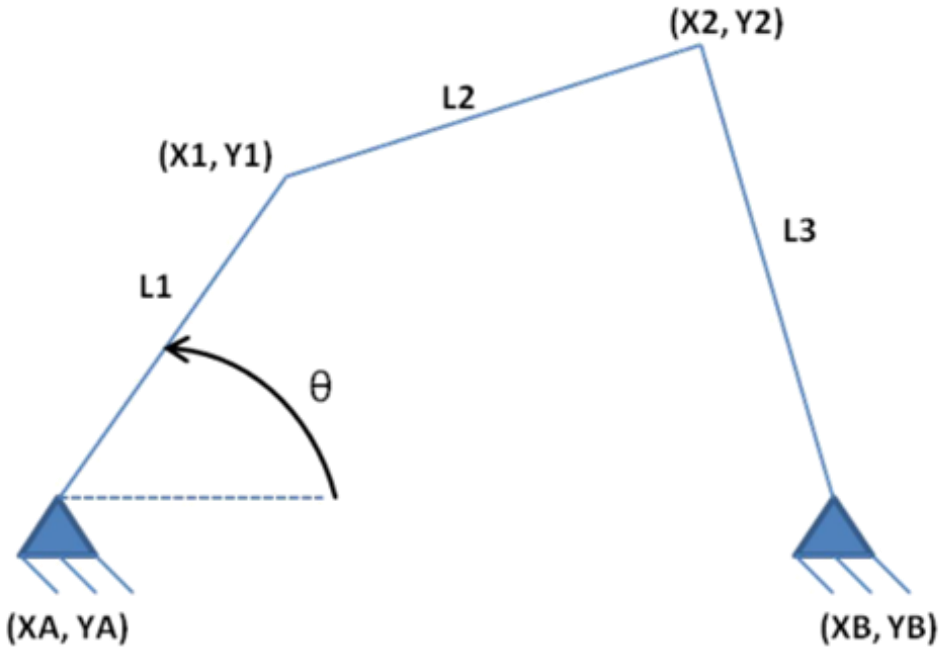
Figura 114: Home móvil

MECANISMO CUATRO BARRAS

PROBLEMA POSICIÓN

PASO 1: MODELADO DEL MECANISMO

El mecanismo de partida es el conocido como cuatro barras. Podemos ver todas las variables en la figura.



PASO 2: GRADOS DE LIBERTAD

Los grados de libertad se calculan:

$G = 3·(n-1) - 2·PI - PII$

Siendo:

PI -> Numero de pares binarios de un grado de libertad

PII -> Número de pares binarios de dos grados de libertad. En este caso tendríamos:
\begin{eqnarray} n &=& 4 \\ P_{I} &=& 4 \\ P_{II} &=& 0 \end{eqnarray}

Por lo tanto:

$G = 3·(4-1) - 2·4 - 0 = 1$

PASO 3: DEFINICIÓN DEL VECTOR q

El vector $\mathbf{q}$ de dimensión $N \times 1$ contiene las coordenadas dependientes del mecanismo. Es decir, contiene las coordenadas que no se mantienen fijas ya que varían con el tiempo.

Lo hemos modelado empleando las 5 coordenadas:
\begin{equation} \mathbf{q} = \begin{bmatrix} x_1 \\ y_1 \\ x_2 \\ y_2 \\ \theta \end{bmatrix} \end{equation}

PASO 4: IMPLEMENTACIÓN EN PYTHON

Al igual que en otros entornos de programación, necesitamos añadir librerías que contengan las funciones que vamos a utilizar. Esto es necesario hacerlo al principio del código. Las que vamos a usar son las siguientes:

1. numpy -> Sirve para trabajar con arrays y matrices, ofreciendo una interfaz similar a los comandos en MATLAB.
2. math -> La utilizaremos para usar funciones matemáticas.

3. pprint -> "pretty print", su función es ayudar a depurar el código.
4. matplotlib.pyplot -> Es necesaria para dibujar gráficas.

In [1]:

```
#PASO 4

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

PASO 5: LECTURA DE DATOS

Los datos iniciales de los que partiremos para resolver este mecanismo mediante análisis cinemático por métodos numéricos son los parámetros constantes que definen el mecanismo, es decir, las variables que no cambian con el tiempo. En este caso serían las longitudes de las barras y las posiciones de los apoyos.

Además, como el mecanismo tiene un único grado de libertad, tenemos que escoger la **variable independiente** entre las componentes del vector q. En este caso hemos escogido el ángulo, por lo que también será un dato de partida.

1. Longitudes de las barras: $L_1, ~ L_2 ~ y ~ L_3$.
2. Posición de los dos apoyos: $X_A, ~ Y_A, ~ X_B ~ e ~ Y_B$.
3. Ángulo que forma la primera barra respecto a la horizontal en radianes: $θ(t=0)$.

Una vez tengamos esos datos, definiremos una posición inicial.

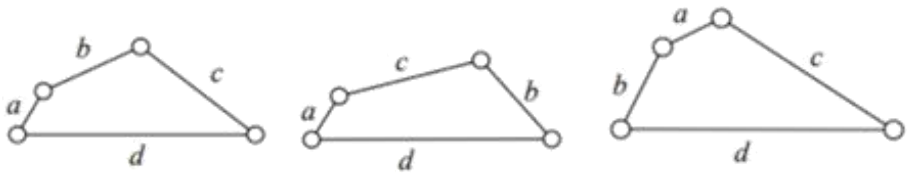
In [2]:

```
#PASO 5

print ('MECANISMO DE CUATRO BARRAS')
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["L3"] = float (input ('Introduce longitud L3:'))
meca["theta"] = float (input ('Introduce angulo inicial theta:')) #En rad
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 L2:2
Introduce longitud L3:2.5
Introduce angulo inicial theta:0
Introduce coordenada en x del punto B:3
q: [[0.1]
 [1. ]
 [1. ]
 [0.2]
 [0. ]]
```

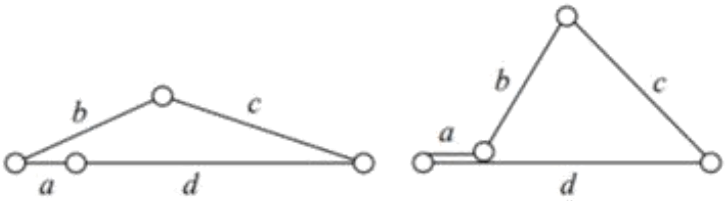
PASO 6: LEY DE GRASHOFF

El mecanismo cuatro barras es un tipo de cuadrilátero articulado. Partimos de las cuatro longitudes de los elementos de la cadena cinemática a, b, c y d, siendo $a<b<c<d$. Con estas barras hay tres configuraciones posibles de cadenas cinemáticas distintas.



(a) Configuración I (b) Configuración II (c) Configuración III

Para estudiar las condiciones geométricas que deben darse para que una barra sea manivela o balancín, hay que saber cuándo una barra puede dar vueltas completas con respecto a otra. Si demostramos que una barra puede dar vueltas completas con respecto a otra en una de las tres configuraciones de la figura, también podrá hacerlo en las otras dos.



(a) Barras extendidas (b) Barras opuestas

Para que la barra a pueda dar vueltas completas con respecto a la barra d, deben poder alcanzarse las dos posiciones representadas en la figura anterior, en las cuales a y d aparecen alineadas. Estas configuraciones son conocidas como posiciones límite.

Podemos escribir:

$\text{a)} \rightarrow b+c > a+d $

$\text{b)} \rightarrow d-a > c-b $

La primera ecuación expresa una propiedad de los triángulos que establece que la suma de las longitudes de dos de sus lados es mayor que la del tercero. Por su parte, la segunda ecuación expresa la propiedad que establece que la diferencia de las longitudes de dos lados es menor que la del tercero.

La segunda ecuación también se puede escribir como:

$\text{b)} \rightarrow d+b > c+a $

Para obtener estas expresiones partimos de la hipótesis de que a puede dar vueltas completas con respecto a d.
Podríamos seguir el mismo procedimiento con el resto de combinaciones entre barras para obtener una tabla de desigualdades como la siguiente:

Barras	*Desigualdad* I	*Desigualdad* II
a y d	$b+c>a+d\ (iii)$	$b+d>a+c\ (i)$
a y c	$b+d>a+c\ (i)$	$b+c>a+d\ (iii)$
a y b	$c+d>a+b\ (i)$	$b+c>a+d\ (iii)$
b y d	$a+c>b+d\ (ii)$	$a+d>b+c\ (iv)$
b y c	$a+d>b+c\ (iv)$	$a+c>b+d\ (ii)$
c y d	$a+b>c+d\ (ii)$	$a+d>b+c\ (iv)$

En la tabla apreciamos que hay solo cuatro tipos de desigualdades diferentes: i, ii, iii y iv.

i -> Se sumplen automáticamente, como por ejemplo que la suma de las dos barras más largas es mayor que la suma de las dos barras más cortas.

ii -> Son imposibles, como que la suma de las dos barras más cortas es mayor que la suma de las dos más largas.

iii y iv -> Son opuesta la una a la otra y pueden cumplirse o no. La desigualdad iii es conocida como desigualdad de Grashoff.

Podemos sacar las siguientes conclusiones de la tabla:

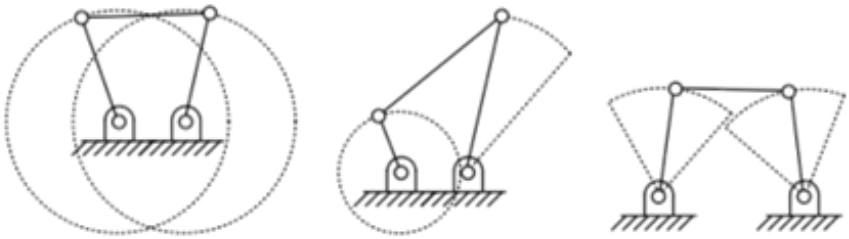
1. La única barra que puede dar vueltas completas con respecto a las demás es la pequeña. Esto se puede verificar comprobando en la tabla que cuando la barra a no aparece en la primera columna siempre aparece una condición imposible.
2. Si la barra pequeña puede dar vueltas completas con respecto de otra barra, también podrá hacerlo con respecto al resto. Es

decir, para que la barra a pueda dar vueltas completas tiene que cumplir la desigualdad de Grashoff y una vez tengamos esto se satisfacen también las condiciones necesarias para que de vueltas completas con respecto a b, c y d.

Si se cumple la desigualdad de Grashoff hay varios movimientos posibles para el cuadrilátero:

1. Doble manivela si el elemento a es fijo. Las barras contiguas al fijo dan vueltas completas, por lo que son manivelas.
2. Manivela-balancín si a es contiguo al elemento fijo.
3. Doble balancín si el elemento a es opuesto al fijo.

En caso de no cumplirse la desigualdad de Grashoff, el cuadrilátero será de doble balancín.



(a) Doble manivela (b) Manivela-balancín (c) Doble balancín

In [3]:

```
# PASO 6

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.")
```

```
[3.0, 2.5, 2.0, 1.0]
```

PASO 7: MATRIZ DE RESTRICCIONES $\Phi(q)$

Este vector agrupa las ecuaciones de restricción y será de dimensión $mx1$.

Estas ecuaciones podrían definirse empleando diferentes tipos de coordenadas: independientes, dependientes, relativas dependientes, de punto de referencia y naturales. Estas últimas son las que vamos a usar nosotros.

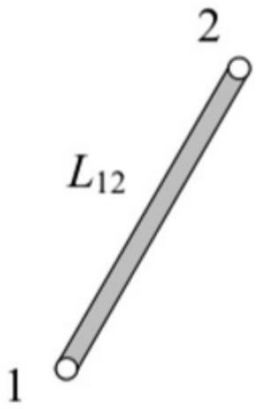
Para coordenadas naturales en el plano es necesario seguir un procedimiento:

1. Cada sólido debe tener al menos 2 puntos.
2. Cada par de rotación debe tener 1 punto.
3. Cada par prismático debe tener 3 puntos alineados.

4. Se pueden añadir tantos puntos adicionales como fuera necesario.
5. De los puntos mencionados, los fijos no entran en el vector q.

Para la formación de la matriz de restricciones, tenemos que tener en cuenta que hay restricciones de sólido rígido y de pares cinemáticos.

En este caso necesitamos:



Una única restricción:

$$(x_2 - x_1)^2 + (y_2 - y_1)^2 - L_{12}^2 = 0$$

Esta restricción tiene la función de imponer que los puntos de los extremos de cada barra permanezcan a una distancia constante. Debemos aplicarla a las 3 barras del mecanismo. Es decir, tendríamos:

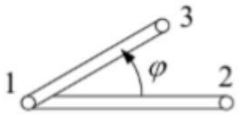
$\text{Barra 1} \rightarrow (X_1 - X_A)^2 + (Y_1 - Y_A)^2 - L_1^2 = 0$

$\text{Barra 2} \rightarrow (X_1 - X_1) ^2 + (Y_2 - Y_1)^2 - L_2^2 = 0$

$\text{Barra 3} \rightarrow (X_B - X_2) ^2 + (Y_B - Y_2) ^2 - L_3^2 = 0$

Además, como el vector de coordenadas dependientes tiene 5 componentes tenemos que añadir una ecuación de restricción para el ángulo.

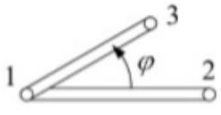
Esta última ecuación depende de si el ángulo en cuestión es demasiado pequeño. Esto se debe a que cuando un ángulo tiende a 0, su seno también lo hace, por lo que para esos casos utilizaríamos la restricción del coseno. En cambio, cuando el ángulo tiende más a 90°, es el coseno el que se aproxima a 0, por lo que en esos casos la restricción a utilizar sería la del seno.



Se añade ϕ al vector $\mathbf{q}$.
Usando el producto escalar: $\rightarrow\vec{12} \cdot \vec{13} = L_{12}L_{13}\cos\phi$.
Usando el producto vectorial: $\rightarrow\vec{12} \times \vec{13} = L_{12}L_{13}\sin\phi$.

Podemos añadir una de las dos restricciones a $\mathbf{\Phi}(\mathbf{q})$, pero mejor ambas (evitar singularidades):

$$\begin{aligned}(x_2 - x_1)(x_3 - x_1) + (y_2 - y_1)(y_3 - y_1) &= L_{12}L_{13}\cos\phi \\ (x_2 - x_1)(y_3 - y_1) - (y_2 - y_1)(x_3 - x_1) &= L_{12}L_{13}\sin\phi\end{aligned}$$



Cuando el ángulo ϕ (orientación *absoluta*) es relativo a tierra (12 es tierra), las ecuaciones se simplifican:

$$
\begin{aligned}
\cancel{L_{12}}(x_3 - x_1) + \cancel{(y_2 - y_1)}^{0}(y_3 - y_1) &= \cancel{L_{12}} L_{13}\cos\phi \\
\cancel{L_{12}}(y_3 - y_1) - \cancel{(y_2 - y_1)}^{0}(x_3 - x_1) &= \cancel{L_{12}} L_{13}\sin\phi
\end{aligned}
$$

$$\downarrow$$

$$
\begin{aligned}
x_3 - x_1 &= L_{13}\cos\phi \\
y_3 - y_1 &= L_{13}\sin\phi
\end{aligned}
$$

Para nuestro caso tendríamos:

$Si ~~ \cos (θ) < \frac{1}{\sqrt{2}} \rightarrow (X_1 - X_A) - L_1 · \cos(θ)$

$Si ~~ \cos (θ) > \frac{1}{\sqrt{2}} \rightarrow (Y_1 - Y_A) - L_1 · \sin (θ)$

La matriz quedaría:

1. $Si ~~ \cos (θ) < \frac{1}{\sqrt{2}}$ \begin{equation} \mathbf{\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_3^2 \\ X_1-L_1\cos(θ)\\ \end{bmatrix} \end{equation}
2. $Si ~~ \cos (θ) > \frac{1}{\sqrt{2}}$

\begin{equation} \mathbf{\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_3^2 \\ Y_1-L_1\sin(θ)\\ \end{bmatrix} \end{equation}

In [4]:

```
#PASO 7

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] = (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 8: Matriz jacobiana Φ_q

Esta matriz de dimensiones mxn está compuesta por las derivadas parciales de las ecuaciones de restricción con respecto al vector de coordeandas dependientes.

Por ejemplo tendríamos:

$\Phi_q(0,0) ~ = ~ Derivada ~ de ~ \Phi (0) ~ respecto ~ a ~ X1. $ $\Phi_q(0,1) ~ = ~ Derivada ~ de ~ \Phi (0) ~ respecto ~ a ~ Y1. $ $\Phi_q(1,0) ~ = ~ Derivada ~ de ~ \Phi (1) ~ respecto ~ a ~ X1. $

Tendríamos que construir la matriz elemento a elemento de esta manera.

Para la ecuación del ángulo hay que tener en cuenta que el jacobiano también tomará dos valores. Los posibles son:

1. $Si ~~ \cos (θ) < \frac{1}{\sqrt{2}}$

   $\Phi_q(3,0)~ = ~ 1$ $\Phi_q(3,4) ~ = ~ L1 · \sin(θ)$

1. $Si ~~ \cos (θ) > \frac{1}{\sqrt{2}}$

$\Phi_q(3,1) = 1$ $\Phi_q(3,4) = -L1 · \cos(θ)$

Es decir, tenemos dos posibles matrices jacobianas:

1. $Si ~~ \cos (θ) < \frac{1}{\sqrt{2}}$

\begin{equation} \mathbf{\Phi_q} = \begin{bmatrix} 2X_1 & 2Y_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) & 2Y_2 & 0\\ 1 & 0 & 0 & 0 & L1\sin(θ)\\ \end{bmatrix} \end{equation}

1. $Si ~~ \cos (θ) > \frac{1}{\sqrt{2}}$

\begin{equation} \mathbf{\Phi_q} = \begin{bmatrix} 2X_1 & 2Y_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) & 2Y_2 & 0 \\ 0 & 1 & 0 & 0 & -L1\cos(θ) \\ \end{bmatrix} \end{equation}

In [5]:

```
#PASO 8

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 9: RESOLUCIÓN DEL PROBLEMA POSICIÓN

El objetivo es obtener los valores de las coordenadas dependientes del vector q.

Para resolverlo partimos de la ecuación:

$\Phi(q+Δq) = \Phi + \Phi_q · Δq = 0$

De donde despejamos:

$\Phi_q · Δq = -\Phi$

Esta ecuación se convertiría en $Ax=b$, siendo A el jacobiano y b Φ. Sin embargo, no se pueden dividir matrices de esa manera, por lo que tenemos que multiplicar a ambos lados de la igualdad por la izquierda por A^{-1}, es decir, por la inversa del jacobiano $Φq^{-1}$:

$ \underbrace{ \Phi q^-1 · Φq}_{= \mathbf{I_n}} · Δq = Φq^{-1} ·-Φ$

Por el lado izquierdo al multiplicar el jacobiano por su inversa toma el valor la unidad, por lo que quedaría:

$Δq = \Phi_q^{-1} · -Φ$

Y para terminar, tendríamos que el nuevo valor de q sería:

$q = q + Δq$

Hay que repetir este proceso hasta que el vector Φ se aproxime a 0, lo que indicaría la validez del vector q calculado. Sin embargo, hay datos iniciales para los que el mecanismo no converge. Por ejemplo, si decimos que $X_A=0, ~ X_A=0, ~ X_B=50, ~ Y_B=0 ~ y ~ L_1=1, ~ L_2=2, ~ L_3=3$, no existe tal posición. Por ello, tenemos que poner un límite de iteraciones, como por ejemplo 100, y si llega a dicho límite tendremos que el mecanismo no converge.

Para poder operar con las matrices de ese modo, es necesario que las dimensiones sean correctas. En este caso tendríamos que añadir una fila a la matriz Φ y a Φ_q. Como el mecanismo tiene un grado de libertad tenemos que asignar un valor conocido a una de las variables dependientes, como ya se explicó anteriormente. Esto se traduce en que para la matriz de restricciones $\Phi(4)=0$, ya que su valor permanece invariable. Por otro lado para la matriz Φ_q tendríamos que para la variable conocido su valor sería uno y el resto 0. Es decir:

$\text{Φq(4)} \rightarrow (0, 0, 0, 0, 1)$

Otra forma de medir el error en el cálculo de q es calculando el módulo del vector $Δq$. Tiene que ser lo más próximo a 0.

Por último, podemos saber si el mecanismo convergerá calculando el rango de la matriz jacobiana. Si su rango es igual al número de coordenadas dependientes, convergerá.

In [6]:

```
#PASO 9

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 s
olucion

        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)
```

```
q=
 array([[0.1],
       [1. ],
       [1. ],
       [0.2],
       [0. ]])
Phi=
array([[ 0.01],
       [-2.55],
       [-2.21],
       [ 1.  ],
       [ 0.  ]])
jacob=
array([[ 0.2,  2.  ,  0. ,  0. ,  0. ],
       [-1.8,  1.6,   1.8, -1.6,  0. ],
       [ 0. ,  0. , -4. ,  0.4,  0. ],
       [ 0. ,  1.  ,  0. ,  0. , -1. ],
       [ 0. ,  0.  ,  0. ,  0. ,  1. ]])
rango=5

error iter1=
19.19195767662633
q=
array([[ 10.05     ],
       [  0.       ],
       [ -1.17605634],
       [-16.03556338],
       [  0.]])
Phi=
array([[100.0025   ],
       [379.16363383],
       [268.32873946],
       [  0.       ],
       [  0.       ]])
jacob=
array([[ 20.1       ,   0.        ,   0.        ,   0.        ,
          0.        ],
       [ 22.45211268,  32.07112676, -22.45211268, -32.07112676,
          0.        ],
       [  0.        ,   0.        ,  -8.35211268, -32.07112676,
          0.        ],
        [  0.       ,    1.        ,   0.        ,   0.       ,
         -1.        ],
        [  0.       ,    0.        ,   0.        ,   0.       ,
          1.        ]])
rango=5

error iter2=
9.748195126631266
q=
array([[ 5.07475124e+00],
       [-9.28043661e-17],
       [-1.23775501e+00],
       [-7.65281912e+00],
       [ 0.00000000e+00]])
Phi=
array([[ 2.47531002e+01],
       [ 9.44133757e+01],
       [ 7.02742080e+01],
       [-9.28043661e-17],
       [ 0.00000000e+00]])
jacob=
array([[ 1.01495025e+01, -1.85608732e-16, 0.00000000e+00,
         0.00000000e+00, 0.00000000e+00],
       [ 1.26250125e+01, 1.53056382e+01, -1.26250125e+01,
        -1.53056382e+01, 0.00000000e+00],
       [ 0.00000000e+00, 0.00000000e+00, -8.47551002e+00,
        -1.53056382e+01, 0.00000000e+00],
       [ 0.00000000e+00, 1.00000000e+00, 0.00000000e+00,
         0.00000000e+00, -1.00000000e+00],
       [ 0.00000000e+00, 0.00000000e+00, 0.00000000e+00,
         0.00000000e+00, 1.00000000e+00]])
rango=5

error iter3=
6.207811152264088
q=
```

```
array([[ 2.63590262e+00],
       [ 7.73393444e-16],
       [-2.84067650e+00],
       [-2.17380649e+00],
       [ 0.00000000e+00]])
Phi=
 array([[5.94798262e+00],
       [3.07183536e+01],
       [3.25889367e+01],
       [7.73393444e-16],
       [0.00000000e+00]])
jacob=
array([[ 5.27180524e+00, 1.54678689e-15, 0.00000000e+00,
         0.00000000e+00, 0.00000000e+00],
       [ 1.09531582e+01, 4.34761299e+00, -1.09531582e+01,
        -4.34761299e+00, 0.00000000e+00],
       [ 0.00000000e+00, 0.00000000e+00, -1.16813530e+01,
        -4.34761299e+00, 0.00000000e+00],
       [ 0.00000000e+00, 1.00000000e+00, 0.00000000e+00,
         0.00000000e+00, -1.00000000e+00],
       [ 0.00000000e+00, 0.00000000e+00, 0.00000000e+00,
         0.00000000e+00, 1.00000000e+00]])
rango=5

error iter4=
49.07572536662256
q=
array([[ 1.50763965e+00],
       [ 7.56463733e-15],
       [ 1.66989119e+01],
       [-4.71777877e+01],
       [ 0.00000000e+00]])
Phi=
 array([[1.27297732e+00],
       [2.45251841e+03],
       [2.40715384e+03],
       [7.56463733e-15],
       [0.00000000e+00]])
jacob=
array([[ 3.01527930e+00, 1.51292747e-14, 0.00000000e+00,
         0.00000000e+00, 0.00000000e+00],
       [-3.03825445e+01, 9.43555755e+01, 3.03825445e+01,
        -9.43555755e+01, 0.00000000e+00],
       [ 0.00000000e+00, 0.00000000e+00, 2.73978238e+01,
        -9.43555755e+01, 0.00000000e+00],
       [ 0.00000000e+00, 1.00000000e+00, 0.00000000e+00,
         0.00000000e+00, -1.00000000e+00],
       [ 0.00000000e+00, 0.00000000e+00, 0.00000000e+00,
         0.00000000e+00, 1.00000000e+00]])
rango=5

error iter5=
27.826714440264766
q=
array([[ 1.08546406e+00],
       [-1.54079305e-15],
       [-2.79749660e+00],
       [-2.73274009e+01],
       [ 0.00000000e+00]])
Phi=
array([[ 1.78232229e-01],
       [ 7.57864221e+02],
       [ 7.74147804e+02],
       [-1.54079305e-15],
       [ 0.00000000e+00]])
jacob=
array([[ 2.17092812e+00, -3.08158611e-15, 0.00000000e+00,
          0.00000000e+00, 0.00000000e+00],
       [  7.76592133e+00, 5.46548017e+01, -7.76592133e+00,
         -5.46548017e+01, 0.00000000e+00],
       [  0.00000000e+00, 0.00000000e+00, -1.15949932e+01,
         -5.46548017e+01, 0.00000000e+00],
       [ 0.00000000e+00, 1.00000000e+00, 0.00000000e+00,
         0.00000000e+00, -1.00000000e+00],
       [ 0.00000000e+00, 0.00000000e+00, 0.00000000e+00,
         0.00000000e+00, 1.00000000e+00]])
rango=5
```

```
error iter6=
13.945738845678857
q=
 array([[ 1.00336451e+00],
       [-3.78141889e-16],
       [ 1.62163225e+00],
       [-1.41006026e+01],
       [ 0.00000000e+00]])
Phi=
array([[ 6.74033666e-03],
       [ 1.95209249e+02],
       [ 1.94476892e+02],
       [-3.78141889e-16],
       [ 0.00000000e+00]])
jacob=
array([[ 2.00672902e+00, -7.56283778e-16,  0.00000000e+00,
         0.00000000e+00,  0.00000000e+00],
       [-1.23653548e+00,  2.82012052e+01,  1.23653548e+00,
        -2.82012052e+01,  0.00000000e+00],
       [ 0.00000000e+00,  0.00000000e+00, -2.75673550e+00,
        -2.82012052e+01,  0.00000000e+00],
       [ 0.00000000e+00,  1.00000000e+00,  0.00000000e+00,
         0.00000000e+00, -1.00000000e+00],
       [ 0.00000000e+00, 0.00000000e+00, 0.00000000e+00,
         0.00000000e+00, 1.00000000e+00]])
rango=5

error iter7=
6.916538585815401
q=
array([[ 1.00000564e+00],
       [ 5.83419759e-16],
       [ 1.43719430e+00],
       [-7.18652441e+00],
       [ 0.00000000e+00]])
Phi=
array([[1.12819902e-05],
       [4.78372670e+01],
       [4.78384947e+01],
       [5.83419759e-16],
       [0.00000000e+00]])
jacob=
array([[ 2.00001128e+00, 1.16683952e-15, 0.00000000e+00,
         0.00000000e+00, 0.00000000e+00],
       [-8.74377323e-01, 1.43730488e+01, 8.74377323e-01,
        -1.43730488e+01, 0.00000000e+00],
       [ 0.00000000e+00, 0.00000000e+00, -3.12561140e+00,
        -1.43730488e+01, 0.00000000e+00],
       [ 0.00000000e+00, 1.00000000e+00, 0.00000000e+00,
         0.00000000e+00, -1.00000000e+00],
       [ 0.00000000e+00, 0.00000000e+00, 0.00000000e+00,
         0.00000000e+00, 1.00000000e+00]])
rango=5

error iter8=
3.328280592449615
q=
array([[ 1.00000000e+00],
       [ 4.06484118e-16],
       [ 1.43750000e+00],
       [-3.85824383e+00],
       [ 0.00000000e+00]])
Phi=
array([[3.18207682e-11],
       [1.10774517e+01],
       [1.10774517e+01],
       [4.06484118e-16],
       [0.00000000e+00]])
jacob=
array([[ 2.00000000e+00,  8.12968236e-16,  0.00000000e+00,
         0.00000000e+00, 0.00000000e+00],
       [-8.75000002e-01, 7.71648766e+00,  8.75000002e-01,
        -7.71648766e+00, 0.00000000e+00],
       [ 0.00000000e+00, 0.00000000e+00, -3.12500000e+00,
        -7.71648766e+00, 0.00000000e+00],
       [ 0.00000000e+00, 1.00000000e+00,  0.00000000e+00,
```

```
         0.00000000e+00, -1.00000000e+00],
       [ 0.00000000e+00, 0.00000000e+00, 0.00000000e+00,
         0.00000000e+00, 1.00000000e+00]])
rango=5

error iter9=
1.435556200143426
q=
 array([[ 1.00000000e+00],
       [-1.82585357e-16],
       [ 1.43750000e+00],
       [-2.42268763e+00],
       [ 0.00000000e+00]])
Phi=
array([[ 0.00000000e+00],
       [ 2.06082160e+00],
       [ 2.06082160e+00],
       [-1.82585357e-16],
       [ 0.00000000e+00]])
jacob=
array([[ 2.00000000e+00, -3.65170714e-16, 0.00000000e+00,
         0.00000000e+00, 0.00000000e+00],
       [-8.75000000e-01, 4.84537526e+00, 8.75000000e-01,
        -4.84537526e+00, 0.00000000e+00],
       [ 0.00000000e+00, 0.00000000e+00, -3.12500000e+00,
        -4.84537526e+00, 0.00000000e+00],
       [ 0.00000000e+00,  1.00000000e+00,  0.00000000e+00,
         0.00000000e+00, -1.00000000e+00],
       [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         0.00000000e+00,  1.00000000e+00]])
rango=5

error iter10=
0.42531723405813104
q=
array([[ 1.0000000e+00],
       [ 3.7379877e-17],
       [ 1.4375000e+00],
       [-1.9973704e+00],
       [ 0.0000000e+00]])
Phi=
 array([[0.0000000e+00],
       [1.8089475e-01],
       [1.8089475e-01],
       [3.7379877e-17],
        [0.0000000e+00]])
jacob=
array([[ 2.00000000e+00,  7.47597539e-17,  0.00000000e+00,
          0.00000000e+00, 0.00000000e+00],
        [-8.75000000e-01, 3.99474079e+00,  8.75000000e-01,
         -3.99474079e+00, 0.00000000e+00],
       [ 0.00000000e+00, 0.00000000e+00, -3.12500000e+00,
        -3.99474079e+00, 0.00000000e+00],
        [0.00000000e+00,  1.00000000e+00,  0.00000000e+00,
         0.00000000e+00, -1.00000000e+00],
       [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         0.00000000e+00,  1.00000000e+00]])
rango=5

error iter11=
0.04528322586829045
q=
array([[ 1.00000000e+00],
       [ 1.06994449e-17],
       [ 1.43750000e+00],
       [-1.95208717e+00],
       [ 0.00000000e+00]])
Phi=
 array([[0.00000000e+00],
       [2.05057055e-03],
       [2.05057055e-03],
       [1.06994449e-17],
       [0.00000000e+00]])
jacob=
array([[ 2.00000000e+00, 2.13988898e-17, 0.00000000e+00,
         0.00000000e+00, 0.00000000e+00],
          [-8.75000000e-01,              3.90417434e+00,
                                          8.75000000e-01,
```

```
        -3.90417434e+00,  0.00000000e+00],
       [ 0.00000000e+00, 0.00000000e+00, -3.12500000e+00,
        -3.90417434e+00, 0.00000000e+00],
       [ 0.00000000e+00, 1.00000000e+00, 0.00000000e+00,
         0.00000000e+00, -1.00000000e+00],
       [ 0.00000000e+00, 0.00000000e+00, 0.00000000e+00,
         0.00000000e+00, 1.00000000e+00]])
rango=5

error iter12=
0.0005252251477831435
q=
array([[ 1.00000000e+00],
       [ 3.56374430e-20],
       [ 1.43750000e+00],
       [-1.95156195e+00],
       [ 0.00000000e+00]])
Phi=
 array([[0.00000000e+00],
       [2.75861455e-07],
       [2.75861455e-07],
       [3.56374430e-20],
       [0.00000000e+00]])
jacob=
array([[ 2.00000000e+00, 7.12748859e-20, 0.00000000e+00,
         0.00000000e+00, 0.00000000e+00],
       [-8.75000000e-01, 3.90312389e+00, 8.75000000e-01,
        -3.90312389e+00, 0.00000000e+00],
       [ 0.00000000e+00, 0.00000000e+00, -3.12500000e+00,
        -3.90312389e+00, 0.00000000e+00],
       [ 0.00000000e+00, 1.00000000e+00, 0.00000000e+00,
         0.00000000e+00, -1.00000000e+00],
       [ 0.00000000e+00, 0.00000000e+00, 0.00000000e+00,
         0.00000000e+00, 1.00000000e+00]])
rango=5

error iter13=
7.067709434954129e-08
q=
array([[ 1.00000000e+00],
       [ 1.24070869e-23],
       [ 1.43750000e+00],
       [-1.95156187e+00],
       [ 0.00000000e+00]])
Phi=
 array([[0.00000000e+00],
       [5.32907052e-15],
       [5.32907052e-15],
       [1.24070869e-23],
       [0.00000000e+00]])
jacob=
array([[ 2.00000000e+00, 2.48141739e-23, 0.00000000e+00,
         0.00000000e+00, 0.00000000e+00],
       [-8.75000000e-01, 3.90312375e+00, 8.75000000e-01,
        -3.90312375e+00, 0.00000000e+00],
       [ 0.00000000e+00, 0.00000000e+00, -3.12500000e+00,
        -3.90312375e+00, 0.00000000e+00],
       [ 0.00000000e+00, 1.00000000e+00, 0.00000000e+00,
         0.00000000e+00, -1.00000000e+00],
       [ 0.00000000e+00, 0.00000000e+00, 0.00000000e+00,
         0.00000000e+00, 1.00000000e+00]])
rango=5

error iter14=
1.3653347480295311e-15
num iters:14
```

PASO 10: Dibujar el mecanismo

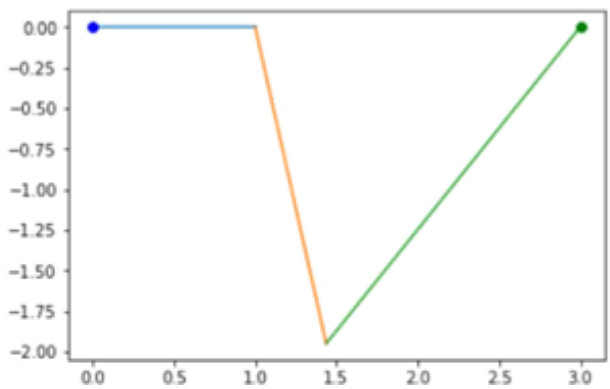
Para dibujar el mecanismo, definimos un cuadro de dibujo con los ejes de la misma dimensión. Seguidamente, dibujamos cada barra por separado. Para dibujar cada barra tendríamos que indicar las posiciones inicial y final, yendo por un lado las coordenadas en el eje X y por otro las coordenadas en el eje Y. Es decir, sería por ejemplo:

$Barra ~ 1 \rightarrow ([X_A, X_1], [Y_A, Y_1])$

In [7]:

```
#PASO 10


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')


    plt.plot ([meca["XA"], X1], [meca["YA"], Y1])    #[pos inicial (x1,x2), pos final (y1,y2)]
    plt.plot ([X1, X2], [Y1, Y2])
    plt.plot ([X2, meca["XB"]], [Y2, meca ["YB"]])

    plt.plot(meca["XA"], meca["YA"], 'bo')
    plt.plot(meca["XB"], meca["YB"], 'go')

    plt.show()#block=False)
    return

dibuja_mecanismo(q,meca)
```

```
q=
array([[ 1.00000000e+00],
       [ 9.12892351e-32],
       [ 1.43750000e+00],
       [-1.95156187e+00],
       [ 0.00000000e+00]])
Phi=
 array([[ 0.00000000e+00],
       [-4.44089210e-16],
       [ 0.00000000e+00],
       [ 9.12892351e-32],
       [ 0.00000000e+00]])
jacob=
array([[ 2.00000000e+00, 1.82578470e-31, 0.00000000e+00,
         0.00000000e+00, 0.00000000e+00],
       [-8.75000000e-01, 3.90312375e+00, 8.75000000e-01,
        -3.90312375e+00, 0.00000000e+00],
       [ 0.00000000e+00, 0.00000000e+00, -3.12500000e+00,
        -3.90312375e+00, 0.00000000e+00],
       [ 0.00000000e+00, 1.00000000e+00, 0.00000000e+00,
         0.00000000e+00, -1.00000000e+00],
       [ 0.00000000e+00, 0.00000000e+00, 0.00000000e+00,
         0.00000000e+00, 1.00000000e+00]])
rango=5

error iter1=
1.42222370596101e-16
num iters:1
```



MECANISMO CUATRO BARRAS

PROBLEMAS VELOCIDAD Y ACELERACIÓN

PASO 1: MATRIZ JACOBIANA

Para resolver el problema velocidad, necesitamos otra vez la matriz jacobiana. El método de construcción aparece detallado en el notebook Problema_Posición_4B, por lo que para este caso copiaremos el código de los pasos realizados para poder conseguirla.

In [1]:

```
#PASO 1
import numpy as np
import math as math
#import array as arr
import pprint
import matplotlib.pyplot as plt
import matplotlib.animation as animation
import scipy.integrate as integrate
import os
from time import sleep
%matplotlib inline


print ('MECANISMO DE CUATRO BARRAS')
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["L3"] = float (input ('Introduce longitud L3:'))
meca["theta"] = float (input ('Introduce angulo inicial theta:')) #En rad
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))

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.")
# JACOBIANO
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*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


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] = (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



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))
    return q

q = resuelve_prob_posicion
(q,meca) J = jacob_Phiq(q,meca)
print  ("Jacob=")
pprint.pprint(J)
```

```
MECANISMO DE CUATRO BARRAS
==========================
Introduce longitud L1:1
Introduce longitud L2:2
Introduce longitud L3:2.5
Introduce angulo inicial theta:0
Introduce coordenada en x del punto B:3
[3.0, 2.5, 2.0, 1.0]
Jacob=
array([[ 2.00000000e+00, 1.82578470e-31, 0.00000000e+00,
         0.00000000e+00, 0.00000000e+00],
       [-8.75000000e-01, 3.90312375e+00, 8.75000000e-01,
        -3.90312375e+00, 0.00000000e+00],
       [ 0.00000000e+00, 0.00000000e+00, -3.12500000e+00,
        -3.90312375e+00, 0.00000000e+00],
       [ 0.00000000e+00, 1.00000000e+00, 0.00000000e+00,
         0.00000000e+00, -1.00000000e+00],
       [ 0.00000000e+00, 0.00000000e+00, 0.00000000e+00,
         0.00000000e+00, 1.00000000e+00]])
```

PASO 2: PROBLEMA VELOCIDAD

Consiste en determinar las velocidades de todas las variables del mecanismo una vez sabemos su posición q y la velocidad de los grados de libertad.

Partimos de la ecuación:

$\Phi_q =0 $

Derivando se obtiene:

$\Phi_q \dot{q} + \Phi_t = 0$

Siendo $\dot{q}$ el vector velocidad, Φ_q el jacobiano y Φ_t la derivada parcial de las ecuaciones de restricción respecto al tiempo. Para las ecuaciones de sólido rígido el valor de esta derivada es 0. Solo tendría un valor no nulo la correspondiente al ángulo, que en ese caso tendría la velocidad que nosotros le indiquemos.

De este modo la expresión quedaría:

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

Este sistema de ecuaciones tiene infinitas soluciones y por tanto hay que ampliar añadiendo un dato conocido de velocidad, lo que se hace añadiendo una fila a la matriz de coeficientes del lado izquierdo y un dato a la columna del lado derecho de la ecuación por cada grado de libertad.

De esta forma llegamos a un sistema de ecuaciones lineal matricial de la forma:

$\mathbf{A} \mathbf{x} = \mathbf{b}$

Habría que multiplicar en ambas partes de la igualdad por la A invertida en el lado izquierdo, del mismo modo que se hizo en el problema de posición. De esta manera quedaría:

$x = -A^{-1} b$

In [6]:

```
#PASO 2

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
```

Out[6]:

```
array([[-9.12892351e-32],
       [ 1.00000000e+00],
       [-9.75780937e-01],
       [ 7.81250000e-01],
       [ 1.00000000e+00]])
```

PASO 3: PROBLEMA ACELERACIÓN

El problema aceleración trata de determinar las aceleraciones de todas las variables del mecanismo, conociendo la posiciónq, la velocidad $\dot{q}$ y las aceleraciones de los grados de libertad.

Partimos la ecuación que se obtiene tras derivar la ecuación inicial para el problema de velocidad, es decir:

$\Phi_q \dot{q} + \Phi_t = 0$

Se deriva por segunda vez:

$\dot{\Phi}_q \dot{q} + \Phi_q \ddot{q} + \dot{\Phi}_t = 0$

Despejamos $\Phi_q \ddot{q}$:

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

Siendo Φ_q el jacobiano, $\ddot{q}$ el vector aceleración, $\dot{q}$ el vector velocidad, $\dot{\Phi}_q$ la derivada del jacobiano respecto al tiempo y $\dot{\Phi}_t$ es la derivada de las ecuaciones de restricción con respecto al tiempo, cuyo valor es nulo. Es decir, tendríamos:

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

Del mismo modo que en el problema velocidad, llamando b al conjunto formado por $\dot{\Phi}_q \dot{q}$ llegamos a un sistema de ecuaciones lineal matricial:

$ \mathbf{A} \mathbf{x} = \mathbf{b}$

Y despejando la x:

$\mathbf{x} = \mathbf{-A}^{-1} \mathbf{b}$

El vector velocidad será:
\begin{equation} \mathbf{q_p} = \begin{bmatrix} \dot{X}_1 \\ \dot{Y}_1\\ \dot{X}_2 \\ \dot{Y}_2 \\ \dot{\theta}\\ \end{bmatrix} \end{equation}

Por otro lado, para calcular la derivada del jacobiano solo tenemos en cuenta las filas que hacen referencia a las ecuaciones de las coordenadas dependientes, ya que la última que añadimos para poder realizar los cálculos era adicional. Teniendo en cuenta esto, la derivada del jacobiano sería:

1. $Si ~~ \cos (\theta) < \frac{1}{\sqrt{2}}$

\begin{equation} \mathbf{\dot{\Phi}_q} = \begin{bmatrix} 2\dot{X}_1 & 2\dot{Y}_1 & 0 & 0 & 0 \\ -2(\dot{X}_2-\dot{X}_1) & -2(\dot{Y}_2-\dot{Y}_1) & 2(\dot{X}_2-\dot{X}_1) & 2(\dot{Y}_2-\dot{Y}_1) & 0\\ 0 & 0 & 2\dot{X}_2 & 2\dot{Y}_2 & 0 \\ 0 & 0 & 0 & 0 & \theta^2L_1\cos(\theta)\\ \end{bmatrix} \end{equation}

1. $Si ~~ \cos (\theta) > \frac{1}{\sqrt{2}}$

\begin{equation} \mathbf{\dot{\Phi}_q} = \begin{bmatrix} 2\dot{X}_1 & 2\dot{Y}_1 & 0 & 0 & 0 \\ -2(\dot{X}_2-\dot{X}_1) & -2(\dot{Y}_2-\dot{Y}_1) & 2(\dot{X}_2-\dot{X}_1) & 2(\dot{Y}_2-\dot{Y}_1) & 0\\ 0 & 0 & 2\dot{X}_2 & 2\dot{Y}_2 & 0 \\ 0 & 0 & 0 & 0 & \theta^2L_1\sin(\theta)\\ \end{bmatrix} \end{equation}

Como ya tenemos $\dot{\Phi}_q$ y $\dot{q}$, podemos calcular b. La última fila que añadimos es el valor de la aceleración angular, dato que sabemos de antemano.

1. $Si ~~ \cos (\theta) < \frac{1}{\sqrt{2}}$

\begin{equation} \mathbf{b} = \begin{bmatrix} 2\dot{X}_1^2 + 2\dot{Y}_1^2\\ -2\dot{X}_1(\dot{X}_2-\dot{X}_1) - 2\dot{Y}_1(\dot{Y}_2-\dot{Y}_2) + 2\dot{X}_2(\dot{X}_2-\dot{X}_1) - 2\dot{Y}_2·(\dot{Y}_2-\dot{Y}_1)\\ 2\dot{X}_2^2 + 2\dot{Y}_2^2 \\ \dot{\theta}^2·L_1·\cos(\theta)\\ 1\\ \end{bmatrix} \end{equation}

1. $Si ~~ \cos (\theta) > \frac{1}{\sqrt{2}}$

\begin{equation} \mathbf{b} = \begin{bmatrix} 2\dot{X}_1^2 + 2\dot{Y}_1^2\\ -2\dot{X}_1(\dot{X}_2-\dot{X}_1) - 2\dot{Y}_1(\dot{Y}_2-\dot{Y}_2) + 2\dot{X}_2(\dot{X}_2-\dot{X}_1) - 2\dot{Y}_2(\dot{Y}_2-\dot{Y}_1)\\ 2\dot{X}_2^2 + 2\dot{Y}_2^2 \\ \dot{\theta}^2L_1\sin(\theta)\\ 1\\ \end{bmatrix} \end{equation}

In [7]:

```
#PASO 3
# añadir***  q,qp,qpp,meca
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))

    b[4] = 1 #Aceleracion conocida
    qpp = np.linalg.solve(-jacob_Phiq(q,meca),b)
    #print ("qpp=")
    #pprint.pprint(qpp)

    return qpp

qpp=np.zeros((5,1))
qpp=resuelve_prob_aceleracion(q,qp,qpp,meca)
qpp
```

Out[7]:

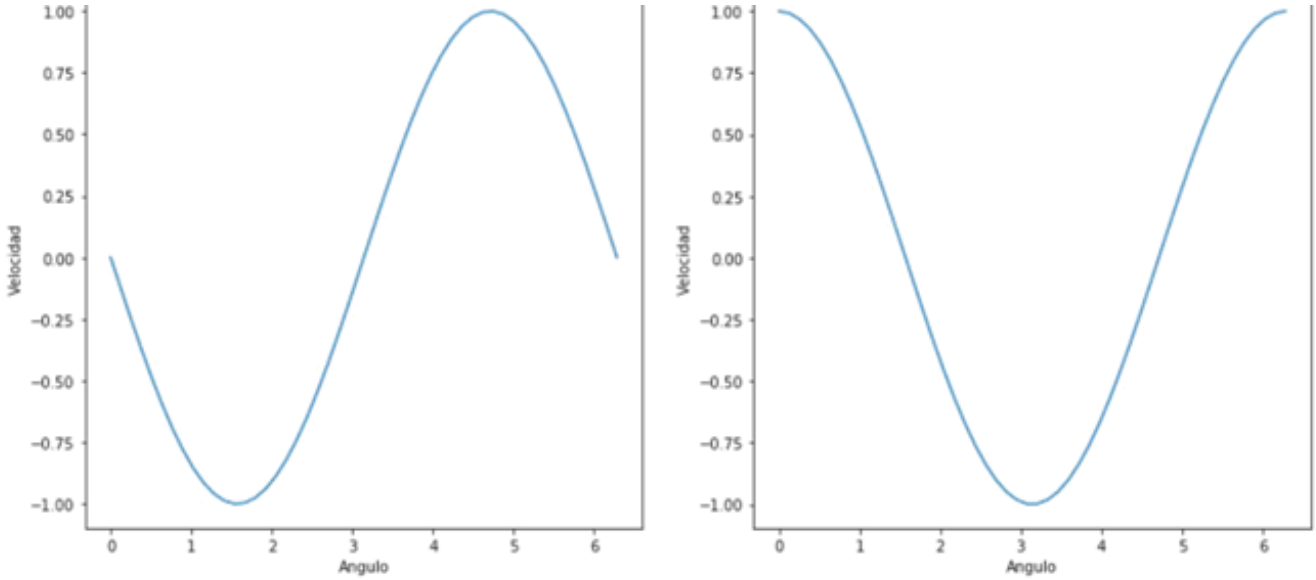
```
array([[-1.        ],
       [-1.        ],
       [ 1.03828094],
       [-0.03064928],
       [-1.]])
```

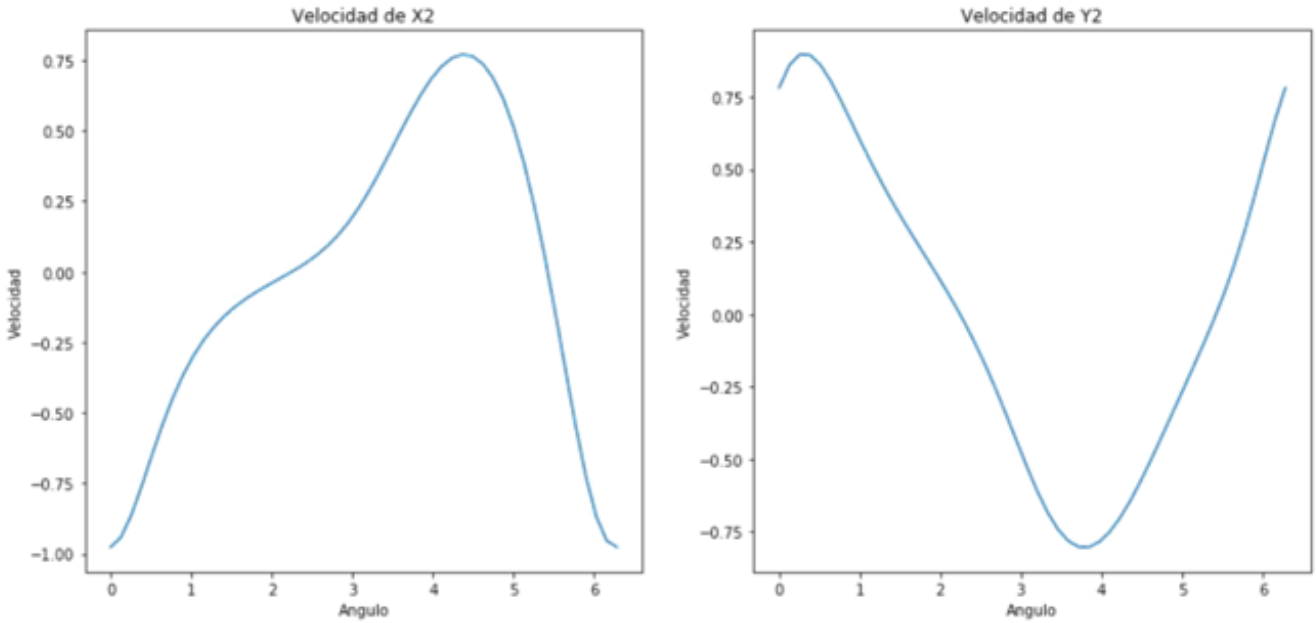
PASO 4: GRÁFICAS DE VELOCIDADES

Vamos a representar por separado la gráfica de la velocidad en cada coordenada $(X_1, ~ Y_1, ~ X_2 ~ e ~ Y_2)$.

In [18]:

```
#PASO 4: GRÁFICAS DE VELOCIDADES
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))
        #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)
```





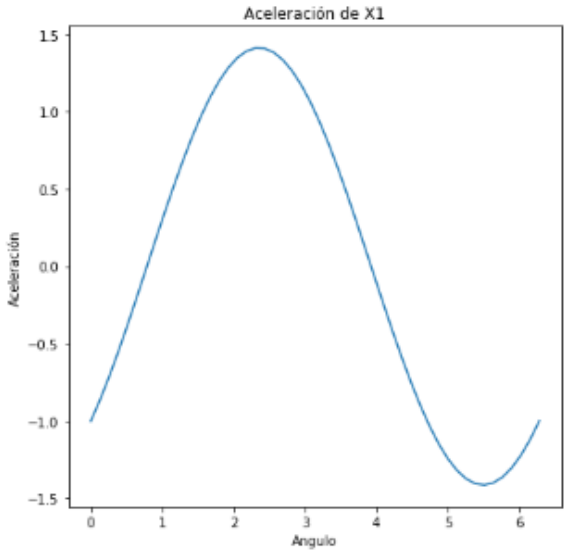
PASO 5: GRÁFICAS ACELERACIONES

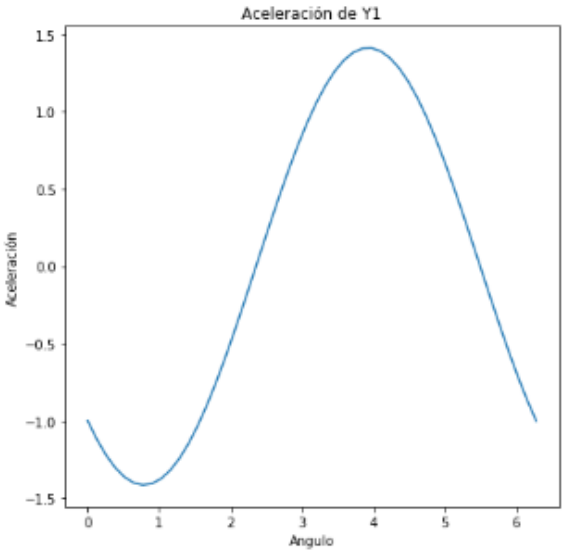
Haremos el mismo procedimiento que para la velocidad, representando en celdas separadas la aceleración de cada coordenada.

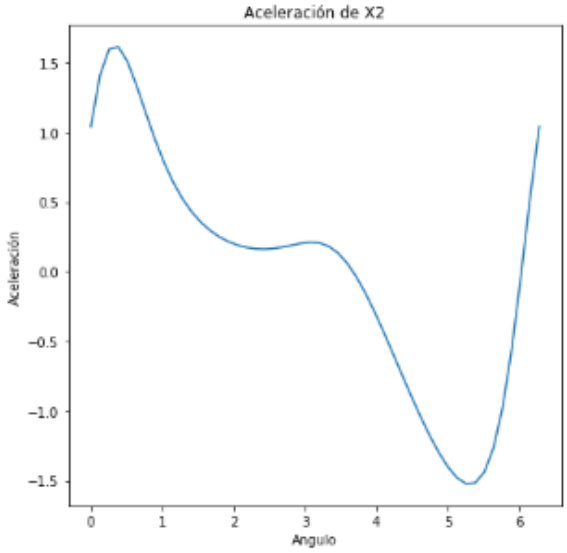
In [19]:

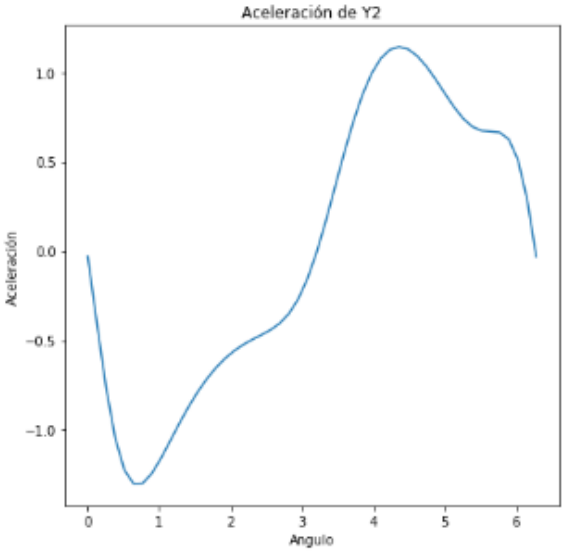
```
#PASO 5: GRÁFICAS ACELERACIONES

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))


    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 ('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)
```


Aceleración de X1
Aceleración
Angulo


Aceleración de Y1
Aceleración
Angulo


Aceleración de X2
Aceleración
Angulo


Aceleración de Y2
Aceleración
Angulo

MECANISMO CUATRO BARRAS

ANIMACIÓN

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 mpanim
import scipy.integrate as integrate
import os
from time import sleep
%matplotlib inline

print ('MECANISMO DE CUATRO BARRAS')
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["L3"] = float (input ('Introduce longitud L3:'))
meca["theta"] = float (input ('Introduce angulo inicial theta:')) #En rad
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))

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.")

# JACOBIANO
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


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] = (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



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)

        J = jacob_Phiq(q,meca)


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

    return q

q=resuelve_prob_posicion(q,meca)


def dibuja_mecanismo(q, meca):


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


    plt.axis('equal')


    plt.plot ([meca["XA"], X1], [meca["YA"], Y1])    #[pos inicial (x1,x2), pos final (y1,y2)]
    plt.plot ([X1, X2], [Y1, Y2])
    plt.plot ([X2, meca["XB"]], [Y2, meca ["YB"]])

    plt.plot(meca["XA"], meca["YA"], 'bo')
    plt.plot(meca["XB"], meca["YB"], 'go')


    plt.show(block=False)
    return
```

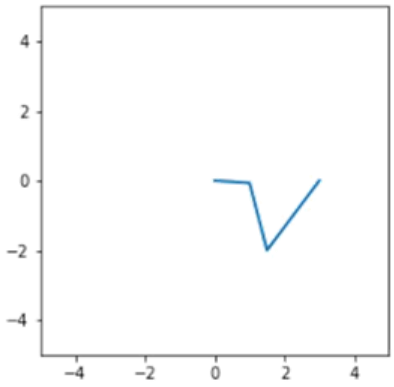
```
MECANISMO DE CUATRO BARRAS
==========================
Introduce longitud L1:1
Introduce longitud L2:2
Introduce longitud L3:2.5
Introduce angulo inicial theta:0
Introduce coordenada en x del punto B:3
q: [[0.1]
 [1. ]
 [1. ]
 [0.2]
 [0. ]]
[3.0, 2.5, 2.0, 1.0]
```

In [2]:

```
%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())
```

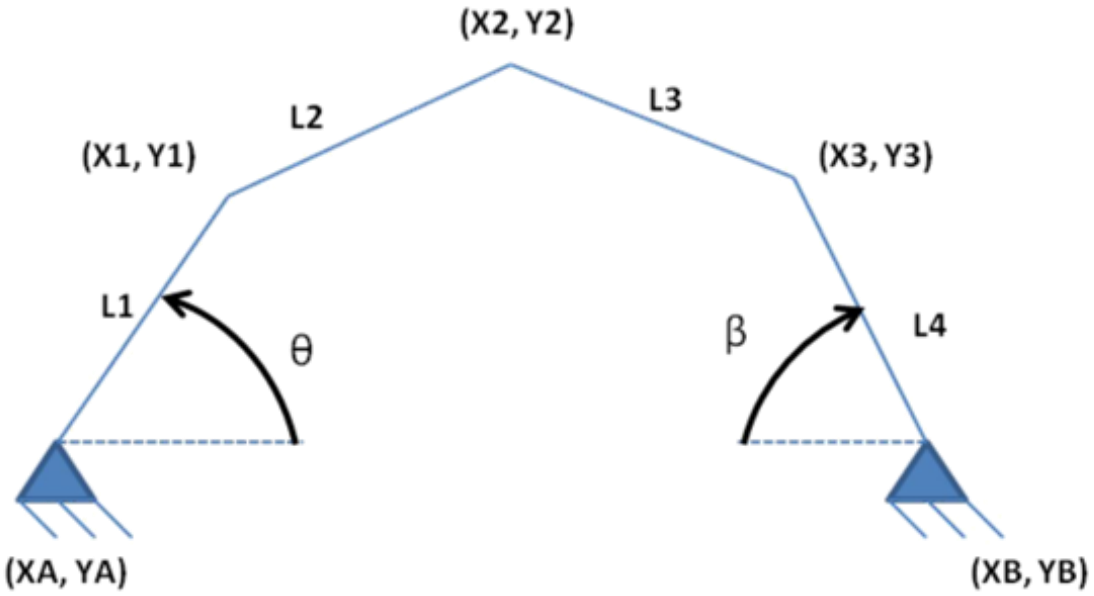
Out[2]:



MECANISMO CINCO BARRAS

PROBLEMA POSICIÓN

PASO 1: MODELADO DEL MECANISMO



PASO 2: GRADOS DE LIBERTAD

Los grados de libertad se calculan:

$G = 3·(n-1) - 2·PI - PII$

Siendo:

PI -> Numero de pares binarios de un grado de libertad

PII -> Número de pares binarios de dos grados de libertad. En este caso tendríamos:
\begin{eqnarray} n &=& 5 \\ P_{I} &=& 5 \\ P_{II} &=& 0 \end{eqnarray}

Por lo tanto:

$G = 3·(5-1) - 2·5 - 0 = 2$

PASO 3: DEFINICIÓN DEL VECTOR q

El vector $\mathbf{q}$ de dimensión $N \times 1$ contiene las coordenadas dependientes del mecanismo. Es decir, contiene las coordenadas que no se mantienen fijas ya que varían con el tiempo.

Lo hemos modelado enpleando las 8 coordenadas:
\begin{equation} \mathbf{q} = \begin{bmatrix} x_1 \\ y_1 \\ x_2 \\ y_2 \\ x_3 \\ y_3 \\ \theta \\ \beta \end{bmatrix} \end{equation}

PASO 4: IMPLEMENTACIÓN EN PYTHON

Al igual que en otros entornos de programación, necesitamos añadir librerías que contengan las funciones que vamos a utilizar. Esto es necesario hacerlo al principio del código. Las que vamos a usar son las siguientes:

1. numpy -> Sirve para trabajar con arrays y matrices, ofreciendo una interfaz similar a los comandos en MATLAB.
2. math -> La utilizaremos para usar funciones matemáticas.
3. pprint -> "pretty print", su función es ayudar a depurar el código.
4. matplotlib.pyplot -> Es necesaria para dibujar gráficas.

In [1]:

```
#PASO 4

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

PASO 5: LECTURA DE DATOS

Los datos iniciales de los que partiremos para resolver este mecanismo mediante análisis cinemático por métodos numéricos son los parámetros constantes que definen el mecanismo, es decir, las variables que no cambian con el tiempo. En este caso serían las longitudes de las barras y las posiciones de los apoyos.

Además, como el mecanismo tiene dos grados de libertad, tenemos que escoger las **variables independientes** entre las componentes del vector q. En este caso hemos escogido los ángulos, por lo que también serán un dato de partida.

1. Longitudes de las barras: $L_1, ~ L_2 ~ L_3 ~ y ~ L_4$.
2. Posición de los dos apoyos: $X_A, ~ Y_A, ~ X_B ~ e ~ Y_B$.
3. Ángulo que forma la primera barra respecto a la horizontal en radianes: $θ(t=0)$.
4. Ángulo que forma la cuarta barra respecto a la horizontal en radianes: $\beta(t=0)$

Una vez tengamos esos datos, definiremos una posición inicial.

In [2]:

```
# PASO 5

print ('MECANISMO DE CINCO BARRAS')
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["L3"] = float (input ('Introduce longitud L3:'))
meca["L4"] = float (input ('Introduce longitud L4: '))
meca["theta"] = float (input ('Introduce angulo inicial theta:')) #En rad
meca["beta"] = float (input('Introduce angulo inicial beta: ')) #En rad
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], [2.02], [0.1], [meca["theta"]], [meca["beta"]]])
print('q: ' + str(q))
```

```
MECANISMO DE CINCO BARRAS
============================
Introduce longitud L1:2
Introduce longitud L2:3.5
Introduce longitud L3:4
Introduce longitud L4: 1
Introduce angulo inicial theta:0.3
Introduce angulo inicial beta: 0.9
Introduce coordenada en x del punto B:2
q: [[0.1 ]
 [2. ]
 [1.  ]
 [0.2 ]
 [2.02]
 [0.1 ]
 [0.3 ]
 [0.9 ]]
```

PASO 6: MATRIZ DE RESTRICCIONES $\Phi(q)$

Este vector agrupa las ecuaciones de restricción y será de dimensión $mx1$.

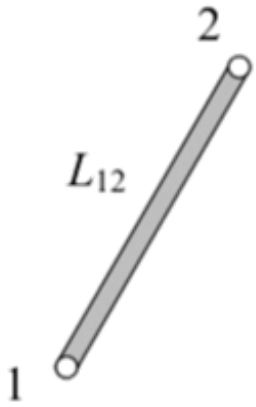
Estas ecuaciones podrían definirse empleando diferentes tipos de coordenadas: independientes, dependientes, relativas dependientes, de punto de referencia y naturales. Estas últimas son las que vamos a usar nosotros.

Para coordenadas naturales en el plano es necesario seguir un procedimiento:

1. Cada sólido debe tener al menos 2 puntos.
2. Cada par de rotación debe tener 1 punto.
3. Cada par prismático debe tener 3 puntos alineados.
4. Se pueden añadir tantos puntos adicionales como fuera necesario.
5. De los puntos mencionados, los fijos no entran en el vector q.

Para la formación de la matriz de restricciones, tenemos que tener en cuenta que hay restricciones de sólido rígido y de pares cinemáticos.

En este caso necesitamos:



Una única restricción:

$$(x_2 - x_1)^2 + (y_2 - y_1)^2 - L_{12}^2 = 0$$

Esta restricción tiene la función de imponer que los puntos de los extremos de cada barra permanezcan a unan distancia constante. Debemos aplicarla a las 4 barras del mecanismo. Es decir, tendríamos:

$\text{Barra 1} \rightarrow (X_1 - X_A)^2 + (Y_1 - Y_A)^2 - L_1^2 = 0$

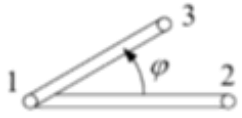
$\text{Barra 2} \rightarrow (X_1 - X_1) ^2 + (Y_2 - Y_1)^2 - L_2^2 = 0$

$\text{Barra 3} \rightarrow (X_3 - X_2) ^2 + (Y3 - Y_2) ^2 - L_3^2 = 0$

$\text{Barra 4} \rightarrow (X_B - X_3) ^2 + (YB - Y_3) ^2 - L_4^2 = 0$

Además, tenemos que añadir una ecuación de restricción para cada ángulo.

Estas últimas ecuaciones dependen de si el ángulo en cuestión es demasiado pequeño. Esto se debe a 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°, es el coseno el que se aproxima a 0, por lo que en esos casos la restricción a utilizar sería la del seno.

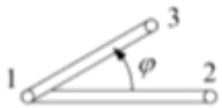


Se añade ϕ al vector $\mathbf{q}$.
Usando el producto escalar: $\rightarrow\vec{12} \cdot \vec{13} = L_{12}L_{13}\cos\phi$.
Usando el producto vectorial: $\rightarrow\vec{12} \times \vec{13} = L_{12}L_{13}\sin\phi$.

Podemos añadir una de las dos restricciones a $\mathbf{\Phi(q)}$, pero mejor ambas (evitar singularidades):

$$(x_2 - x_1)(x_3 - x_1) + (y_2 - y_1)(y_3 - y_1) \quad = \quad L_{12}L_{13}\cos\phi$$

$$(x_2 - x_1)(y_3 - y_1) - (y_2 - y_1)(x_3 - x_1) = L_{12}L_{13}\sin\phi$$



Cuando el ángulo ϕ (orientación *absoluta*) es relativo a tierra (12 es tierra), las ecuaciones se simplifican:

$$\begin{aligned} L_{12}(x_3 - x_1) + (y_2 - y_1)^{0}(y_3 - y_1) &= L_{12}L_{13}\cos\phi \\ L_{12}(y_3 - y_1) - (y_2 - y_1)^{0}(x_3 - x_1) &= L_{12}L_{13}\sin\phi \end{aligned}$$

$$\downarrow$$

$$\begin{aligned} x_3 - x_1 &= L_{13}\cos\phi \\ y_3 - y_1 &= L_{13}\sin\phi \end{aligned}$$

Para nuestro caso tendríamos:

$Si ~~ \cos (θ) < \frac{1}{\sqrt{2}} \rightarrow ~~ \Phi(3) = (X_1 - X_A) - L_1 · \cos(θ)$

$Si ~~ \cos (θ) > \frac{1}{\sqrt{2}} \rightarrow ~~ \Phi(3) = (Y_1 - Y_A) - L_1 · \sin (θ)$

$Si ~~ \cos (\beta) < \frac{1}{\sqrt{2}} \rightarrow ~~ \Phi(4) = (X_3 - X_B) - L_4 · \cos(\beta)$

$Si ~~ \cos (\beta) > \frac{1}{\sqrt{2}} \rightarrow ~~ \Phi(4) = (Y_3 - Y_B) - L_4 · \sin (\beta)$

La matriz quedaría:
\begin{equation} \mathbf{\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_3-X_2)^2 + (Y_3-Y_2)^2 - L_3^2 \\ (X_B-X_3)^2 + (Y_B-Y_3)^2 - L_4^2 \\ \Phi(3)\\ \Phi(4) \\ \end{bmatrix} \end{equation}

In [3]:

```
# PASO 6

def Phi (q,meca):
    #Inicializa a cero Phi
    Phi = np.zeros((8,1))

    #Extraer coordenadas
    X1 = q[0]
    Y1 = q[1]
    X2 = q[2]
    Y2 = q[3]
    X3 = q[4]
    Y3 = q[5]
    theta = q[6]
    beta = q[7]

    Phi[0] = X1**2 + Y1**2 - meca["L1"]**2
    Phi[1] = (X2-X1)**2 + (Y2-Y1)**2 -meca["L2"]**2
    Phi[2] = (X3-X2)**2 + (Y3-Y2)**2 - meca["L3"]**2
    Phi[3] = (meca["XB"] - X3)**2 + (meca["YB"] - Y3)**2 - meca["L4"]**2

    if (abs(math.cos(theta)) < 0.5 ):
        Phi[4] = X1-meca["L1"]*math.cos(theta)
    else:
        Phi[4] = Y1-meca["L1"]*math.sin(theta)

    if (abs(math.cos(beta)) < 0.5 ):
        Phi[5] = (X3-meca["XB"])-meca["L4"]*math.cos(beta)
    else:
        Phi[5] = Y3-meca["L4"]*math.sin(beta)

    return Phi
```

PASO 7: Matriz jacobiana Φ_q

Esta matriz de dimensiones mxn está compuesta por las derivadas parciales de las ecuaciones de restricción con respecto al vector de coordenadas dependientes.

Por ejemplo endríamos:

Φ_q(0,0) = Derivada de Φ(0) respecto a X1.

Φ_q(0,1) = Derivada de Φ (0) respecto a Y1.

Φ_q(1,0) = Derivada de Φ(1) respecto a X1.

Tendríamos que construir la matriz elemento a elemento de esta manera.

Para la ecuación del ángulo hay que tener en cuenta que el jacobiano también tomará dos valores. Los posibles son:

1. $Si ~~ \cos (θ) < \frac{1}{\sqrt{2}}$

   $\Phi_q(4,0) = 1 \quad $ $\Phi_q(4,6) = L_1 · \sin(θ)$

1. $Si ~~ \cos (θ) > \frac{1}{\sqrt{2}}$

   $\Phi_q(4,1) = 1//$ $\Phi_q(4,6) = -L_1 · \cos(θ)$

1. $Si ~~ \cos (\beta) < \frac{1}{\sqrt{2}}$

   $\Phi_q(5,4) = 1//$ $\Phi_q(5,7) = L_4 · \sin(\beta)$

1. $Si ~~ \cos (\beta) > \frac{1}{\sqrt{2}}$

   $\Phi_q(5,5) = 1//$ $\Phi_q(5,7) = -L_4 · \cos(\beta)$

La matriz jacobiana sería:
\begin{equation} \mathbf{\Phi_q} = \begin{bmatrix} 2X_1 & 2Y_1 & 0 & 0 & 0 & 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 & 0\\ 0 & 0 & -2(X_3-X_2) & -2(Y_3-Y_2) & 2(X_3-X_2) & 2(Y_3-Y_2) & 0 & 0\\ 0 & 0 & 0 & 0 & -2(X_B-X_3) & -2(Y_B-Y_3) & 0 & 0\\ \Phi_q(4,0) & \Phi_q(4,1) & 0 & 0 & 0 & 0 & \Phi_q(4,6) & \Phi_q(4,7)\\ 0 & 0 & 0 & 0 & \Phi_q(5,4) & \Phi_q(5,5) & \Phi_q(5,6) & \Phi_q(5,7)\\ \end{bmatrix} \end{equation}

In [4]:

```
# PASO 7

def jacob_Phiq(q,meca):
    #Inicializa a cero la matriz jacobiana
    Jacob = np.zeros((8,8))

    #Extraer coordenadas

    X1 = q[0]
    Y1 = q[1]
    X2 = q[2]
    Y2 = q[3]
    X3 = q[4]
    Y3 = q[5]
    theta = q[6]
    beta = q[7]

    #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*(X3-X2)
    Jacob[2,3] = -2*(Y3-Y2)
    Jacob[2,4] = 2*(X3-X2)
    Jacob[2,5] = 2*(Y3-Y2)
    Jacob[3,4] = -2*(meca["XB"]-X3)
    Jacob[3,5] = -2*(meca["YB"]-Y3)
```

```
    if (abs(math.cos(theta)) < 0.5 ):
        Jacob[4,6] = meca["L1"]*math.sin(theta)
        Jacob[4,0] = 1
    else:
        Jacob[4,6] = -meca["L1"]*math.cos(theta)
        Jacob[4,1] = 1


    if (abs(math.cos(beta)) < 0.5 ):
        Jacob[5,7] = meca["L4"]*math.sin(beta)
        Jacob[5,4] = 1
    else:
        Jacob[5,7] = -meca["L4"]*math.cos(beta)
        Jacob[5,5] = 1

    Jacob[6,6] = 1
    Jacob[7,7] = 1

    return Jacob
```

PASO 8: RESOLUCIÓN DEL PROBLEMA POSICIÓN

El objetivo es obtener los valores de las coordenadas dependientes del vector q.

Para resolverlo partimos de la ecuación:

$\Phi(q+Δq) = \Phi + \Phi_q · Δq = 0$

De donde despejamos:

$\Phi_q · Δq = -\Phi$

Esta ecuación se convertiría en $Ax=b$, siendo A el jacobiano y b Φ. Sin embargo, no se pueden dividir matrices de esa manera, por lo que tenemos que multiplicar a ambos lados de la igualdad por la izquierda por A^{-1}, es decir, por la inversa del jacobiano $Φq^{-1}$:

$ \underbrace{ \Phi q^-1 · Φq}_{= \mathbf{I_n}} · Δq = Φq^{-1} ·-Φ$

Por el lado izquierdo al multiplicar el jacobiano por su inversa toma el valor la unidad, por lo que quedaría:

$Δq = \Phi_q^{-1} · -Φ$

Y para terminar, tendríamos que el nuevo valor de q sería:

$q = q + Δq$

Hay que repetir este proceso hasta que el vector Φ se aproxime a 0, lo que indicaría la validez del vector q calculado. Sin embargo, hay datos iniciales para los que el mecanismo non converge. Por ejemplo, si decimos que $X_A=0, ~ X_A=0, ~ X_B=50, ~ Y_B=0 ~ y ~ L_1=1, ~ L_2=2, ~ L_3=3$, no existe tal posición. Por ello, tenemos que poner un límite de iteraciones, como por ejemplo 100, y si llega a dicho límite tendremos que el mecanismo no converge.

Para poder operar con las matricces de ese modo, es necesario que las dimensiones sean correctas. En este caso tendríamos que añadir dos filas a la matriz Φ y a Φ_q. Como el mecanismo tiene dos grados de libertad tenemos que asignar un valor conocido a dos de las variables dependientes, como ya se explicó anteriormente. Esto se traduce en que para la matriz de restricciones $\Phi(6)=0$ y $\Phi(7)=0$, ya que su valor permanece invariable. Por otro lado para la matriz Φ_q tendríamos que para las variables cuyo valor es conocido sería 1 y el resto 0. Es decir:

$\text{Φq(6)} \rightarrow (0, 0, 0, 0, 0, 0, 1, 0)$

$\text{Φq(7)} \rightarrow (0, 0, 0, 0, 0, 0, 0, 1)$

Otra forma de medir el error en el cálculo de q es calculando el módulo del vector $Δq$. Tiene que ser lo más próximo a 0.

Por último, podemos saber si el mecanismo convergerá calculando el rango de la matriz jacobiana. Si su rango es igual al número de coordenadas dependientes, convergerá.

In [5]:

```
# PASO 8

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 ((8,1))
    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]
        X3 = q[4]
        Y3 = q[5]
        theta = q[6]
        beta = q[7]

        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[6]=0.3
q[7]=0.9
q=resuelve_prob_posicion(q,meca)
```

```
q=
 array([[0.1],
        [2.  ],
        [1.  ],
         [0.2],
        [2.02],
         [0.1],
         [0.3],
         [0.9]])
jacob=
array([[ 0.2       ,  4.        ,  0.        ,  0.        , 0.        ,
         0.        ,  0.        ,  0.        ],
       [-1.8       ,  3.6       ,  1.8       , -3.6       , 0.        ,
         0.        ,  0.        ,  0.        ],
       [ 0.        ,  0.        , -2.04      ,  0.2       , 2.04      ,
        -0.2       ,  0.        ,  0.        ],
       [ 0.        ,  0.        ,  0.        ,  0.        , 0.04      ,
          0.2      ,  0.        ,  0.        ],
       [ 0.        ,  1.        ,  0.        ,  0.        , 0.        ,
         0.        , -1.91067298,  0.        ],
       [ 0.        ,  0.        ,  0.        ,  0.        , 0.        ,
         1.        ,  0.        , -0.62160997],
```

```
       [ 0.         ,  0.        ,  0.        ,   0.        ,  0.       ,
         0.         ,  1.        ,  0.        ],
       [ 0.         ,  0.        ,  0.        ,   0.        ,  0.       ,
         0.         ,  0.        ,  1.        ]])
error iter1=
39.25953639517527
q=
array([[ 28.22919173],
       [  0.59104041],
       [ 13.81604819],
       [-11.14330913],
       [ 23.34336545],
       [  0.78332691],
        [ 0.3       ],
        [ 0.9       ]])
 jacob=
 array([[ 56.45838347,   1.18208083,   0.        ,   0.       ,
          0.         ,   0.        ,   0.        ,   0.       ],
       [ 28.82628708,  23.4686991 , -28.82628708, -23.4686991 ,
          0.         ,   0.        ,   0.        ,   0.       ],
       [ 0.         ,   0.        , -19.05463452, -23.85327209,
         19.05463452,  23.85327209,   0.        ,   0.       ],
       [ 0.         ,   0.        ,   0.        ,   0.       ,
         42.6867309  ,   1.56665382,   0.        ,   0.       ],
       [ 0.         ,   1.        ,   0.        ,   0.       ,
          0.         ,   0.        ,  -1.91067298,   0.       ],
       [ 0.         ,   0.        ,   0.        ,   0.       ,
          0.         ,   1.        ,   0.        ,  -0.62160997],
       [ 0.         ,   0.        ,   0.        ,   0.       ,
          0.         ,   0.        ,   1.        ,   0.       ],
       [ 0.         ,   0.        ,   0.        ,   0.       ,
          0.         ,   0.        ,   0.        ,   1.       ]])
error iter2=
20.904890696270627
q=
array([[14.17925714],
       [ 0.59104041],
       [ 5.33865016],
       [-3.79104275],
       [12.68073469],
       [ 0.78332691],
        [ 0.3       ],
        [ 0.9       ]])
jacob=
array([[ 28.35851429,  1.18208083,   0.        ,   0.       ,
          0.        ,  0.        ,   0.        ,   0.       ],
       [ 17.68121397,  8.76416632, -17.68121397, -8.76416632,
          0.        ,  0.        ,   0.        ,   0.       ],
       [  0.        ,  0.        , -14.68416907, -9.14873932,
         14.68416907,  9.14873932,   0.        ,   0.       ],
       [  0.        ,  0.        ,   0.        ,   0.       ,
         21.36146939,  1.56665382,   0.        ,   0.       ],
       [  0.        ,  1.        ,   0.        ,   0.       ,
          0.        ,  0.        ,  -1.91067298,   0.       ],
       [  0.        ,  0.        ,   0.        ,   0.       ,
          0.        ,  1.        ,   0.        , -0.62160997],
       [  0.        ,  0.        ,   0.        ,   0.       ,
          0.        ,  0.        ,   1.        ,   0.       ],
       [  0.        ,  0.        ,   0.        ,   0.       ,
          0.        ,  0.        ,   0.        ,   1.       ]])
error iter3=
12.181643862124048
q=
array([[  7.21836138],
       [ 0.59104041],
       [-0.04500024],
       [ 2.73797208],
       [ 7.35845594],
       [ 0.78332691],
        [ 0.3       ],
        [ 0.9       ]])
 jacob=
 array([[ 14.43672276,  1.18208083,   0.        ,  0.       ,
          0.        ,  0.        ,   0.        ,  0.       ],
       [ 14.52672324, -4.29386334, -14.52672324,  4.29386334,
          0.        ,  0.        ,   0.        ,  0.       ],
       [  0.        ,  0.        , -14.80691236,  3.90929034,
```

```
          14.80691236,  -3.90929034,   0.        ,   0.       ],
        [  0.        ,   0.        ,   0.        ,   0.       ,
          10.71691188,   1.56665382,   0.        ,   0.       ],
        [  0.        ,   1.        ,   0.        ,   0.       ,
           0.        ,   0.        ,  -1.91067298,   0.       ],
        [  0.        ,   0.        ,   0.        ,   0.       ,
           0.        ,   1.        ,   0.        ,  -0.62160997],
        [  0.        ,   0.        ,   0.        ,   0.       ,
           0.        ,   0.        ,   1.        ,   0.       ],
        [  0.        ,   0.        ,   0.        ,   0.       ,
           0.        ,   0.        ,   0.        ,   1.       ]])
error iter4=
16.566153866441173
q=
array([[ 3.86205465],
       [ 0.59104041],
       [ 4.26103513],
       [18.15368989],
       [ 4.71528304],
       [ 0.78332691],
        [ 0.3       ],
        [ 0.9       ]])
jacob=
array([[  7.7241093  ,   1.18208083,   0.        ,   0.       ,
           0.        ,   0.        ,   0.        ,   0.       ],
       [ -0.79796097, -35.12529894,   0.79796097, 35.12529894,
           0.        ,   0.        ,   0.        ,   0.       ],
       [  0.        ,   0.        ,  -0.90849581, 34.74072595,
          0.90849581, -34.74072595,   0.        ,   0.       ],
       [  0.        ,   0.        ,   0.        ,   0.       ,
          5.43056607,   1.56665382,   0.        ,   0.       ],
       [  0.        ,   1.        ,   0.        ,   0.       ,
           0.        ,   0.        ,  -1.91067298,   0.       ],
       [  0.        ,   0.        ,   0.        ,   0.       ,
           0.        ,   1.        ,   0.        , -0.62160997],
       [  0.        ,   0.        ,   0.        ,   0.       ,
           0.        ,   0.        ,   1.        ,   0.       ],
       [  0.        ,   0.        ,   0.        ,   0.       ,
           0.        ,   0.        ,   0.        ,   1.       ]])
error iter5=
10.230859831856687
q=
array([[  2.40366062],
       [ 0.59104041],
        [-1.33180514],
       [ 9.81050922],
       [ 3.42879413],
       [ 0.78332691],
       [ 0.3       ],
       [ 0.9       ]])
jacob=
array([[   4.80732124,   1.18208083,   0.        ,   0.       ,
           0.        ,   0.        ,   0.        ,   0.       ],
       [  7.47093153, -18.4389376 ,  -7.47093153,  18.4389376 ,
           0.        ,   0.        ,   0.        ,   0.       ],
        [  0.        ,   0.        ,  -9.52119854, 18.05436461,
          9.52119854, -18.05436461,   0.        ,   0.       ],
        [  0.        ,   0.        ,   0.        ,   0.       ,
          2.85758826,   1.56665382,   0.        ,   0.       ],
        [  0.        ,   1.        ,   0.        ,   0.       ,
           0.        ,   0.        ,  -1.91067298,   0.       ],
        [  0.        ,   0.        ,   0.        ,   0.       ,
           0.        ,   1.        ,   0.        , -0.62160997],
        [  0.        ,   0.        ,   0.        ,   0.       ,
           0.        ,   0.        ,   1.        ,   0.       ],
        [  0.        ,   0.        ,   0.        ,   0.       ,
           0.        ,   0.        ,   0.        ,   1.       ]])
error iter6=
4.4257900611952525
q=
array([[ 1.96122854],
       [ 0.59104041],
       [-0.88699162],
       [ 5.46786432],
       [ 2.84961562],
       [ 0.78332691],
       [ 0.3       ],
```

```
       [ 0.9       ]])
jacob=
array([[ 3.92245708,  1.18208083,  0.        ,  0.        ,  0.        ,
         0.        ,  0.        ,  0.        ],
       [ 5.69644031, -9.75364781, -5.69644031,  9.75364781,  0.        ,
         0.        ,  0.        ,  0.        ],
       [ 0.        ,  0.        , -7.47321448,  9.36907482, 7.47321448,
        -9.36907482,  0.        ,  0.        ],
       [ 0.        ,  0.        ,  0.        ,  0.        , 1.69923124,
         1.56665382,  0.        ,  0.        ],
       [ 0.        ,  1.        ,  0.        ,  0.        ,  0.        ,
         0.        , -1.91067298,  0.        ],
       [ 0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
         1.        ,  0.        , -0.62160997],
       [ 0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
         0.        ,  1.        ,  0.        ],
       [ 0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
         0.        ,  0.        ,  1.        ]])
error iter7=
2.0454288141200667
q=
array([[ 1.91132458],
       [ 0.59104041],
       [-0.97007002],
       [ 3.43429226],
       [ 2.65220414],
       [ 0.78332691],
       [ 0.3       ],
       [ 0.9       ]])
jacob=
array([[ 3.82264915,  1.18208083,  0.        ,  0.        ,  0.        ,
         0.        ,  0.        ,  0.        ],
       [ 5.76278919, -5.68650369, -5.76278919,  5.68650369,  0.        ,
         0.        ,  0.        ,  0.        ],
       [ 0.        ,  0.        , -7.24454832,  5.3019307 , 7.24454832,
        -5.3019307 ,  0.        ,  0.        ],
       [ 0.        ,  0.        ,  0.        ,  0.        , 1.30440828,
         1.56665382,  0.        ,  0.        ],
       [ 0.        ,  1.        ,  0.        ,  0.        ,  0.        ,
         0.        , -1.91067298,  0.        ],
       [ 0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
         1.        ,  0.        , -0.62160997],
       [ 0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
         0.        ,  1.        ,  0.        ],
       [ 0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
         0.        ,  0.        ,  1.        ]])
error iter8=
0.686053143009355
q=
array([[ 1.91067309],
       [ 0.59104041],
       [-0.92797619],
       [ 2.75018409],
       [ 2.62232754],
       [ 0.78332691],
       [ 0.3       ],
       [ 0.9       ]])
 jacob=
array([[ 3.82134618,  1.18208083,  0.        ,  0.        ,  0.        ,
         0.        ,  0.        ,  0.        ],
       [ 5.67729856, -4.31828735, -5.67729856,  4.31828735,  0.        ,
         0.        ,  0.        ,  0.        ],
       [ 0.        ,  0.        , -7.10060746,  3.93371436, 7.10060746,
        -3.93371436,  0.        ,  0.        ],
       [ 0.        ,  0.        ,  0.        ,  0.        , 1.24465508,
         1.56665382,  0.        ,  0.        ],
       [ 0.        ,  1.        ,  0.        ,  0.        ,  0.        ,
         0.        , -1.91067298,  0.        ],
       [ 0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
         1.        ,  0.        , -0.62160997],
       [ 0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
         0.        ,  1.        ,  0.        ],
       [ 0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
         0.        ,  0.        ,  1.        ]])
error iter9=
0.08408104758577133
q=
```

```
array([[ 1.91067298],
       [ 0.59104041],
       [-0.90718592],
       [ 2.66871708],
       [ 2.62161038],
       [ 0.78332691],
       [ 0.3       ],
       [ 0.9       ]])
jacob=
array([[ 3.82134596,  1.18208083,  0.        ,  0.        ,  0.        ,
         0.        ,  0.        ,  0.        ],
       [ 5.63571779, -4.15535334, -5.63571779,  4.15535334,  0.        ,
         0.        ,  0.        ,  0.        ],
       [ 0.        ,  0.        , -7.0575926 ,  3.77078035, 7.0575926 ,
        -3.77078035,  0.        ,  0.        ],
       [ 0.        ,  0.        ,  0.        ,  0.        , 1.24322076,
         1.56665382,  0.        ,  0.        ],
       [ 0.        ,  1.        ,  0.        ,  0.        ,  0.        ,
         0.        , -1.91067298,  0.        ],
       [ 0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
         1.        ,  0.        , -0.62160997],
       [ 0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
         0.        ,  1.        ,  0.        ],
       [ 0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
         0.        ,  0.        ,  1.        ]])
error iter10=
0.0012747159824017227
q=
array([[ 1.91067298],
       [ 0.59104041],
       [-0.90683518],
       [ 2.66749157],
       [ 2.62160997],
       [ 0.78332691],
       [ 0.3       ],
       [ 0.9       ]])
jacob=
array([[ 3.82134596,  1.18208083,  0.        ,  0.        ,  0.        ,
         0.        ,  0.        ,  0.        ],
       [ 5.63501631, -4.15290232, -5.63501631,  4.15290232,  0.        ,
         0.        ,  0.        ,  0.        ],
       [ 0.        ,  0.        , -7.05689029,  3.76832932, 7.05689029,
        -3.76832932,  0.        ,  0.        ],
       [ 0.        ,  0.        ,  0.        ,  0.        , 1.24321994,
         1.56665382,  0.        ,  0.        ],
       [ 0.        ,  1.        ,  0.        ,  0.        ,  0.        ,
         0.        , -1.91067298,  0.        ],
       [ 0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
         1.        ,  0.        , -0.62160997],
       [ 0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
         0.        ,  1.        ,  0.        ],
       [ 0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
         0.        ,  0.        ,  1.        ]])
error iter11=
2.963535767525358e-07
q=
array([[ 1.91067298],
       [ 0.59104041],
       [-0.9068351 ],
       [ 2.66749129],
       [ 2.62160997],
       [ 0.78332691],
       [ 0.3       ],
       [ 0.9       ]])
 jacob=
array([[ 3.82134596,  1.18208083,  0.        ,  0.        ,  0.        ,
         0.        ,  0.        ,  0.        ],
       [ 5.63501616, -4.15290175, -5.63501616,  4.15290175,  0.        ,
         0.        ,  0.        ,  0.        ],
       [ 0.        ,  0.        , -7.05689014,  3.76832875, 7.05689014,
        -3.76832875,  0.        ,  0.        ],
       [ 0.        ,  0.        ,  0.        ,  0.        , 1.24321994,
         1.56665382,  0.        ,  0.        ],
       [ 0.        ,  1.        ,  0.        ,  0.        ,  0.        ,
         0.        , -1.91067298,  0.        ],
       [ 0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
         1.        ,  0.        , -0.62160997],
```

```
        [ 0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
          0.        ,  1.        ,  0.        ],
        [ 0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
          0.        ,  0.        ,  1.        ]])
error iter12=
1.6207360324296026e-14
num iters:12
```

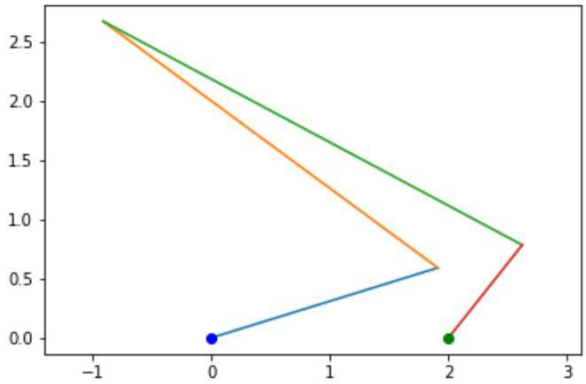
PASO 9: Dibujar el mecanismo

Para dibujar el mecanismo, definimos un cuadro de dibujo con los ejes de la misma dimensión. Seguidamente, dibujamos cada barra por separado. Para dibujar cada barra tendríamos que indicar las posiciones inicial y final, yendo por un lado las coordenadas en el eje X y por otro las coordenadas en el eje Y. Es decir, sería por ejemplo:

$Barra ~ 1 \rightarrow ([X_A, X_1], [Y_A, Y_1])$

In [6]:

```
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]

    theta = q[6]
    beta = q[7]


    plt.axis('equal')


    plt.plot ([meca["XA"], X1], [meca["YA"], Y1])        #[pos inicial (x1,x2), pos final (y1,y2)]
    plt.plot ([X1, X2], [Y1, Y2])
    plt.plot ([X2, X3], [Y2, Y3])
    plt.plot([X3,meca["XB"]], [Y3, meca["YB"]])

    plt.plot(meca["XA"], meca["YA"], 'bo')
    plt.plot(meca["XB"], meca["YB"], 'go')


    plt.show()#block=False)
    return

dibuja_mecanismo(q,meca)
```



MECANISMO CINCO BARRAS

PROBLEMAS VELOCIDAD Y ACELERACIÓN

PASO 1: MATRIZ JACOBIANA

Para resolver el problema velocidad, necesitamos otra vez la matriz jacobiana. El método de construcción aparece detallado en el notebook Problema_Posición_5B, por lo que para este caso copiaremos el código de los pasos realizados para poder conseguirla.

In [1]:

```
#! /usr/bin/env python

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
#%matplotlib inline (Para notebook)



print ('MECANISMO DE CINCO BARRAS')
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["L3"] = float (input ('Introduce longitud L3:'))
meca["L4"] = float (input ('Introduce longitud L4: '))
meca["theta"] = float (input ('Introduce angulo inicial theta:')) #En rad
meca["beta"] = float (input('Introduce angulo inicial beta: ')) #En rad
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], [2.02], [0.1], [meca["theta"]], [meca["beta"]]])
print('q: ' + str(q))

# JACOBIANO
def jacob_Phiq(q,meca):
    #Inicializa a cero la matriz jacobiana
    Jacob = np.zeros((8,8))

    #Extraer coordenadas

    X1 = q[0]
    Y1 = q[1]
    X2 = q[2]
    Y2 = q[3]
    X3 = q[4]
    Y3 = q[5]
    theta = q[6]
    beta = q[7]

    #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*(X3-X2)
    Jacob[2,3] = -2*(Y3-Y2)
    Jacob[2,4] = 2*(X3-X2)
    Jacob[2,5] = 2*(Y3-Y2)
    Jacob[3,4] = -2*(meca["XB"]-X3)
    Jacob[3,5] = -2*(meca["YB"]-Y3)

    if (abs(math.cos(theta)) < 0.7 ):
        Jacob[4,6] = meca["L1"]*math.sin(theta)
        Jacob[4,0] = 1
    else:
        Jacob[4,6] = -meca["L1"]*math.cos(theta)
        Jacob[4,1] = 1


    if (abs(math.cos(beta)) < 0.7):
        Jacob[5,7] = meca["L4"]*math.sin(beta)
        Jacob[5,4] = 1
    else:
        Jacob[5,7] = -meca["L4"]*math.cos(beta)
        Jacob[5,5] = 1

    Jacob[6,6] = 1
    Jacob[7,7] = 1

    return Jacob


def Phi (q,meca):
    #Inicializa a cero Phi
    Phi = np.zeros((8,1))

    #Extraer coordenadas
    X1 = q[0]
    Y1 = q[1]
    X2 = q[2]
    Y2 = q[3]
    X3 = q[4]
    Y3 = q[5]
    theta = q[6]
    beta = q[7]

    Phi[0] = X1**2 + Y1**2 - meca["L1"]**2
    Phi[1] = (X2-X1)**2 + (Y2-Y1)**2 -meca["L2"]**2
    Phi[2] = (X3-X2)**2 + (Y3-Y2)**2 - meca["L3"]**2
    Phi[3] = (meca["XB"] - X3)**2 + (meca["YB"] - Y3)**2 - meca["L4"]**2

    if (abs(math.cos(theta)) < 0.7 ):
        Phi[4] = X1-meca["L1"]*math.cos(theta)
    else:
        Phi[4] = Y1-meca["L1"]*math.sin(theta)

    if (abs(math.cos(beta)) < 0.7):
        Phi[5] = (X3-meca["XB"])-meca["L4"]*math.cos(beta)
    else:
        Phi[5] = Y3-meca["L4"]*math.sin(beta)

    return Phi



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 ((8,1))
    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]
        X3 = q[4]
        Y3 = q[5]
        theta = q[6]
        beta = q[7]

        fi=Phi(q,meca)

        J = jacob_Phiq(q,meca)


        deltaQ = np.linalg.solve(J,-fi)
        q = q + deltaQ
        error = np.linalg.norm(deltaQ) # El error es el modulo del
        vector i=i+1
    if (error > tolerancia):
        raise Exception ('No se puede alcanzar la posición')


    return q


J = jacob_Phiq(q,meca)
print ("Jacob=")
pprint.pprint(J)
```

```
MECANISMO DE CINCO BARRAS
=============================
Introduce longitud L1:2
Introduce longitud L2:3.5
Introduce longitud L3:4
Introduce longitud L4: 1
Introduce angulo inicial theta:0.3
Introduce angulo inicial beta: 0.9
Introduce coordenada en x del punto B:2
q: [[0.1 ]
 [2. ]
 [1.  ]
 [0.2 ]
 [2.02]
 [0.1 ]
 [0.3 ]
 [0.9 ]]
Jacob=
array([[ 0.2       ,  4.        ,  0.        ,  0.        , 0.        ,
         0.        ,  0.        ,  0.        ],
       [-1.8       ,  3.6       ,  1.8       , -3.6       , 0.        ,
         0.        ,  0.        ,  0.        ],
       [ 0.        ,  0.        , -2.04      ,  0.2       , 2.04      ,
        -0.2       ,  0.        ,  0.        ],
       [ 0.        ,  0.        ,  0.        ,  0.        , 0.04      ,
         0.2       ,  0.        ,  0.        ],
       [ 0.        ,  1.        ,  0.        ,  0.        , 0.        ,
         0.        , -1.91067298,  0.        ],
       [ 0.        ,  0.        ,  0.        ,  0.        , 1.        ,
         0.        ,  0.        ,  0.78332691],
       [ 0.        ,  0.        ,  0.        ,  0.        , 0.        ,
         0.        ,  1.        ,  0.        ],
       [ 0.        ,  0.        ,  0.        ,  0.        , 0.        ,
         0.        ,  0.        ,  1.        ]])
```

PASO 2: PROBLEMA VELOCIDAD

Consiste en determinar las velocidades de todas las variables del mecanismo una vez sabemos su posición q y la velocidad de los grados de libertad.

Partimos de la ecuación:

$\Phi_q =0 $

Derivando se obtiene:

$\Phi_q \dot{q} + \Phi_t = 0$

Siendo $\dot{q}$ el vector velocidad, Φ_q el jacobiano y Φ_t la derivada parcial de las ecuaciones de restricción respecto al tiempo. Para las ecuaciones de sólido rígido el valor de esta derivada es 0. Solo tendría un valor no nulo la correspondiente al ángulo, que en ese caso tendría la velocidad que nosotros le indiquemos.

De este modo la expresión quedaría:

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

Este sistema de ecuaciones tiene infinitas soluciones y por tanto hay que ampliar añadiendo un dato conocido de velocidad, lo que se hace añadiendo una fila a la matriz de coeficientes del lado izquierdo y un dato a la columna del lado derecho de la ecuación por cada grado de libertad.

De esta forma llegamos a un sistema de ecuaciones lineal matricial de la forma:

$\mathbf{A} \mathbf{x} = \mathbf{b}$

Habría que multiplicar en ambas partes de la igualdad por la A invertida en el lado izquierdo, del mismo modo que se hizo en el problema de posición. De esta manera quedaría:

$x = -A^{-1} b$

In [2]:

```
# PASO 2

def resuelve_prob_velocidad(q,qp, meca):

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

    return qp


qp = np.zeros ((8,1))
    #Velocidad del gdl.
qp[6]=0.1
qp[7] =0
qp=resuelve_prob_velocidad (q,qp, meca)

qp
```

Out[2]:

```
array([[-3.82134596],
       [ 0.1910673 ],
       [ 0.21667426],
       [ 2.2100774 ],
       [ 0.        ],
       [ 0.        ],
       [ 0.1       ],
       [ 0.        ]])
```

PASO 3: PROBLEMA ACELERACIÓN

El problema aceleración trata de determinar las aceleraciones de todas las variables del mecanismo, conociendo la posiciónq, la velocidad $\dot{q}$ y las aceleraciones de los grados de libertad.

Partimos la ecuación que se obtiene tras derivar la ecuación inicial para el problema de velocidad, es decir:

$\Phi_q \dot{q} + \Phi_t = 0$

Se deriva por segunda vez:

$\dot{\Phi}_q \dot{q} + \Phi_q \ddot{q} + \dot{\Phi}_t = 0$

Despejamos $\Phi_q \ddot{q}$:

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

Siendo Φ_q el jacobiano, $\ddot{q}$ el vector aceleración, $\dot{q}$ el vector velocidad, $\dot{\Phi}_q$ la derivada del jacobiano respecto al tiempo y $\dot{\Phi}_t$ es la derivada de las ecuaciones de restricción con respecto al tiempo, cuyo valor es nulo. Es decir, tendríamos:

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

Del mismo modo que en el problema velocidad, llamando b al conjunto formado por $\dot{\Phi}_q \dot{q}$ llegamos a un sistema de ecuaciones lineal matricial:

$ \mathbf{A} \mathbf{x} = \mathbf{b}$

Y despejando la x:

$\mathbf{x} = \mathbf{A}^{-1} \mathbf{b}$

El vector velocidad será:
\begin{equation} \mathbf{\dot{q}} = \begin{bmatrix} \dot{X}_1 \\ \dot{Y}_1 \\ \dot{X}_2 \\ \dot{Y}_2 \\ \dot{X}_3 \\ \dot{Y}_3 \\ \dot{\theta}\\ \dot{\beta} \\ \end{bmatrix} \end{equation}

Por otro lado, para calcular la derivada del jacobiano solo tenemos en cuenta las filas que hacen referencia a las ecuaciones de las coordenadas dependientes, ya que la última que añadimos para poder realizar los cálculos era adicional. Teniendo en cuenta esto, la derivada del jacobiano sería:
\begin{equation} \mathbf{\dot{\Phi}_q} = \begin{bmatrix} 2\dot{X}_1 & 2\dot{Y}_1 & 0 & 0 & 0 & 0 & 0 & 0\\ -2(\dot{X}_2-\dot{X}_1) & - 2(\dot{Y}_2-\dot{Y}_1) & 2(\dot{X}_2-\dot{X}_1) & 2(\dot{Y}_2-\dot{Y}_1) & 0 & 0 & 0 & 0\\ 0 & 0 & -2(\dot{X}_3-\dot{X}_2) & -2(\dot{Y}_3-\dot{Y}_2) & 2(\dot{X}_3-\dot{X}_2) & 2(\dot{Y}_3-\dot{Y}_2) & 0 & 0\\ 0 & 0 & 0 & 0 & 2\dot{X}_3 & 2\dot{Y}_3 & 0 & 0\\ \dot{\Phi}_q(4)\\ \dot{\Phi}_q(5)\\ \end{bmatrix} \end{equation}

1. $Si ~~ \cos (\theta) < {0.95} \rightarrow \dot{\Phi}_q(4) = \dot{\theta} · L_1 · \cos(\theta)$
2. $Si ~~ \cos (\theta) > {0.95} \rightarrow \dot{\Phi}_q(4) = \dot{\theta} · L_1 · \sin(\theta)$
3. $Si ~~ \cos (\beta) < {0.95} \rightarrow \dot{\Phi}_q(5) = \dot{\beta} · L_4 · \cos(\beta)$
4. $Si ~~ \cos (\beta) > {0.95} \rightarrow \dot{\Phi}_q(5) = \dot{\beta} · L_4 · \sin(\beta)$

Como ya tenemos $\dot{\Phi}_q$ y $\dot{q}$, podemos calcular b. Las dos últimas filas que añadimos son los valores de las aceleraciones angulares, datos que sabemos de antemano.
\begin{equation} \mathbf{b} = \begin{bmatrix} 2\dot{X}_1^2 & 2\dot{Y}_1^2 & 0 & 0 & 0 & 0 & 0 & 0\\ -2\dot{X}_1(\dot{X}_2-\dot{X}_1) & -2\dot{Y}_1(\dot{Y}_2-\dot{Y}_1) & 2\dot{X}_2(\dot{X}_2-\dot{X}_1) & 2\dot{Y}_2(\dot{Y}_2-\dot{Y}_1) & 0 & 0 & 0 & 0\\ 0 & 0 & -2\dot{X}_2(\dot{X}_3-\dot{X}_2) & -2\dot{Y}_2(\dot{Y}_3-\dot{Y}_2) & 2\dot{X}_3(\dot{X}_3-\dot{X}_2) & 2\dot{Y}_3(\dot{Y}_3-\dot{Y}_2) & 0 & 0\\ 0 & 0 & 0 & 0 & 2\dot{X}_3^2 & 2\dot{Y}_3^2 & 0 & 0\\ b(4)\\ b(5)\\ 1\\ 1\\ \end{bmatrix} \end{equation}

1. $Si ~~ \cos (\theta) < {0.95} \rightarrow b(4) = \dot{\theta}^2 · L_1 · \cos(\theta)$
2. $Si ~~ \cos (\theta) > {0.95} \rightarrow b(4) = \dot{\theta}^2 · L_1 · \sin(\theta)$
3. $Si ~~ \cos (\beta) < {0.95} \rightarrow b(5) = \dot{\beta}^2 · L_4 · \cos(\beta)$
4. $Si ~~ \cos (\beta) > {0.95} \rightarrow b(5) = \dot{\beta}^2 · L_4 · \sin(\beta)$

In [3]:

```
#PASO 3

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]
    beta = q[7]
    #Extraer las velocidades
    X1q = qp[0]
    Y1q = qp[1]
    X2q = qp[2]
    Y2q = qp[3]
    X3q = qp[4]
    Y3q = qp[5]
    thetaq = qp[6]
    betaq = qp[7]
    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*(X3q-X2q) - 2*Y2q*(Y3q-Y2q) + 2*X3q*(X3q-X2q) + 2*Y3q*(Y3q-Y2q)
    b[3] = 2*X3q**2 + 2*Y3q**2

    if (abs(math.cos(theta)) < (math.sqrt(2)/2) ):
        b[4] = thetaq**2 * (meca["L1"] * math.cos(theta))
    else:
        b[4] =  thetaq**2 * (meca["L1"] * math.sin(theta))

    if (abs(math.cos(beta)) < (math.sqrt(2)/2) ):
        b[5] = betaq**2 * (meca["L4"] * math.cos(beta))
    else:
        b[5] =  betaq**2 * (meca["L4"] * math.sin(beta))

    b[6] = -1 #Aceleracion conocida
    b[7] = -1
    qpp = np.linalg.solve(-jacob_Phiq(q,meca),b)


    return qpp

qpp=np.zeros ((8,1))
qpp = resuelve_prob_aceleracion(q,qp, qpp,meca)
qpp
```

Out[3]:

```
array([[-1.84487168e+02],
       [ 1.90476257e+00],
       [ 1.51174210e+01],
       [ 1.13030395e+02],
       [-7.83326910e-01],
       [ 1.56665382e-01],
       [ 1.00000000e+00],
       [ 1.00000000e+00]])
```

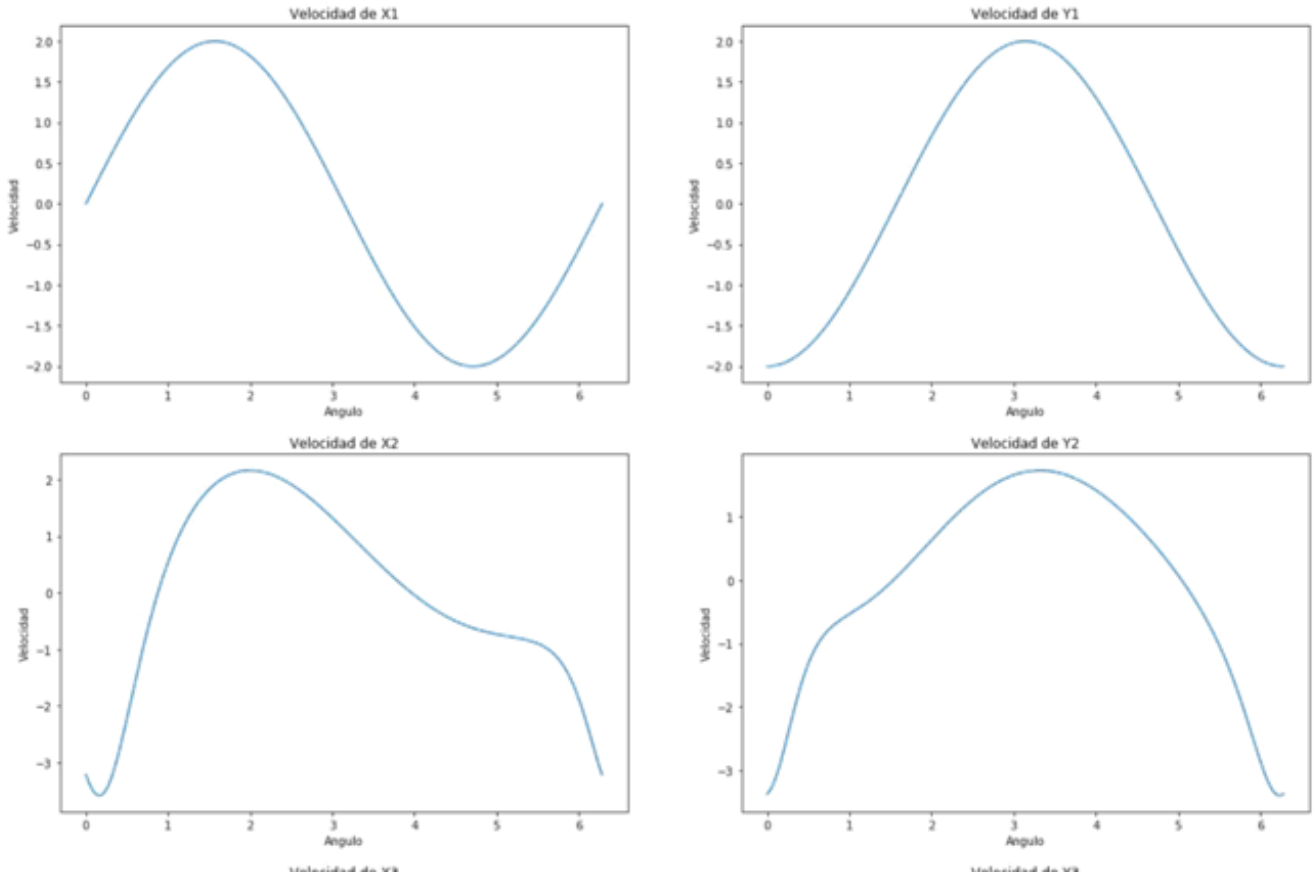
PASO 4: GRÁFICAS DE VELOCIDADES

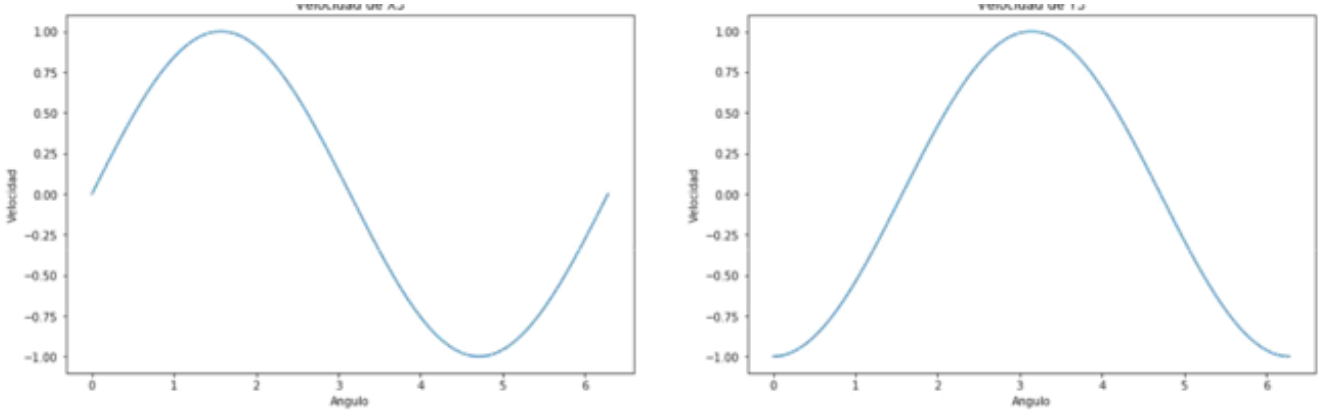
Vamos a representar por separado la gráfica de la velocidad en cada coordenada $(X_1, ~ Y_1, ~ X_2, ~ Y_2, ~ X_3 ~ e ~ Y_3)$.

In [4]:

```
#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[7] = t
        q = resuelve_prob_posicion (q,meca)

        qp = np.zeros((8,1))
        qp[6] = -1 #inicializar qp en 0 con qp[6] = 1
        rad/s qp[7] = -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])
        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)
```





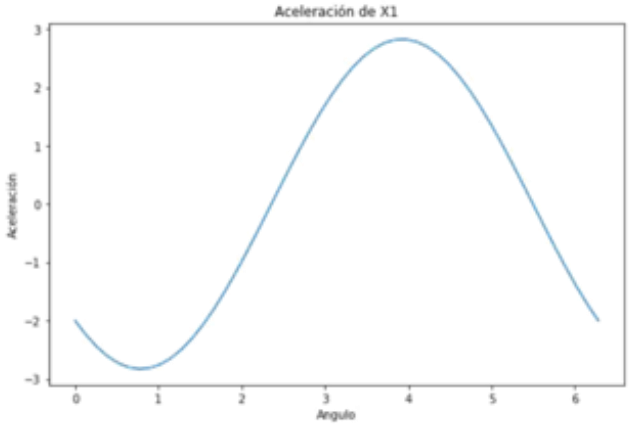
PASO 5: GRÁFICAS ACELERACIONES

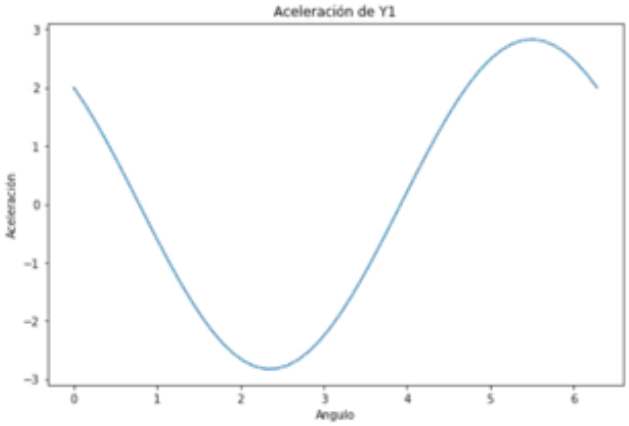
Haremos el mismo procedimiento que para la velocidad, representando en celdas separadas la aceleración de cada coordenada.

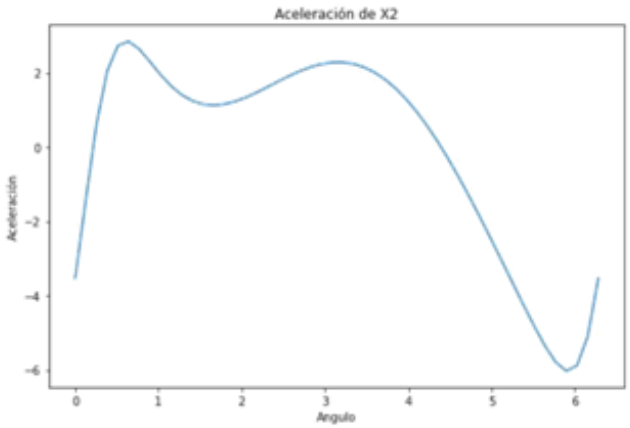
In [5]:

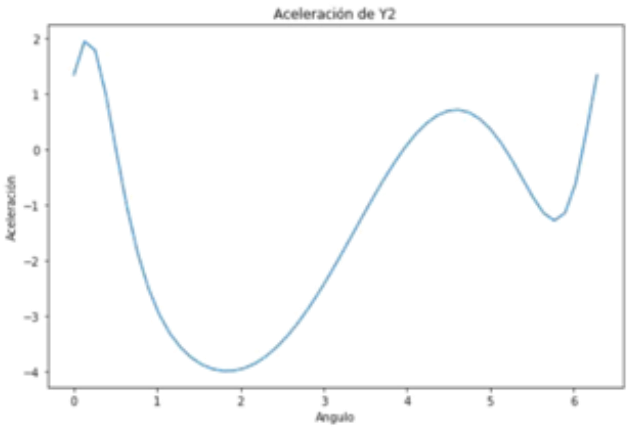
```
#PASO 5: GRÁFICAS ACELERACIONES

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

        qp = np.zeros((8,1))
        qp[6] = 1 #inicializar qp en 0 con qp[6] = 1
        rad/s qp[7] = 2
        qp = resuelve_prob_velocidad(q, qp, meca)

        qpp = np.zeros((8,1))
        qpp[6] = 1 #inicializar qp en 0 con qpp[4] = 1
        rad/s**2 qpp[7] = 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)
```

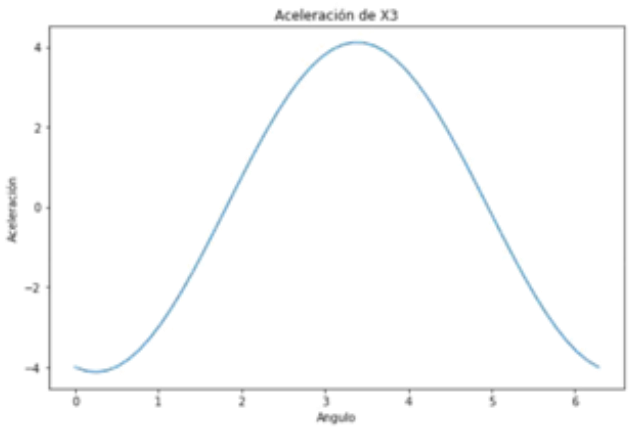
```
    plt.plot(th,AX2)
    plt.xlabel ('Angulo')
    plt.ylabel ('Aceleración')
    plt.title ('Aceleración de X2')

    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,mec
```

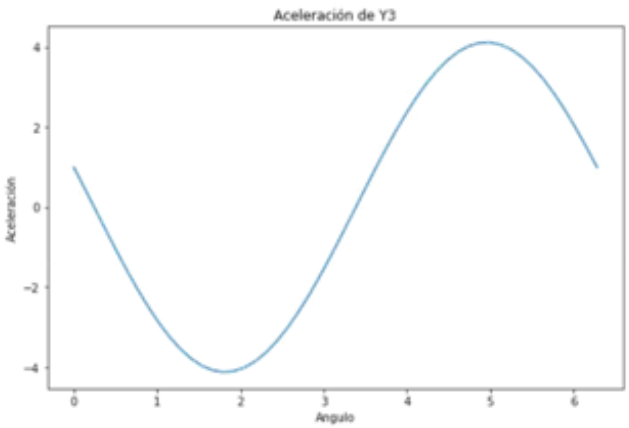












MECANISMO CINCO BARRAS

ANIMACIÓN

In [1]:

```
#! /usr/bin/env python

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
#%matplotlib inline (Para notebook)



print ('MECANISMO DE CINCO BARRAS')
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["L3"] = float (input ('Introduce longitud L3:'))
meca["L4"] = float (input ('Introduce longitud L4: '))
meca["theta"] = float (input ('Introduce angulo inicial theta:')) #En rad
meca["beta"] = float (input('Introduce angulo inicial beta: ')) #En rad
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], [2.02], [0.1], [meca["theta"]], [meca["beta"]]])
print('q: ' + str(q))

# JACOBIANO
def jacob_Phiq(q,meca):
    #Inicializa a cero la matriz jacobiana
    Jacob = np.zeros((8,8))

    #Extraer coordenadas

    X1 = q[0]
    Y1 = q[1]
    X2 = q[2]
    Y2 = q[3]
    X3 = q[4]
    Y3 = q[5]
    theta = q[6]
    beta = q[7]

    #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*(X3-X2)
    Jacob[2,3] = -2*(Y3-Y2)
    Jacob[2,4] = 2*(X3-X2)
    Jacob[2,5] = 2*(Y3-Y2)
    Jacob[3,4] = -2*(meca["XB"]-X3)
```

```
    Jacob[3,5] = -2*(meca["YB"]-Y3)

    if (abs(math.cos(theta)) < 0.7 ):
        Jacob[4,6] = meca["L1"]*math.sin(theta)
        Jacob[4,0] = 1
    else:
        Jacob[4,6] = -meca["L1"]*math.cos(theta)
        Jacob[4,1] = 1


    if (abs(math.cos(beta)) < 0.7):
        Jacob[5,7] = meca["L4"]*math.sin(beta)
        Jacob[5,4] = 1
    else:
        Jacob[5,7] = -meca["L4"]*math.cos(beta)
        Jacob[5,5] = 1

    Jacob[6,6] = 1
    Jacob[7,7] = 1

    return Jacob


def Phi (q,meca):
    #Inicializa a cero Phi
    Phi = np.zeros((8,1))

    #Extraer coordenadas
    X1 = q[0]
    Y1 = q[1]
    X2 = q[2]
    Y2 = q[3]
    X3 = q[4]
    Y3 = q[5]
    theta = q[6]
    beta = q[7]

    Phi[0] = X1**2 + Y1**2 - meca["L1"]**2
    Phi[1] = (X2-X1)**2 + (Y2-Y1)**2 -meca["L2"]**2
    Phi[2] = (X3-X2)**2 + (Y3-Y2)**2 - meca["L3"]**2
    Phi[3] = (meca["XB"] - X3)**2 + (meca["YB"] - Y3)**2 - meca["L4"]**2

    if (abs(math.cos(theta)) < 0.7 ):
        Phi[4] = X1-meca["L1"]*math.cos(theta)
    else:
        Phi[4] = Y1-meca["L1"]*math.sin(theta)

    if (abs(math.cos(beta)) < 0.7):
        Phi[5] = (X3-meca["XB"])-meca["L4"]*math.cos(beta)
    else:
        Phi[5] = Y3-meca["L4"]*math.sin(beta)

    return Phi



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 ((8,1))
    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]
        X3 = q[4]
        Y3 = q[5]
        theta = q[6]
        beta = q[7]

        fi=Phi(q,meca)

        J = jacob_Phiq(q,meca)


        deltaQ = np.linalg.solve(J,-fi)
        q = q + deltaQ
        error = np.linalg.norm(deltaQ) # El error es el modulo del
        vector i=i+1
    if (error > tolerancia):
        raise Exception ('No se puede alcanzar la posición')


    return q


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]
    beta = q[7]


    plt.axis('equal')


    plt.plot ([meca["XA"], X1], [meca["YA"], Y1])       #[pos inicial (x1,x2), pos final (y1,y2)]
    plt.plot ([X1, X2], [Y1, Y2])
    plt.plot ([X2, X3], [Y2, Y3])
    plt.plot([X3,meca["XB"]], [Y3, meca["YB"]])

    plt.plot(meca["XA"], meca["YA"], 'bo')
    plt.plot(meca["XB"], meca["YB"], 'go')

    plt.show()#block=False)
    return
```

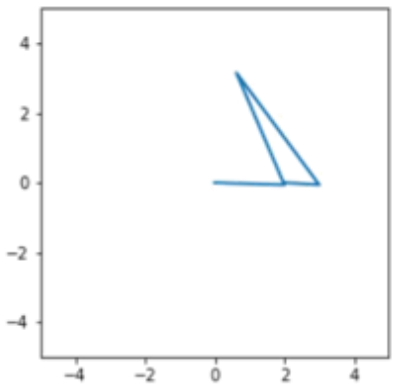
```
MECANISMO DE CINCO BARRAS
=============================
Introduce longitud L1:2
Introduce longitud L2:3.5
Introduce longitud L3:4
Introduce longitud L4: 1
Introduce angulo inicial theta:0.3
Introduce angulo inicial beta: 0.9
Introduce coordenada en x del punto B:2
q: [[0.1 ]
 [2. ]
 [1.  ]
 [0.2 ]
 [2.02]
 [0.1 ]
 [0.3 ]
 [0.9 ]]
```

In [2]:

```
%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(( -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*3.14159/200 # vel. angular
    q[6] = i*omega
    q[7] = i*omega*2

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

    # 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]
    beta = q[7]

    x=[meca["XA"], X1, X2, X3, meca["XB"]]
    y=[meca["YA"], Y1, Y2, Y3, meca["YB"]]


    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())
```

Out[2]:

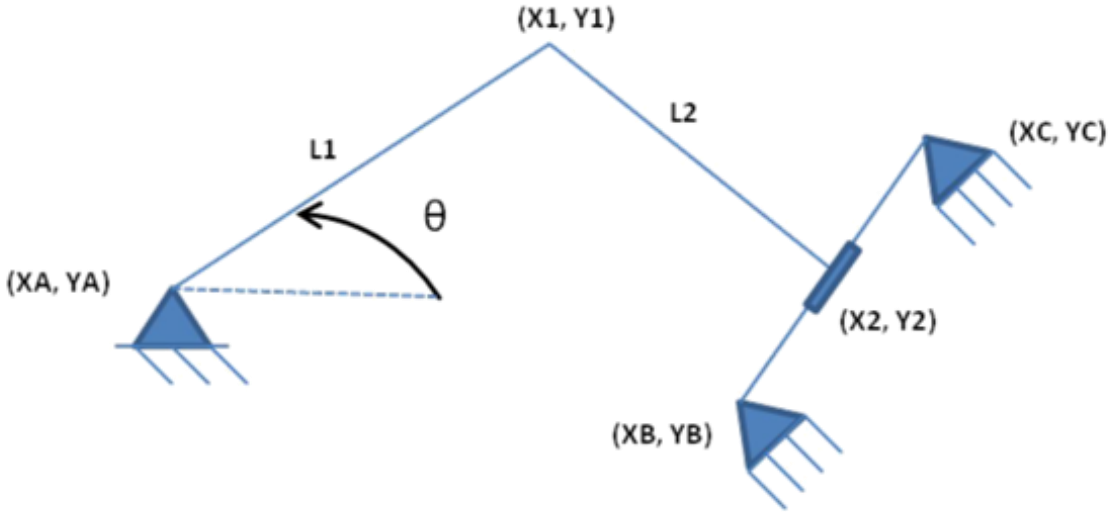


MECANISMO BIELA-MANIVELA

PROBLEMA POSICIÓN

PASO 1: MODELADO DEL MECANISMO

El mecanismo a modelar es el Biela-Manivela. Podemos ver todas las variables en la figura.



PASO 2: GRADOS DE LIBERTAD

Los grados de libertad se calculan:

$G = 3·(n-1) - 2·PI - PII$

Siendo:

PI -> Numero de pares binarios de un grado de libertad

PII -> Número de pares binarios de dos grados de libertad. En este caso tendríamos:
\begin{eqnarray} n &=& 4 \\ P_{I} &=& 4 \\ P_{II} &=& 0 \end{eqnarray}

Por lo tanto:

$G = 3·(4-1) - 2·4 - 0 = 1$

PASO 3: DEFINICIÓN DEL VECTOR q

El vector $\mathbf{q}$ de dimensión $N \times 1$ contiene las coordenadas dependientes del mecanismo. Es decir, contiene las coordenadas que no se mantienen fijas ya que varían con el tiempo.

Lo hemos modelizado enpleando las 5 coordenadas:
\begin{equation} \mathbf{q} = \begin{bmatrix} x_1 \\ y_1 \\ x_2 \\ y_2 \\ \theta \end{bmatrix} \end{equation}

PASO 4: IMPLEMENTACIÓN EN PYTHON

Al igual que en otros entornos de programación, necesitamos añadir librerías que contengan las funciones que vamos a utilizar. Esto es necesario hacerlo al principio del código. Las que vamos a usar son las siguientes:

1. numpy -> Sirve para trabajar con arrays y matrices, ofreciendo una interfaz similar a los comandos en MATLAB.
2. math -> La utilizaremos para usar funciones matemáticas.
3. pprint -> "pretty print", su función es ayudar a depurar el código.
4. matplotlib.pyplot -> Es necesaria para dibujar gráficas.

In [1]:

```
#PASO 4

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

PASO 5: LECTURA DE DATOS

Los datos iniciales de los que partiremos para resolver este mecanismo mediante análisis cinemático por métodos numéricos son los parámetros constantes que definen el mecanismo, es decir, las variables que no cambian con el tiempo. En este caso serían las longitudes de las barras y las posiciones de los apoyos.

Además, como el mecanismo tiene un único grado de libertad, tenemos que escoger la **variable independiente** entre las componentes del vector q. En este caso hemos escogido el ángulo, por lo que también será un dato de partida.

1. Longitudes de las barras: $L_1, ~ y ~ L_2$.
2. Posición de los tres apoyos: $X_A, ~ Y_A, ~ X_B, ~ Y_B ~ X_C ~ y ~ Y_C$.
3. Ángulo que forma la primera barra respecto a la horizontal en radianes: $θ(t=0)$.

Una vez tengamos esos datos, definiremos una posición inicial.

In [2]:

```
#PASO 5
print ('BIELA-MANIVELA')
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["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], [0.2], [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]
 [0.2]
 [1. ]
 [0.2]
 [0.5]]
```

PASO 6: MATRIZ DE RESTRICCIONES $\Phi(q)$

Este vector agrupa las ecuaciones de restricción y será de dimensión $mx1$.

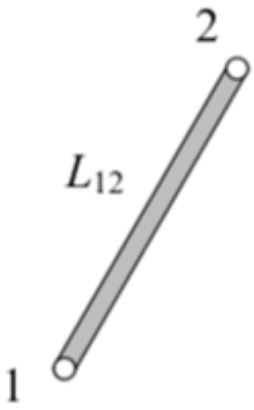
Estas ecuaciones podrían definirse empleando diferentes tipos de coordenadas: independientes, dependientes, relativas dependientes, de punto de referencia y naturales. Estas últimas son las que vamos a usar nosotros.

Para coordenadas naturales en el plano es necesario seguir un procedimiento:

1. Cada sólido debe tener al menos 2 puntos.
2. Cada par de rotación debe tener 1 punto.
3. Cada par prismático debe tener 3 puntos alineados.
4. Se pueden añadir tantos puntos adicionales como fuera necesario.
5. De los puntos mencionados, los fijos no entran en el vector q.

Para la formación de la matriz de restricciones, tenemos que tener en cuenta que hay restricciones de sólido rígido y de pares cinemáticos.

En este caso necesitamos:



Una única restricción:

$$(x_2 - x_1)^2 + (y_2 - y_1)^2 - L_{12}^2 = 0$$

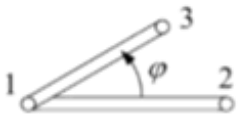
También necesitamos imponer una condición para la corredera, que consiste en hacer el producto escalar entre $B2$ y BC. Se basa en imponer que los 3 puntos estén alineados.

Colinealidad $\vec{12}$–$\vec{13}$:

$$\begin{aligned} \vec{12} \times \vec{13} &= 0 \\ (x_2 - x_1)(y_3 - y_1) - (y_2 - y_1)(x_3 - x_1) &= 0 \end{aligned}$$

Además, como el vector de coordenadas dependientes tiene 5 componentes tenemos que añadir una ecuación de restricción para el ángulo.

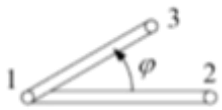
Esta última ecuación depende de si el ángulo en cuestión es demasiado pequeño. Esto se debe a que cuando un ángulo tiende a 0, su seno también lo hace, por lo que para esos casos utilizaríamos la restricción del coseno. En cambio, cuando el ángulo tiende más a 90°, es el coseno el que se aproxima a 0, por lo que en esos casos la restricción a utilizar sería la del seno.



Se añade ϕ al vector $\mathbf{q}$.
Usando el producto escalar: $\rightarrow\vec{12} \cdot \vec{13} = L_{12}L_{13}\cos\phi$.
Usando el producto vectorial: $\rightarrow\vec{12} \times \vec{13} = L_{12}L_{13}\sin\phi$.

Podemos añadir una de las dos restricciones a $\mathbf{\Phi(q)}$, pero mejor ambas (evitar singularidades):

$$\begin{aligned} (x_2 - x_1)(x_3 - x_1) + (y_2 - y_1)(y_3 - y_1) &= L_{12}L_{13}\cos\phi \\ (x_2 - x_1)(y_3 - y_1) - (y_2 - y_1)(x_3 - x_1) &= L_{12}L_{13}\sin\phi \end{aligned}$$



Cuando el ángulo ϕ (orientación *absoluta*) es relativo a tierra (12 es tierra), las ecuaciones se simplifican:

$$
\begin{aligned}
L_{12}(x_3 - x_1) + (y_2 - y_1)\overbrace{(y_3 - y_1)}^{0} &= L_{12}L_{13}\cos\phi \\
L_{12}(y_3 - y_1) - (y_2 - y_1)\overbrace{(x_3 - x_1)}^{0} &= L_{12}L_{13}\sin\phi
\end{aligned}
$$

$$
\downarrow
$$

$$
\begin{aligned}
x_3 - x_1 &= L_{13}\cos\phi \\
y_3 - y_1 &= L_{13}\sin\phi
\end{aligned}
$$

Para nuestro caso tendríamos:

$Si ~~ \cos (θ) < \frac{1}{\sqrt{2}} \rightarrow (X_1 - X_A) - L_1 · \cos(θ)$

$Si ~~ \cos (θ) > \frac{1}{\sqrt{2}} \rightarrow (Y_1 - Y_A) - L_1 · \sin (θ)$

La matriz quedaría:

1. $Si ~~ \cos (θ) < \frac{1}{\sqrt{2}}$ \begin{equation} \mathbf{\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_C-X_B)(Y_2-Y_B) - (Y_C-Y_B)(X_2-X_B) \\ X_1-L_1\cos(θ)\\ \end{bmatrix} \end{equation}
2. $Si ~~ \cos (θ) > \frac{1}{\sqrt{2}}$

\begin{equation} \mathbf{\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_C-X_B)(Y_2-Y_B) - (Y_C-Y_B)(X_2-X_B) \\ Y_1-L_1\sin(θ)\\ \end{bmatrix} \end{equation}

In [3]:

```
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 7: Matriz jacobiana Φq

Esta matriz de dimensiones mxn está compuesta por las derivadas parciales de las ecuaciones de restricción con respecto al vector de cooordeandas dependientes.

Por ejemplo endríamos:

Φ_q(0,0) = Derivada de Φ(0) respecto a X1. Φ_q(0,1) = Derivada de Φ (0) respecto a Y1. Φ_q(1,0) = Derivada de Φ(1) respecto a X1.

Tendríamos que construir la matriz elemento a elemento de esta manera.

Para la ecuación del ángulo hay que tener en cuenta que el jacobiano también tomará dos valores. Los posibles son:

1. $Si ~~ \cos (θ) < \frac{1}{\sqrt{2}}$

   $\Phi_q(3,0) = 1$ $\Phi_q(3,4) = L1 · \sin(θ)$

1. $Si ~~ \cos (θ) > \frac{1}{\sqrt{2}}$

   $\Phi_q(3,1) = 1$ $\Phi_q(3,4) = -L1 · \cos(θ)$

Es decir, tenemos dos posibles matrices jacobianas:

1. $Si ~~ \cos (θ) < \frac{1}{\sqrt{2}}$

\begin{equation} \mathbf{\Phi_q} = \begin{bmatrix} 2X_1 & 2Y_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_B-X_C & 0\\ 1 & 0 & 0 & 0 & L1\sin(θ)\\ \end{bmatrix} \end{equation}

1. $Si ~~ \cos (θ) > \frac{1}{\sqrt{2}}$

\begin{equation} \mathbf{\Phi_q} = \begin{bmatrix} 2X_1 & 2Y_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_B-X_C & 0 \\ 0 & 1 & 0 & 0 & -L1\cos(θ) \\ \end{bmatrix} \end{equation}

In [4]:

```
#PASO 7

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["XB"] - meca["XC"]

    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 8: RESOLUCIÓN DEL PROBLEMA POSICIÓN

El objetivo es obtener los valores de las coordenadas dependientes del vector q.

Para resolverlo partimos de la ecuación:

$\Phi(q+Δq) = \Phi + \Phi_q · Δq = 0$

De donde despejamos:

$\Phi_q · Δq = -\Phi$

Esta ecuación se convertiría en $Ax=b$, siendo A el jacobiano y b Φ. Sin embargo, no se pueden dividir matrices de esa manera, por lo que tenemos que multiplicar a ambos lados de la igualdad por la izquierda por A^{-1}, es decir, por la inversa del jacobiano $Φq^{-1}$:

$ \underbrace{ \Phi q^-1 · Φq}_{= \mathbf{I_n}} · Δq = Φq^{-1} ·-Φ$

Por el lado izquierdo al multiplicar el jacobiano por su inversa toma el valor la unidad, por lo que quedaría:

$Δq = \Phi_q^{-1} · -Φ$

Y tpara terminar, tendríamos que el nuevo valor de q sería:

$q = q + Δq$

Hay que repetir este proceso hasta que el vector Φ se aproxime a 0, lo que indicaría la validez del vector q calculado. Sin embargo, hay datos iniciales para los que el mecanismo non converge. Por ejemplo, si decimos que $X_A=0, ~ X_A=0, ~ X_B=50, ~ Y_B=0 ~ y ~ L_1=1, ~ L_2=2, ~ L_3=3$, no existe tal posición. Por ello, tenemos que poner un límite de iteraciones, como por ejemplo 100, y si llega a dicho límite tendremos que el mecanismo no converge.

Para poder operar con las matricces de ese modo, es necesario que las dimensiones sean correctas. En este caso tendríamos que añadir una fila a la matriz Φ y a Φ_q. Como el mecanismo tiene un grado de libertad tenemos que asignar un valor conocido a una de las variables dependientes, como ya se explicó anteriormente. Esto se traduce en que para la matriz de restricciones $\Phi(4)=0$, ya que su valor permanece invariable. Por otro lado para la matriz Φ_q tendríamos que para la variable conocido su valor sería uno y el resto 0. Es decir:

$\text{Φq(4)} \rightarrow (0, 0, 0, 0, 1)$

Otra forma de meidir el error en el cálculo de q es calculando el módulo del vector $Δq$. Tiene que ser lo más próximo a 0.

Por último, podemos saber si el mecanismo convergerá calculando el rango de la matriz jacobiana. Si su rango es igual al número de coordenadas dependientes, convergerá.

In [5]:

```
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 s
olucion

        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)
```

```
q=
array([[0.1],
       [0.2],
       [1. ],
       [0.2],
       [0.5]])
Phi=
array([[-0.95      ],
       [-3.19      ],
       [-0.2       ],
       [-0.27942554],
       [ 0.        ]])
jacob=
array([[ 0.2       ,  0.4       , 0.        ,  0.       ,  0.       ],
       [-1.8       , -0.        , 1.8       ,  0.       ,  0.       ],
       [ 0.        ,  0.        , 0.        , -1.       ,  0.       ],
       [ 0.        ,  1.        , 0.        ,  0.       , -0.87758256],
       [ 0.        ,  0.        , 0.        ,  0.       ,  1.       ]])
rango=5

error iter1=
7.296958499116243
q=
array([[4.29114892],
       [0.47942554],
       [6.96337115],
       [0.        ],
       [0.5       ]])
Phi=
array([[17.64380792],
       [ 3.37062045],
       [ 0.        ],
       [ 0.        ],
       [ 0.        ]])
jacob=
array([[ 8.58229785,  0.95885108, 0.        ,  0.       ,  0.       ],
       [-5.34444444,  0.95885108, 5.34444444, -0.95885108,  0.       ],
       [ 0.        ,  0.        , 0.        , -1.       ,  0.       ],
       [ 0.        ,  1.        , 0.        ,  0.       , -0.87758256],
       [ 0.        ,  0.        , 0.        ,  0.       ,  1.       ]])
rango=5

error iter2=
3.3828727800834577
q=
array([[2.23531164],
       [0.47942554],
       [4.27685643],
       [0.        ],
       [0.5       ]])
Phi=
 array([[4.22646695],
       [0.39775401],
       [0.        ],
       [0.        ],
       [0.        ]])
jacob=
array([[ 4.47062327,  0.95885108, 0.        ,  0.       ,  0.       ],
       [-4.0830896 ,  0.95885108, 4.0830896 , -0.95885108,  0.       ],
       [ 0.        ,  0.        , 0.        , -1.       ,  0.       ],
       [ 0.        ,  1.        , 0.        ,  0.       , -0.87758256],
       [ 0.        ,  0.        , 0.        ,  0.       ,  1.       ]])
rango=5

error iter3=
```

```
1.407547718645164
q=
array([[1.28992512],
       [0.47942554],
       [3.23405496],
       [0.        ],
       [0.5       ]])
Phi=
array([[ 8.93755663e-01],
       [ 9.48967456e-03],
       [ 0.00000000e+00],
       [-4.99600361e-16],
       [ 0.00000000e+00]])
jacob=
 array([[ 2.57985024,  0.95885108,  0.        ,  0.        ,  0.        ],
        [-3.88825968,  0.95885108,  3.88825968, -0.95885108,  0.        ],
        [ 0.        ,  0.        ,  0.        , -1.        ,  0.        ],
        [ 0.        ,  1.        ,  0.        ,  0.        , -0.87758256],
        [ 0.        ,  0.        ,  0.        ,  0.        ,  1.        ]])
rango=5

error iter4=
0.4916647418817361
q=
array([[0.94348809],
       [0.47942554],
       [2.88517733],
       [0.        ],
       [0.5       ]])
Phi=
array([[ 1.20018618e-01],
       [ 5.95651328e-06],
       [ 0.00000000e+00],
       [-1.66533454e-16],
       [ 0.00000000e+00]])
jacob=
 array([[ 1.88697617,  0.95885108,  0.        ,  0.        ,  0.        ],
        [-3.88337848,  0.95885108,  3.88337848, -0.95885108,  0.        ],
        [ 0.        ,  0.        ,  0.        , -1.        ,  0.        ],
        [ 0.        ,  1.        ,  0.        ,  0.        , -0.87758256],
        [ 0.        ,  0.        ,  0.        ,  0.        ,  1.        ]])
rango=5

error iter5=
0.0899502632512201
q=
array([[0.87988441],
       [0.47942554],
       [2.82157212],
       [0.        ],
       [0.5       ]])
Phi=
array([[4.04542737e-03],
       [2.35278463e-12],
       [0.00000000e+00],
       [0.00000000e+00],
       [0.00000000e+00]])
jacob=
 array([[ 1.75976883,  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 iter6=
0.0032510510290496538
q=
array([[0.87758557],
       [0.47942554],
       [2.81927328],
       [0.        ],
       [0.5       ]])
Phi=
array([[5.2846664e-06],
       [0.0000000e+00],
       [0.0000000e+00],
```

```
       [0.0000000e+00],
       [0.0000000e+00]])
jacob=
array([[  1.75517115,  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 iter7=
4.258073013352064e-06
q=
array([[0.87758256],
       [0.47942554],
       [2.81927027],
       [0.        ],
       [0.5       ]])
Phi=
 array([[ 9.06563713e-12],
       [-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 iter8=
7.304419941333233e-12
num iters:8
```

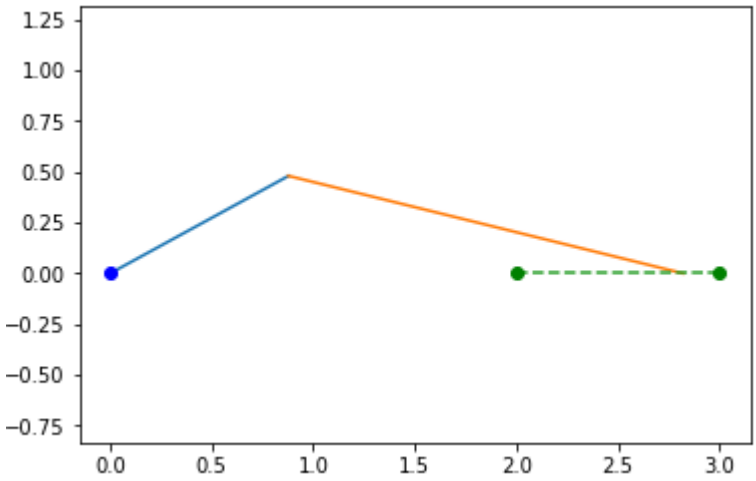
PASO 9: Dibujar el mecanismo

Para dibujar el mecanismo, definimos un cuadro de dibujo con los ejes de la misma dimensión. Seguidamente, dibujamos cada barra por separado. Para dibujar cada barra tendríamos que indicar las posiciones inicial y final, yendo por un lado las coordenadas en el eje X y por otro las coordenadas en el eje Y. Es decir, sería por ejemplo:

$Barra ~ 1 \rightarrow ([X_A, X_1], [Y_A, Y_1])$

In [6]:

```
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])    #[pos inicial (x1,x2), pos final (y1,y2)]
    plt.plot ([X1, X2], [Y1, Y2])
    plt.plot ([meca["XB"],meca["XC"]], [meca["YB"],meca["YC"]], linestyle='dashed')

    plt.plot(meca["XA"], meca["YA"], 'bo')
    plt.plot(meca["XB"], meca["YB"], 'go')
    plt.plot(meca["XC"], meca["YC"], 'go')
    plt.show()#block=False)
    return

dibuja_mecanismo(q,meca)
```


1.25
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
0.0
0.5
1.0
1.5
2.0
2.5
3.0

MECANISMO BIELA-MANIVELA

PROBLEMAS VELOCIDAD Y ACELERACIÓN

PASO 1: MATRIZ JACOBIANA

Para resolver el problema velocidad, necesitamos otra vez la matriz jacobiana. El método de construcción aparece detallado en el notebook Problema_Posición_4B, por lo que para este caso copiaremos el código de los pasos realizados para poder conseguirla.

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
#%matplotlib inline (Para notebook)



print ('BIELA-MANIVELA')
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["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], [0.2], [1], [0.2], [meca["theta"]]])
print('q: ' + str(q))

# JACOBIANO
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["YB"] - meca["YC"]
    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


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] = (meca["XC"] - meca["XB"])*(Y2-meca["YB"]) - (meca["YC"]-meca["YB"])*(X2-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



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)

        J = jacob_Phiq(q,meca)


        deltaQ = np.linalg.solve(J,-fi)
        q = q + deltaQ
        error = np.linalg.norm(deltaQ) # El error es el modulo del
        vector i=i+1
    if (error > tolerancia):
        raise Exception ('No se puede alcanzar la posición')

    return q
```

```
q = resuelve_prob_posicion(q,meca)
```

```
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]
 [0.2]
 [1. ]
 [0.2]
 [0.5]]
```

PASO 2: PROBLEMA VELOCIDAD

Consiste en determinar las velocidades de todas las variables del mecanismo una vez sabemos su posición q y la velocidad de los grados de libertad.

Partimos de la ecuación:

$\Phi_q =0 $

Derivando se obtiene:

$\Phi_q \dot{q} + \Phi_t = 0$

Siendo $\dot{q}$ el vector velocidad, Φ_q el jacobiano y Φ_t la derivada parcial de las ecuaciones de restricción respecto al tiempo. Para las ecuaciones de sólido rígido el valor de esta derivada es 0. Solo tendría un valor no nulo la correspondiente al ángulo, que en ese caso tendría la velocidad que nosotros le indiquemos.

De este modo la expresión quedaría:

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

Este sistema de ecuaciones tiene infinitas soluciones y por tanto hay que ampliar añadiendo un dato conocido de velocidad, lo que se hace añadiendo una fila a la matriz de coeficientes del lado izquierdo y un dato a la columna del lado derecho de la ecuación por cada grado de libertad.

De esta forma llegamos a un sistema de ecuaciones lineal matricial de la forma:

$\mathbf{A} \mathbf{x} = \mathbf{b}$

Habría que multiplicar en ambas partes de la igualdad por la A invertida en el lado izquierdo, del mismo modo que se hizo en el problema de posición. De esta manera quedaría:

$x = -A^{-1} b$

In [12]:

```
#PASO 2

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
```

Out[12]:

```
array([[ 0.54630249],
       [-1.        ],
       [ 0.79321425],
```

```
       [ 0.        ],
       [ 1.        ]])
```

PASO 3: PROBLEMA ACELERACIÓN

El problema aceleración trata de determinar las aceleraciones de todas las variables del mecanismo, conociendo la posiciónq, la velocidad $\dot{q}$ y las aceleraciones de los grados de libertad.

Partimos la ecuación que se obtiene tras derivar la ecuación inicial para el problema de velocidad, es decir:

$\Phi_q \dot{q} + \Phi_t = 0$

Se deriva por segunda vez:

$\dot{\Phi}_q \dot{q} + \Phi_q \ddot{q} + \dot{\Phi}_t = 0$

Despejamos $\Phi_q \ddot{q}$:

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

Siendo Φ_q el jacobiano, $\ddot{q}$ el vector aceleración, $\dot{q}$ el vector velocidad, $\dot{\Phi}_q$ la derivada del jacobiano respecto al tiempo y $\dot{\Phi}_t$ es la derivada de las ecuaciones de restricción con respecto al tiempo, cuyo valor es nulo. Es decir, tendríamos:

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

Del mismo modo que en el problema velocidad, llamando b al conjunto formado por $\dot{\Phi}_q \dot{q}$ llegamos a un sistema de ecuaciones lineal matricial:

$ \mathbf{A} \mathbf{x} = \mathbf{b}$

Y despejando la x:

$\mathbf{x} = \mathbf{A}^{-1} \mathbf{b}$

El vector velocidad será:
\begin{equation} \mathbf{q_p} = \begin{bmatrix} \dot{X}_1 \\ \dot{Y}_1 \\ \dot{X}_2 \\ \dot{Y}_2 \\ \dot{\theta}\\ \end{bmatrix} \end{equation}

Por otro lado, para calcular la derivada del jacobiano solo tenemos en cuenta las filas que hacen referencia a las ecuaciones de las coordenadas independientes, ya que la última que añadimos para poder realizar los cálculos era adicional. Teniendo en cuenta esto, la derivada del jacobiano sería:

1. $Si ~~ \cos (θ) < \frac{1}{\sqrt{2}}$

\begin{equation} \mathbf{\dot{\Phi}_q} = \begin{bmatrix} 2\dot{X}_1 & 2\dot{Y}_1 & 0 & 0 & 0 \\ -2(\dot{X}_2-\dot{X}_1) & -2(\dot{Y}_2-\dot{Y}_1) & 2(\dot{X}_2-\dot{X}_1) & 2(\dot{Y}_2-\dot{Y}_1) & 0\\ 0 & 0 & \dot{Y}_C-\dot{Y}_B & \dot{X}_B-\dot{X}_C & 0 \\ 1 & 0 & 0 & 0 & \dot{\theta}L_1\cos(\theta)\ \end{bmatrix} \end{equation}

1. $Si ~~ \cos (θ) > \frac{1}{\sqrt{2}}$

\begin{equation} \mathbf{\dot{\Phi}_q} = \begin{bmatrix} 2\dot{X}_1 & 2\dot{Y}_1 & 0 & 0 & 0 \\ -2(\dot{X}_2-\dot{X}_1) & -2(\dot{Y}_2-\dot{Y}_1) & 2(\dot{X}_2-\dot{X}_1) & 2(\dot{Y}_2-\dot{Y}_1) & 0\\ 0 & 0 & \dot{Y}_C-\dot{Y}_B & \dot{X}_B-\dot{X}_C & 0 \\ 0 & 1 & 0 & 0 & \dot{\theta}L_1\sin(\theta)\\ \end{bmatrix} \end{equation}

Como ya tenemos $\dot{\Phi}_q$ y $\dot{q}$, podemos calcular b. La última fila que añadimos es el valor de la aceleración angular, dato que sabemos de antemano.

1. $Si ~~ \cos (\theta) < \frac{1}{\sqrt{2}}$

\begin{equation} \mathbf{b} = \begin{bmatrix} 2\dot{X}_1^2 + 2\dot{Y}_1^2 \\ -2\dot{X}_1(\dot{X}_2-\dot{X}_1) & -2\dot{Y}_1(\dot{Y}_2-\dot{Y}_1) & 2\dot{X}_2(\dot{X}_2-\dot{X}_1) & 2\dot{Y}_2(\dot{Y}_2-\dot{Y}_1) & 0\\ 2\dot{X}_2^2 + 2\dot{Y}_2^2 \\ \dot{\theta}^2L_1\cos(\theta)\\ 1\\ \end{bmatrix} \end{equation}

1. $Si ~~ \cos (\theta) > \frac{1}{\sqrt{2}}$

\begin{equation} \mathbf{b} = \begin{bmatrix} 2\dot{X}_1^2 + 2\dot{Y}_1^2 \\ -2\dot{X}_1(\dot{X}_2-\dot{X}_1) & -2\dot{Y}_1(\dot{Y}_2-\dot{Y}_1) & 2\dot{X}_2(\dot{X}_2-\dot{X}_1) & 2\dot{Y}_2(\dot{Y}_2-\dot{Y}_1) & 0\\ 2\dot{X}_2^2 + 2\dot{Y}_2^2 \\ \dot{\theta}^2L_1\sin(\theta)\\ 1\\ \end{bmatrix} \end{equation}

In [3]:

```
#PASO 3

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))

    b[4] = 1 #Aceleracion conocida
    qpp = np.linalg.solve(-jacob_Phiq(q,meca),b)


    return qpp


qpp=np.zeros((5,1))
qpp=resuelve_prob_aceleracion(q,qp,qpp,meca)
qpp
```

Out[3]:

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

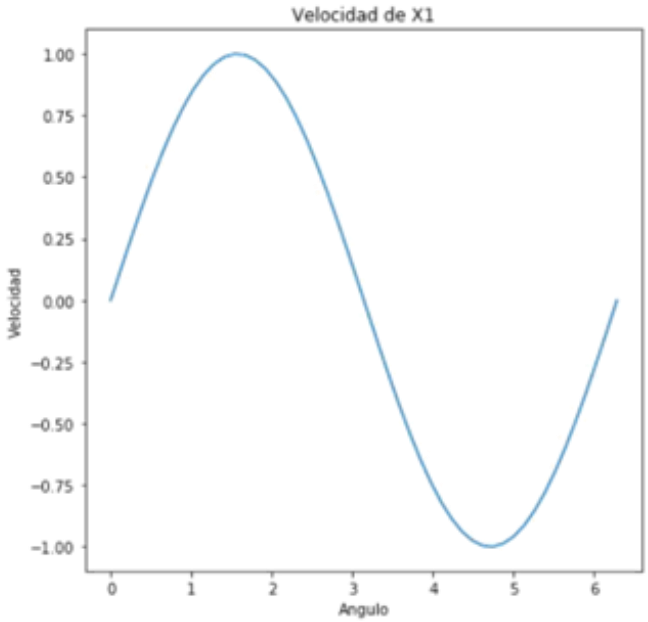
PASO 4: GRÁFICAS DE VELOCIDADES

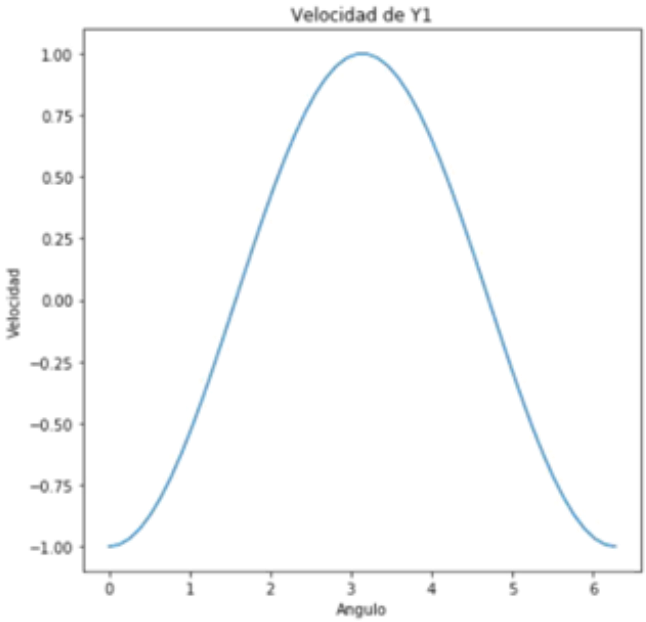
Vamos a representar por separado la gráfica de la velocidad en cada coordenada $(X_1, ~ Y_1, ~ X_2 ~ e ~ Y_2)$.

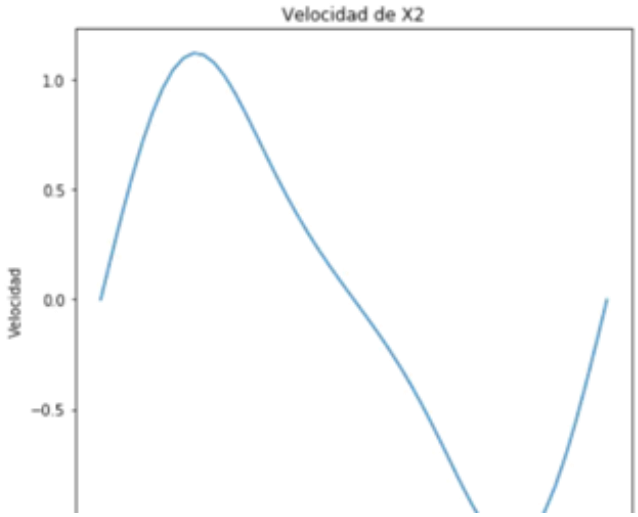
In [14]:

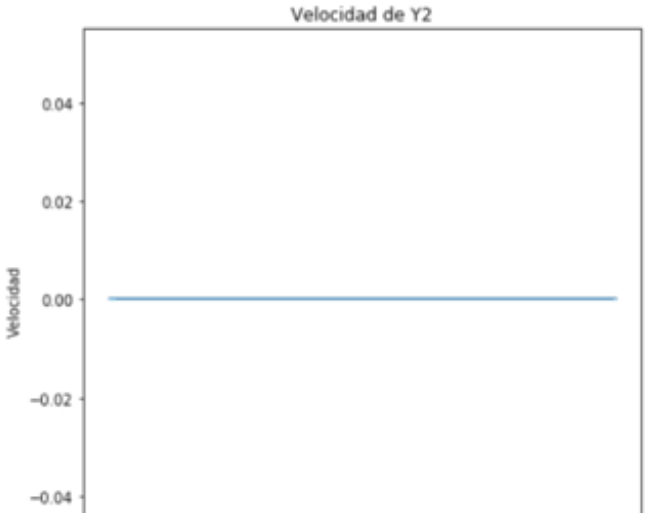
```
#PASO 4: GRÁFICAS DE VELOCIDADES
def grafica_velocidad(q,meca):

    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)
```







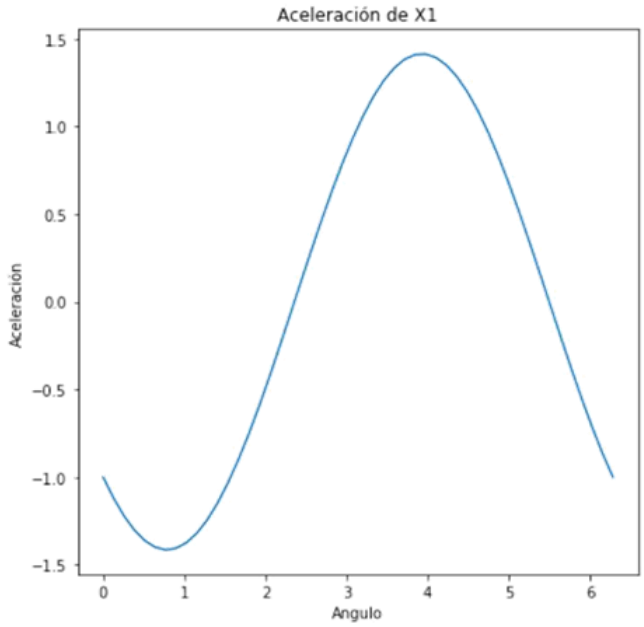


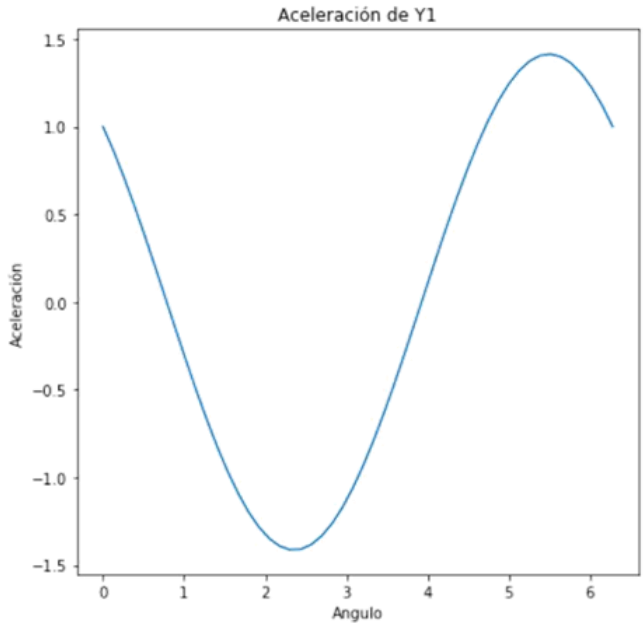
PASO 5: GRÁFICAS ACELERACIONES

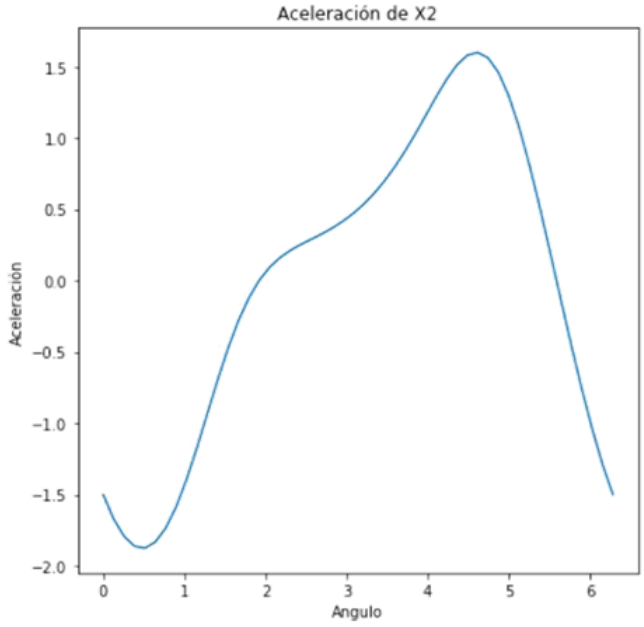
Haremos el mismo procedimiento que para la velocidad, representando en celdas separadas la aceleración de cada coordenada.

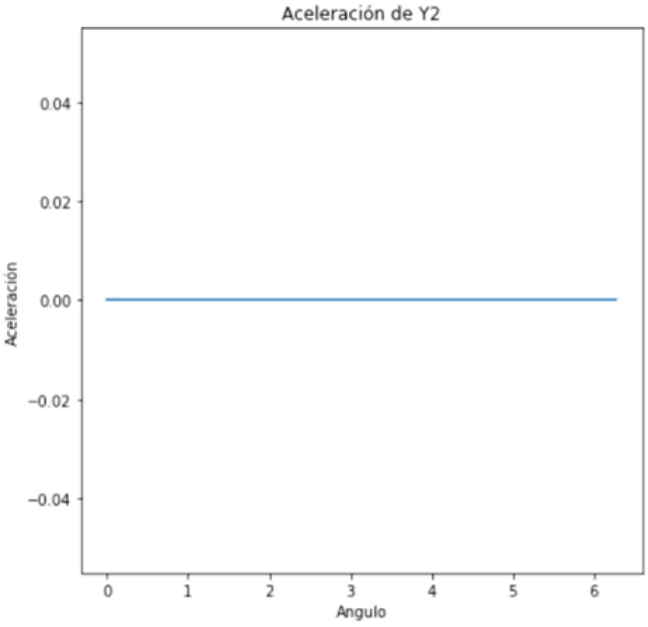
In [16]:

```
#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))


    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 ('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)
```


Aceleración de X1
Aceleración
Angulo


Aceleración de Y1
Aceleración
Angulo


Aceleración de X2
Aceleración
Angulo


Aceleración de Y2
Aceleración
Angulo

MECANISMO BIELA-MANIVELA

ANIMACIÓN

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
#%matplotlib inline (Para notebook)



print ('BIELA-MANIVELA')
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["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], [0.2], [1], [0.2], [meca["theta"]]])
print('q: ' + str(q))

# JACOBIANO
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["YB"] - meca["YC"]
    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


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] = (meca["XC"] - meca["XB"])*(Y2-meca["YB"]) - (meca["YC"]-meca["YB"])*(X2-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



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)

        J = jacob_Phiq(q,meca)


        deltaQ = np.linalg.solve(J,-fi)
        q = q + deltaQ
        error = np.linalg.norm(deltaQ) # El error es el modulo del
        vector i=i+1
    if (error > tolerancia):
        raise Exception ('No se puede alcanzar la posición')

    return q

q = resuelve_prob_posicion(q,meca)

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]) #[pos inicial (x1,x2), pos final (y1,y2)]
    plt.plot ([X1, X2], [Y1, Y2])
    plt.plot ([meca["XB"],meca["XC"]], [meca["YB"],meca["YC"]], linestyle='dashed')

    plt.plot(meca["XA"], meca["YA"], 'bo')
    plt.plot(meca["XB"], meca["YB"], 'go')
    plt.plot(meca["XC"], meca["YC"], 'go')

    plt.show()#block=False)
    return
```

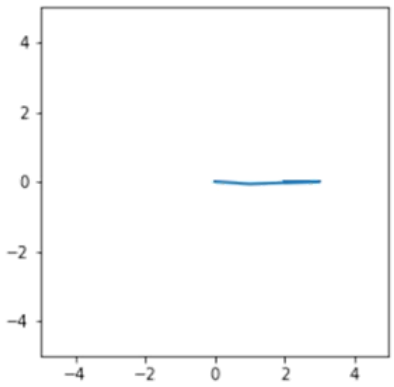
```
BIELA-MANIVELA
==============================
Introduce longitud L1:1
Introduce longitud L2:2
Introduce angulo inicial theta:0.5
Introduce coordenada en x del punto B:2
Introduce coordenada en x del punto C:3
Introduce coordenada en y del punto B:0
Introduce coordenada en y del punto C:0
q: [[0.1]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

In [2]:

```
%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"], 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())
```

Out[2]:

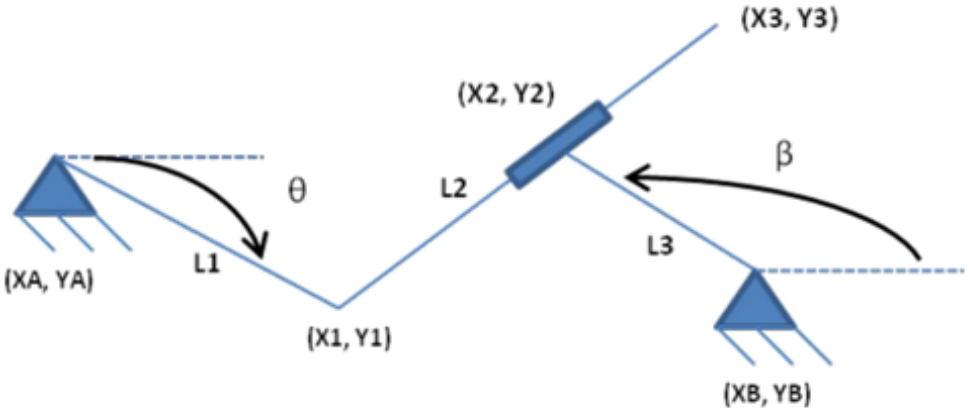


MECANISMO BIELA-MANIVELA INVERTIDA

PROBLEMA POSICIÓN

PASO 1: MODELADO DEL MECANISMO

El mecanismo de partida es el conocido como biela-manivela invertida. Podemos ver todas las variables en la figura.



PASO 2: GRADOS DE LIBERTAD

Los grados de libertad se calculan:

$G = 3·(n-1) - 2·PI - PII$

Siendo:

PI -> Numero de pares binarios de un grado de libertad

PII -> Número de pares binarios de dos grados de libertad. En este caso tendríamos:
\begin{eqnarray} n &=& 5 \\ P_{I} &=& 5 \\ P_{II} &=& 0 \end{eqnarray}

Por lo tanto:

$G = 3·(5-1) - 2·5 - 0 = 2$

PASO 3: DEFINICIÓN DEL VECTOR q

El vector $\mathbf{q}$ de dimensión $N \times 1$ contiene las coordenadas dependientes del mecanismo. Es decir, contiene las coordenadas que no se mantienen fijas ya que varían con el tiempo.

Lo hemos modelizado enpleando las 8 coordenadas:
\begin{equation} \mathbf{q} = \begin{bmatrix} x_1 \\ y_1 \\ x_2 \\ y_2 \\ x_3 \\ y_3 \\ \theta \\ \beta \end{bmatrix} \end{equation}

PASO 4: IMPLEMENTACIÓN EN PYTHON

Al igual que en otros entornos de programación, necesitamos añadir librerías que contengan las funciones que vamos a utilizar. Esto es necesario hacerlo al principio del código. Las que vamos a usar son las siguientes:

1. numpy -> Sirve para trabajar con arrays y matrices, ofreciendo una interfaz similar a los comandos en MATLAB.
2. math -> La utilizaremos para usar funciones matemáticas.
3. pprint -> "pretty print", su función es ayudar a depurar el código.
4. matplotlib.pyplot -> Es necesaria para dibujar gráficas.

In [1]:

```
#PASO 4
```

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

PASO 5: LECTURA DE DATOS

Los datos iniciales de los que partiremos para resolver este mecanismo mediante análisis cinemático por métodos numéricos son los parámetros constantes que definen el mecanismo, es decir, las variables que no cambian con el tiempo. En este caso serían las longitudes de las barras y las posiciones de los apoyos.

Además, como el mecanismo tiene dos grados de libertad, tenemos que escoger las **variables independientes** entre las componentes del vector q. En este caso hemos escogido los ángulos, por lo que también será un dato de partida.

1. Longitudes de las barras: $L_1, ~ L_2 ~ L_3 ~ y ~ L_4$.
2. Posición de los dos apoyos: $X_A, ~ Y_A, ~ X_B ~ e ~ Y_B$.
3. Ángulo que forma la primera barra respecto a la horizontal en radianes: $θ(t=0)$.
4. Ángulo que forma la cuarta barra respecto a la horizontal en radianes: $\beta(t=0)$

Una vez tengamos esos datos, definiremos una posición inicial.

In [2]:

```
print ('BIELA-MANIVELA INVERTIDA')
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["L3"] = float (input ('Introduce longitud L3: '))
meca["theta"] = float (input ('Introduce angulo inicial theta:')) #En rad
meca["beta"] = float (input ('Introduce angulo inicial beta: '))
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 ([[0.1], [0.2], [1], [0.2], [meca["XB"]+meca["L3"]*math.cos(meca["beta"])],
[meca["YB" ]+meca["L3"]*math.sin(meca["beta"])], [meca["theta"]], [meca["beta"]]])
print('q: ' + str(q))
```

```
BIELA-MANIVELA INVERTIDA
============================
Introduce longitud L1:2
Introduce longitud L2:3
Introduce longitud L3: 1
Introduce angulo inicial theta:0.5
Introduce angulo inicial beta: 0.8
Introduce coordenada en x del punto B:2
Introduce coordenada en y del punto B:0
q: [[0.1       ]
 [0.2        ]
 [1.         ]
 [0.2        ]
 [2.69670671]
 [0.71735609]
  [0.5       ]
  [0.8       ]]
```

PASO 6: MATRIZ DE RESTRICCIONES $\Phi(q)$

Este vector agrupa las ecuaciones de restricción y será de dimensión $mx1$.

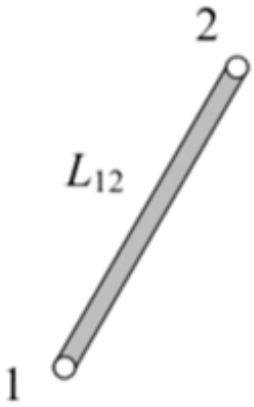
Estas ecuaciones podrían definirse empleando diferentes tipos de coordenadas: independientes, dependientes, relativas dependientes, de punto de referencia y naturales. Estas últimas son las que vamos a usar nosotros.

Para coordenadas naturales en el plano es necesario seguir un procedimiento:

1. Cada sólido debe tener al menos 2 puntos.
2. Cada par de rotación debe tener 1 punto.
3. Cada par prismático debe tener 3 puntos alineados.
4. Se pueden añadir tantos puntos adicionales como fuera necesario.
5. De los puntos mencionados, los fijos no entran en el vector q.

Para la formación de la matriz de restricciones, tenemos que tener en cuenta que hay restricciones de sólido rígido y de pares cinemáticos.

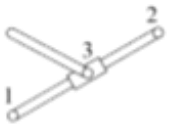
En este caso necesitamos:



Una única restricción:

$$(x_2 - x_1)^2 + (y_2 - y_1)^2 - L_{12}^2 = 0$$

También necesitamos imponer una condición para la corredera, que consiste en hacer el producto escalar entre 13 y 32. Se basa en imponer que los 3 puntos estén alineados.

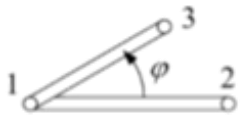


Colinealidad $\vec{12}$–$\vec{13}$:

$$\begin{aligned} \vec{12} \times \vec{13} &= 0 \\ (x_2 - x_1)(y_3 - y_1) - (y_2 - y_1)(x_3 - x_1) &= 0 \end{aligned}$$

Además, como el vector de coordenadas dependientes tiene 8 componentes tenemos que añadir una ecuación de restricción para cada ángulo.

Esta última ecuación depende de si el ángulo en cuestión es demasiado pequeño. Esto se debe a 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°, es el coseno el que se aproxima a 0, por lo que en esos casos la restricción a utilizar sería la del seno.

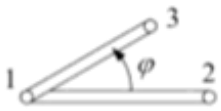


Se añade ϕ al vector $\mathbf{q}$.
Usando el producto escalar: $\rightarrow\vec{12} \cdot \vec{13} = L_{12}L_{13}\cos\phi$.
Usando el producto vectorial: $\rightarrow\vec{12} \times \vec{13} = L_{12}L_{13}\sin\phi$.

Podemos añadir una de las dos restricciones a $\mathbf{\Phi(q)}$, pero mejor ambas (evitar singularidades):

$$\begin{aligned} (x_2 - x_1)(x_3 - x_1) + (y_2 - y_1)(y_3 - y_1) &= L_{12}L_{13}\cos\phi \\ (x_2 - x_1)(y_3 - y_1) - (y_2 - y_1)(x_3 - x_1) &= L_{12}L_{13}\sin\phi \end{aligned}$$

$$(x_2 - x_1)(y_3 - y_1) - (y_2 - y_1)(x_3 - x_1) = L_{12}L_{13}\sin\phi$$



Cuando el ángulo ϕ (orientación *absoluta*) es relativo a tierra (12 es tierra), las ecuaciones se simplifican:

$$\begin{aligned} L_{12}(x_3 - x_1) + (y_2 - y_1)^{0}(y_3 - y_1) &= L_{12}L_{13}\cos\phi \\ L_{12}(y_3 - y_1) - (y_2 - y_1)^{0}(x_3 - x_1) &= L_{12}L_{13}\sin\phi \end{aligned}$$

$$\downarrow$$

$$\begin{aligned} x_3 - x_1 &= L_{13}\cos\phi \\ y_3 - y_1 &= L_{13}\sin\phi \end{aligned}$$

Para nuestro caso tendríamos para el ángulo θ:

$Si ~~ \cos (\theta) < {0.95} \rightarrow (X_1 - X_A) - L_1 · \cos(\theta)$

$Si ~~ \cos (\theta) > {0.95} \rightarrow (Y_1 - Y_A) - L_1 · \sin (\theta)$

Para el ángulo β:

$Si ~~ \cos (\beta) < {0.95} \rightarrow (X_3 - X_B) - L_3 · \cos(\beta)$

$Si ~~ \cos (\beta) > {0.95} \rightarrow (Y_3 - Y_B) - L_3 · \sin (\beta)$

La matriz quedaría:
\begin{equation} \mathbf{\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_3)^2 + (Y_B-Y_3)^2 - L_3^2 \\ (X_3-X_1)(Y_2-Y_1) - (X_2-X_1)(Y_3-Y_1) \\ \Phi(4)\\ \Phi(5)\\ \end{bmatrix} \end{equation}

In [3]:

```
#PASO 6

def Phi (q,meca):
    #Inicializa a cero Phi
    Phi = np.zeros((8,1))

    #Extraer coordenadas
    X1 = q[0]
    Y1 = q[1]
    X2 = q[2]
    Y2 = q[3]
    X3 = q[4]
    Y3 = q[5]
    theta = q[6]
    beta = q[7]


    Phi[0] = X1**2 + Y1**2 - meca["L1"]**2
    Phi[1] = (X2-X1)**2 + (Y2-Y1)**2 - meca["L2"]**2
    Phi[2] = (meca["XB"]-X3)**2 + (meca["YB"]-Y3)**2 - meca["L3"]**2
    Phi[3] = (X3-X1)*(Y2-Y1) - (X2-X1)*(Y3-Y1)

    if (abs(math.cos(theta)) < 0.95):
        Phi[4] = X1-meca["L1"]*math.cos(theta)
    else:
        Phi[4] = Y1-meca["L1"]*math.sin(theta)


    if (abs(math.cos(beta)) < 0.95):
        Phi[5] = (X3-meca["XB"])-meca["L3"]*math.cos(beta)
    else:
        Phi[5] = (Y3-meca["YB"])-meca["L3"]*math.sin(beta)
```

```
    return Phi
```

PASO 7: Matriz jacobiana Φ_q

Esta matriz de dimensiones mxn está compuesta por las derivadas parciales de las ecuaciones de restricción con respecto al vector de cooordeandas dependientes.

Por ejemplo endríamos:

Φ_q(0,0) = Derivada de Φ(0) respecto a X1. Φ_q(0,1) = Derivada de Φ (0) respecto a Y1. Φ_q(1,0) = Derivada de Φ(1) respecto a X1.

Tendríamos que construir la matriz elemento a elemento de esta manera.

Para la ecuación del ángulo hay que tener en cuenta que el jacobiano también tomará dos valores. Los posibles son:

1. $Si ~~ \cos (\theta) < {0.95}$

   $\Phi_q(4,0) = 1$ $\Phi_q(4,6) = L_1 · \sin(\theta)$

1. $Si ~~ \cos (\theta) > {0.95}$

   $\Phi_q(4,1) = 1$ $\Phi_q(4,6) = -L_1 · \cos(\theta)$

1. $Si ~~ \cos (\beta) < {0.95}$

   $\Phi_q(5,4) = 1$ $\Phi_q(5,7) = L_3 · \sin(\beta)$

1. $Si ~~ \cos (\beta) > \frac{1}{\sqrt{2}}$

   $\Phi_q(5,5) = 1$ $\Phi_q(5,7) = -L_3 · \cos(\beta)$

La matriz jacobiana sería:
\begin{equation} \mathbf{\Phi_q} = \begin{bmatrix} 2X_1 & 2Y_1 & 0 & 0 & 0 & 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 & 0\\ 0 & 0 & 0 & 0 & -2(X_B-X_3) & -2(Y_B-Y_3) & 0 & 0\\ -2(Y_2-Y_1) + (Y_3-Y_1) & (X_2-X_1)-(X_3-X_1) & -Y_3-Y_1 & X_3-X_1 & Y_2-Y_1 & -(X_2-X_1) & 0 & 0\\ \Phi_q(4,0) & \Phi_q(4,1) & 0 & 0 & 0 & 0 & \Phi_q(4,6) & \Phi_q(4,7)\\ 0 & 0 & 0 & 0 & \Phi_q(5,4) & \Phi_q(5,5) & \Phi_q(5,6) & \Phi_q(5,7)\\ \end{bmatrix} \end{equation}

In [4]:

```
# PASO 7

def jacob_Phiq(q,meca):
    #Inicializa a cero la matriz jacobiana
    Jacob = np.zeros((8,8))

    #Extraer coordenadas

    X1 = q[0]
    Y1 = q[1]
    X2 = q[2]
    Y2 = q[3]
    X3 = q[4]
    Y3 = q[5]
    theta = q[6]
    beta = q[7]

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

```
    Jacob[3,4] = (Y2-Y1)
    Jacob[3,5] = -(X2-X1)

    if (abs(math.cos(theta)) < 0.95 ):
        Jacob[4,6] = meca["L1"]*math.sin(theta)
        Jacob[4,0] = 1
    else:
        Jacob[4,6] = -meca["L1"]*math.cos(theta)
        Jacob[4,1] = 1


    if (abs(math.cos(beta)) < 0.95 ):
        Jacob[5,7] = meca["L3"]*math.sin(beta)
        Jacob[5,4] = 1
    else:
        Jacob[5,7] = -meca["L3"]*math.cos(beta)
        Jacob[5,5] = 1


    Jacob[6,6] = 1
    Jacob[7,7] = 1

    return Jacob
```

PASO 8: RESOLUCIÓN DEL PROBLEMA POSICIÓN

El objetivo es obtener los valores de las coordenadas dependientes del vector q.

Para resolverlo partimos de la ecuación:

$\Phi(q+Δq) = \Phi + \Phi_q · Δq = 0$

De donde despejamos:

$\Phi_q · Δq = -\Phi$

Esta ecuación se convertiría en $Ax=b$, siendo A el jacobiano y b Φ. Sin embargo, no se pueden dividir matrices de esa manera, por lo que tenemos que multiplicar a ambos lados de la igualdad por la izquierda por A^{-1}, es decir, por la inversa del jacobiano $Φq^{-1}$:

$ \underbrace{ \Phi q^-1 · Φq}_{= \mathbf{I_n}} · Δq = Φq^{-1} ·-Φ$

Por el lado izquierdo al multiplicar el jacobiano por su inversa toma el valor la unidad, por lo que quedaría:

$Δq = \Phi_q^{-1} · -Φ$

Y para terminar, tendríamos que el nuevo valor de q sería:

$q = q + Δq$

Hay que repetir este proceso hasta que el vector Φ se aproxime a 0, lo que indicaría la validez del vector q calculado. Sin embargo, hay datos iniciales para los que el mecanismo non converge. Por ejemplo, si decimos que $X_A=0, ~ X_A=0, ~ X_B=50, ~ Y_B=0 ~ y ~ L_1=1, ~ L_2=2, ~ L_3=3$, no existe tal posición. Por ello, tenemos que poner un límite de iteraciones, como por ejemplo 100, y si llega a dicho límite tendremos que el mecanismo no converge.

Para poder operar con las matricces de ese modo, es necesario que las dimensiones sean correctas. En este caso tendríamos que añadir una fila a la matriz Φ y a Φ_q. Como el mecanismo tiene dos grados de libertad tenemos que asignar un valor conocido a dos de las variables dependientes, como ya se explicó anteriormente. Esto se traduce en que para la matriz de restricciones $\Phi(6)=0$ y $\Phi(7)=0$, ya que su valor permanece invariable. Por otro lado para la matriz Φ_q tendríamos que para las variables cuyo valor esconocido sería uno y el resto 0. Es decir:

$\text{Φq(6)} \rightarrow (0, 0, 0, 0, 0, 0, 1, 0)$

$\text{Φq(7)} \rightarrow (0, 0, 0, 0, 0, 0, 0, 1)$

Otra forma de meidir el error en el cálculo de q es calculando el módulo del vector $Δq$. Tiene que ser lo más próximo a 0.

Por último, podemos saber si el mecanismo convergerá calculando el rango de la matriz jacobiana. Si su rango es igual al número de coordenadas dependientes, convergerá.

In [5]:

```
#PASO 8
```

```
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 ((8,1))
    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]
        X3 = q[4]
        Y3 = q[5]
        theta = q[6]
        beta = q[7]

        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 s
olucion

        rango = np.linalg.matrix_rank(J, 1e-5)

        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)
    if (error > tolerancia):
        raise Exception ('No se puede alcanzar la posición')
    return q

q = resuelve_prob_posicion(q,meca)

resuelve_prob_posicion(q,meca)
```

```
 q=
 array([[0.1       ],
        [0.2       ],
        [1.        ],
        [0.2       ],
        [2.69670671],
        [0.71735609],
        [0.5       ],
        [0.8       ]])
Phi=
array([[-3.95000000e+00],
       [-8.19000000e+00],
       [-1.11022302e-16],
       [-4.65620482e-01],
       [-1.65516512e+00],
       [-1.11022302e-16],
       [ 0.00000000e+00],
       [ 0.00000000e+00]])
 jacob=
 array([[ 0.2       ,  0.4       , 0.        ,  0.        ,  0.        ,
          0.        ,  0.        , 0.        ],
        [-1.8       , -0.        , 1.8       ,  0.        ,  0.        ,
```

```
          0.        ,  0.        , 0.        ],
        [ 0.        ,  0.        , 0.        ,   0.        , 1.39341342,
          1.43471218,  0.        , 0.        ],
        [ 0.51735609, -1.69670671, -0.51735609,  2.59670671,  0.       ,
         -0.9       ,  0.        , 0.        ],
        [ 1.        ,  0.        , 0.        ,   0.        ,  0.       ,
          0.        ,  0.95885108, 0.        ],
        [ 0.        ,  0.        , 0.        ,   0.        ,  1.       ,
          0.        ,  0.        , 0.71735609],
        [ 0.        ,  0.        , 0.        ,   0.        ,  0.       ,
          0.        ,  1.        , 0.        ],
        [ 0.        ,  0.        , 0.        ,   0.        ,  0.       ,
          0.        ,  0.        , 1.        ]])
rango=8

error iter1=
13.117322190851484
q=
array([[1.75516512],
       [9.24741744],
       [7.20516512],
       [7.1974805 ],
       [2.69670671],
       [0.71735609],
       [0.5       ],
       [0.8       ]])
Phi=
array([[ 8.45953339e+01],
       [ 2.49047414e+01],
       [-1.11022302e-16],
       [ 4.45587335e+01],
       [-4.44089210e-16],
       [-1.11022302e-16],
       [ 0.00000000e+00],
       [ 0.00000000e+00]])
jacob=
array([[   3.51033025,  18.49483488,   0.        ,  0.        ,
           0.        ,   0.        ,   0.        ,  0.        ],
       [-10.9       ,   4.09987387,  10.9       , -4.09987387,
           0.        ,   0.        ,   0.        ,  0.        ],
       [   0.        ,   0.        ,   0.        ,  0.        ,
           1.39341342,   1.43471218,   0.        ,  0.        ],
        [ -6.48012441,   4.50845841,   8.53006135,  0.94154159,
          -2.04993694,  -5.45      ,   0.        ,  0.        ],
       [   1.        ,   0.        ,   0.        ,  0.        ,
           0.        ,   0.        ,   0.95885108,  0.        ],
       [   0.        ,   0.        ,   0.        ,  0.        ,
           1.        ,   0.        ,   0.        ,  0.71735609],
       [   0.        ,   0.        ,   0.        ,  0.        ,
           0.        ,   0.        ,   1.        ,  0.        ],
       [   0.        ,   0.        ,   0.        ,  0.        ,
           0.        ,   0.        ,   0.        ,  1.        ]])
rango=8

error iter2=
6.887158717428214
q=
array([[1.75516512],
       [4.67341965],
       [4.90758032],
       [2.58959558],
       [2.69670671],
       [0.71735609],
       [0.5       ],
       [0.8       ]])
Phi=
array([[ 2.09214558e+01],
       [ 5.28004429e+00],
       [-1.11022302e-16],
       [ 1.05091478e+01],
       [-2.44249065e-15],
       [-1.11022302e-16],
       [ 0.00000000e+00],
       [ 0.00000000e+00]])
jacob=
array([[ 3.51033025,  9.34683929,  0.        ,  0.        ,  0.        ,
         0.        ,  0.        ,  0.        ],
```

```
        [-6.30483038,  4.16764813, 6.30483038, -4.16764813,  0.       ,
          0.        ,  0.        , 0.        ],
        [ 0.        ,  0.        , 0.        ,  0.        , 1.39341342,
          1.43471218,  0.        , 0.        ],
        [-1.87223949,  2.21087361, 3.95606356,  0.94154159, -2.08382407,
         -3.15241519,  0.        , 0.        ],
        [ 1.        ,  0.        , 0.        ,  0.        ,  0.       ,
          0.        ,  0.95885108, 0.        ],
        [ 0.        ,  0.        , 0.        ,  0.        ,  1.       ,
          0.        ,  0.        , 0.71735609],
        [ 0.        ,  0.        , 0.        ,  0.        ,  0.       ,
          0.        ,  1.        , 0.        ],
        [ 0.        ,  0.        , 0.        ,  0.        ,  0.       ,
          0.        ,  0.        , 1.        ]])
rango=8

error iter3=
3.308111175625341
q=
array([[1.75516512],
       [2.43507413],
       [4.04411707],
       [0.31191252],
       [2.69670671],
       [0.71735609],
       [0.5       ],
       [0.8       ]])
Phi=
array([[ 5.01019064e+00],
       [ 7.47116228e-01],
       [-1.11022302e-16],
       [ 1.93272909e+00],
       [ 6.66133815e-16],
       [-1.11022302e-16],
       [ 0.00000000e+00],
       [ 0.00000000e+00]])
jacob=
array([[  3.51033025,  4.87014826, 0.        ,  0.        ,  0.       ,
          0.        ,  0.        , 0.        ],
        [-4.57790388,  4.24632323, 4.57790388, -4.24632323,  0.       ,
          0.        ,  0.        , 0.        ],
       [  0.        ,  0.        , 0.        ,  0.        , 1.39341342,
          1.43471218,  0.        , 0.        ],
       [  0.40544357,  1.34741036, 1.71771804,  0.94154159, -2.12316161,
         -2.28895194,  0.        , 0.        ],
       [  1.        ,  0.        , 0.        ,  0.        ,  0.       ,
          0.        ,  0.95885108, 0.        ],
       [  0.        ,  0.        , 0.        ,  0.        ,  1.       ,
          0.        ,  0.        , 0.71735609],
       [  0.        ,  0.        , 0.        ,  0.        ,  0.       ,
          0.        ,  1.        , 0.        ],
       [  0.        ,  0.        , 0.        ,  0.        ,  0.       ,
          0.        ,  0.        , 1.        ]])
rango=8

error iter4=
1.2775534963251716
q=
array([[ 1.75516512],
       [ 1.40631887],
       [ 4.13793482],
       [-0.43975482],
       [ 2.69670671],
       [ 0.71735609],
       [ 0.5       ],
       [ 0.8       ]])
Phi=
array([[ 1.05833738e+00],
       [ 8.55794894e-02],
       [-1.11022302e-16],
       [-9.65155088e-02],
       [-4.44089210e-16],
       [-1.11022302e-16],
       [ 0.00000000e+00],
       [ 0.00000000e+00]])
jacob=
array([[ 3.51033025,  2.81263775,  0.        ,  0.        ,  0.       ,
```

```
          0.        ,  0.        , 0.        ],
        [-4.76553939,  3.69214738, 4.76553939, -3.69214738,  0.       ,
          0.        ,  0.        , 0.        ],
        [ 0.        ,  0.        , 0.        ,  0.        , 1.39341342,
          1.43471218,  0.        , 0.        ],
        [ 1.15711091,  1.44122811, 0.68896278,  0.94154159, -1.84607369,
         -2.3827697 ,  0.        , 0.        ],
        [ 1.        ,  0.        , 0.        ,  0.        ,  0.       ,
          0.        ,  0.95885108, 0.        ],
        [ 0.        ,  0.        , 0.        ,  0.        ,  1.       ,
          0.        ,  0.        , 0.71735609],
        [ 0.        ,  0.        , 0.        ,  0.        ,  0.       ,
          0.        ,  1.        , 0.        ],
        [ 0.        ,  0.        , 0.        ,  0.        ,  0.       ,
          0.        ,  0.        , 1.        ]])
rango=8

error iter5=
0.7035111410335451
q=
array([[ 1.75516512],
       [ 1.03003957],
       [ 4.64799798],
       [-0.13450511],
       [ 2.69670671],
       [ 0.71735609],
        [ 0.5       ],
        [ 0.8       ]])
 Phi=
 array([[ 1.41586117e-01],
       [ 7.24646222e-01],
       [-1.11022302e-16],
       [-1.91926212e-01],
       [ 0.00000000e+00],
       [-1.11022302e-16],
       [ 0.00000000e+00],
       [ 0.00000000e+00]])
jacob=
array([[  3.51033025,  2.06007913, 0.        ,  0.        ,  0.       ,
          0.        ,  0.        , 0.        ],
        [-5.78566571,  2.32908935, 5.78566571, -2.32908935,  0.       ,
          0.        ,  0.        , 0.        ],
       [  0.        ,  0.        , 0.        ,  0.        , 1.39341342,
          1.43471218,  0.        , 0.        ],
       [  0.8518612 ,  1.95129127, 0.31268347,  0.94154159, -1.16454467,
         -2.89283285,  0.        , 0.        ],
       [  1.        ,  0.        , 0.        ,  0.        ,  0.       ,
          0.        ,  0.95885108, 0.        ],
       [  0.        ,  0.        , 0.        ,  0.        ,  1.       ,
          0.        ,  0.        , 0.71735609],
       [  0.        ,  0.        , 0.        ,  0.        ,  0.       ,
          0.        ,  1.        , 0.        ],
       [  0.        ,  0.        , 0.        ,  0.        ,  0.       ,
          0.        ,  0.        , 1.        ]])
rango=8

error iter6=
0.3430150064301391
q=
array([[1.75516512],
       [0.96131108],
       [4.68488414],
       [0.19952348],
       [2.69670671],
       [0.71735609],
       [0.5       ],
       [0.8       ]])
Phi=
array([[ 4.72360464e-03],
       [ 1.63573853e-01],
       [-1.11022302e-16],
       [-2.53512998e-03],
       [ 0.00000000e+00],
       [-1.11022302e-16],
       [ 0.00000000e+00],
       [ 0.00000000e+00]])
jacob=
```

```
array([[  3.51033025,  1.92262216, 0.        ,  0.        ,  0.       ,
          0.        ,  0.        , 0.        ],
       [-5.85943803,  1.52357519, 5.85943803, -1.52357519,  0.       ,
          0.        ,  0.        , 0.        ],
       [  0.        ,  0.        , 0.        ,  0.        , 1.39341342,
          1.43471218,  0.        , 0.        ],
       [  0.51783261,  1.98817743, 0.24395499,  0.94154159, -0.7617876 ,
         -2.92971902,  0.        , 0.        ],
       [  1.        ,  0.        , 0.        ,  0.        ,  0.       ,
          0.        ,  0.95885108, 0.        ],
       [  0.        ,  0.        , 0.        ,  0.        ,  1.       ,
          0.        ,  0.        , 0.71735609],
       [  0.        ,  0.        , 0.        ,  0.        ,  0.       ,
          0.        ,  1.        , 0.        ],
       [  0.        ,  0.        , 0.        ,  0.        ,  0.       ,
          0.        ,  0.        , 1.        ]])
 rango=8
error iter7=
0.027583050868126353
q=
array([[1.75516512],
       [0.95885422],
       [4.66124815],
       [0.21352808],
       [2.69670671],
       [0.71735609],
       [0.5       ],
       [0.8       ]])
Phi=
array([[ 6.03613845e-06],
       [ 8.29639237e-04],
       [-1.11022302e-16],
       [ 5.80701982e-05],
       [ 0.00000000e+00],
       [-1.11022302e-16],
       [ 0.00000000e+00],
       [ 0.00000000e+00]])
jacob=
array([[  3.51033025,  1.91770845, 0.        ,  0.        ,  0.       ,
          0.        ,  0.        , 0.        ],
       [-5.81216606,  1.49065229, 5.81216606, -1.49065229,  0.       ,
          0.        ,  0.        , 0.        ],
       [  0.        ,  0.        , 0.        ,  0.        , 1.39341342,
          1.43471218,  0.        , 0.        ],
       [  0.50382801,  1.96454144, 0.24149813,  0.94154159, -0.74532614,
         -2.90608303,  0.        , 0.        ],
       [  1.        ,  0.        , 0.        ,  0.        ,  0.       ,
          0.        ,  0.95885108, 0.        ],
       [  0.        ,  0.        , 0.        ,  0.        ,  1.       ,
          0.        ,  0.        , 0.71735609],
       [  0.        ,  0.        , 0.        ,  0.        ,  0.       ,
          0.        ,  1.        , 0.        ],
       [  0.        ,  0.        , 0.        ,  0.        ,  0.       ,
          0.        ,  0.        , 1.        ]])
rango=8
error iter8=
0.00014751663810338254
q=
array([[1.75516512],
       [0.95885108],
       [4.66110172],
       [0.21351053],
       [2.69670671],
       [0.71735609],
       [0.5       ],
       [0.8       ]])
Phi=
array([[ 9.90763027e-12],
       [ 2.16506884e-08],
       [-1.11022302e-16],
       [ 4.60916638e-10],
       [ 0.00000000e+00],
       [-1.11022302e-16],
       [ 0.00000000e+00],
       [ 0.00000000e+00]])
```

```
jacob=
array([[  3.51033025,  1.91770215, 0.        ,   0.        ,  0.       ,
          0.        ,  0.        , 0.        ],
       [-5.81187319,  1.49068109, 5.81187319, -1.49068109,   0.       ,
          0.        ,  0.        , 0.        ],
       [  0.        ,  0.        , 0.        ,   0.        , 1.39341342,
          1.43471218,  0.        , 0.        ],
       [  0.50384556,  1.96439501, 0.24149499,  0.94154159, -0.74534054,
         -2.9059366 ,  0.        , 0.        ],
       [  1.        ,  0.        , 0.        ,   0.        ,  0.       ,
          0.        ,  0.95885108, 0.        ],
       [  0.        ,  0.        , 0.        ,   0.        ,  1.       ,
          0.        ,  0.        , 0.71735609],
       [  0.        ,  0.        , 0.        ,   0.        ,  0.       ,
          0.        ,  1.        , 0.        ],
       [  0.        ,  0.        , 0.        ,   0.        ,  0.       ,
          0.        ,  0.        , 1.        ]])
rango=8

error iter9=
3.6368558793514218e-09
q=
array([[1.75516512],
       [0.95885108],
       [4.66110172],
       [0.21351053],
       [2.69670671],
       [0.71735609],
       [0.5       ],
       [0.8       ]])
Phi=
array([[ 0.00000000e+00],
       [ 1.77635684e-15],
       [-1.11022302e-16],
       [ 0.00000000e+00],
       [ 0.00000000e+00],
       [-1.11022302e-16],
       [ 0.00000000e+00],
       [ 0.00000000e+00]])
jacob=
array([[  3.51033025,  1.91770215, 0.        ,   0.        ,  0.       ,
          0.        ,  0.        , 0.        ],
       [-5.81187318,  1.49068109, 5.81187318, -1.49068109,   0.       ,
          0.        ,  0.        , 0.        ],
       [  0.        ,  0.        , 0.        ,   0.        , 1.39341342,
          1.43471218,  0.        , 0.        ],
       [  0.50384556,  1.96439501, 0.24149499,  0.94154159, -0.74534054,
         -2.90593659,  0.        , 0.        ],
       [  1.        ,  0.        , 0.        ,   0.        ,  0.       ,
          0.        ,  0.95885108, 0.        ],
       [  0.        ,  0.        , 0.        ,   0.        ,  1.       ,
          0.        ,  0.        , 0.71735609],
       [  0.        ,  0.        , 0.        ,   0.        ,  0.       ,
          0.        ,  1.        , 0.        ],
       [  0.        ,  0.        , 0.        ,   0.        ,  0.       ,
          0.        ,  0.        , 1.        ]])
rango=8

error iter10=
3.1770830756044933e-16
q=
array([[1.75516512],
       [0.95885108],
       [4.66110172],
       [0.21351053],
       [2.69670671],
       [0.71735609],
       [0.5       ],
       [0.8       ]])
Phi=
array([[ 0.00000000e+00],
       [ 1.77635684e-15],
       [-1.11022302e-16],
       [ 1.11022302e-16],
       [ 0.00000000e+00],
       [-1.11022302e-16],
       [ 0.00000000e+00],
```

```
       [ 0.00000000e+00]])
jacob=
array([[  3.51033025,  1.91770215, 0.        ,   0.        ,  0.       ,
          0.        ,  0.        , 0.        ],
        [-5.81187318,  1.49068109, 5.81187318, -1.49068109,   0.       ,
          0.        ,  0.        , 0.        ],
       [ 0.        ,  0.        , 0.        ,   0.        , 1.39341342,
           1.43471218,  0.        , 0.        ],
       [  0.50384556,  1.96439501, 0.24149499,  0.94154159, -0.74534054,
          -2.90593659,  0.        , 0.        ],
       [ 1.        ,  0.        , 0.        ,   0.        ,  0.       ,
          0.        ,  0.95885108, 0.        ],
       [ 0.        ,  0.        , 0.        ,   0.        ,  1.       ,
          0.        ,  0.        , 0.71735609],
       [ 0.        ,  0.        , 0.        ,   0.        ,  0.       ,
          0.        ,  1.        , 0.        ],
       [ 0.        ,  0.        , 0.        ,   0.        ,  0.       ,
          0.        ,  0.        , 1.        ]])
rango=8
error iter1=
3.395998511097105e-16
```

Out[5]:

```
 array([[1.75516512],
       [0.95885108],
       [4.66110172],
       [0.21351053],
       [2.69670671],
       [0.71735609],
       [0.5       ],
       [0.8       ]])
```

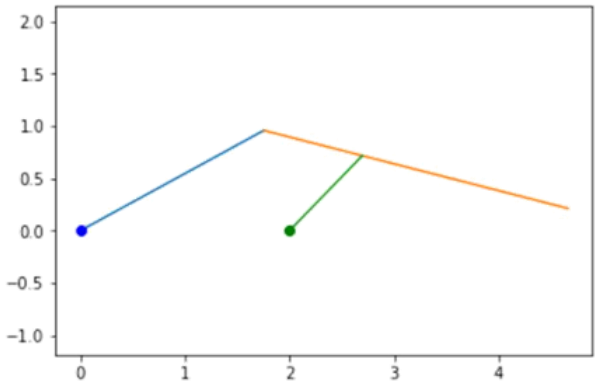
PASO 9: Dibujar el mecanismo

Para dibujar el mecanismo, definimos un cuadro de dibujo con los ejes de la misma dimensión. Seguidamente, dibujamos cada barra por separado. Para dibujar cada barra tendríamos que indicar las posiciones inicial y final, yendo por un lado las coordenadas en el eje X y por otro las coordenadas en el eje Y. Es decir, sería por ejemplo:

$Barra ~ 1 \rightarrow ([X_A, X_1], [Y_A, Y_1])$

In [6]:

```
# PASO 9

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]
    beta = q[7]


    plt.axis('equal')


    plt.plot ([meca["XA"], X1], [meca["YA"], Y1])    #[pos inicial (x1,x2), pos final (y1,y2)]
    plt.plot ([X1, X2], [Y1, Y2])
    plt.plot ([X3, meca["XB"]], [Y3, meca ["YB"]])

    plt.plot(meca["XA"], meca["YA"], 'bo')
    plt.plot(meca["XB"], meca["YB"], 'go')

    plt.show()#block=False)
    return

dibuja_mecanismo(q,meca)
```


2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
0
1
2
3
4

MECANISMO BIELA-MANIVELA INVERTIDA

PROBLEMAS VELOCIDAD Y ACELERACIÓN

PASO 1: MATRIZ JACOBIANA

Para resolver el problema velocidad, necesitamos otra vez la matriz jacobiana. El método de construcción aparece detallado en el notebook Problema_Posición_5B, por lo que para este caso copiaremos el código de los pasos realizados para poder conseguirla.

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
#%matplotlib inline (Para notebook)



print ('BIELA-MANIVELA INVERTIDA')
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["L3"] = float (input ('Introduce longitud L3: '))
meca["theta"] = float (input ('Introduce angulo inicial theta:')) #En rad
meca["beta"] = float (input ('Introduce angulo inicial beta: '))
meca["XB"] = float (input ('Introduce coordenada en x del punto B:'))
meca["YB"] = 0 #float (input ('Introduce coordenada en y del punto B:'))
meca["XA"] = 0
meca["YA"] = 0

# Defino posicion inicial:
q = np.array ([[0.1], [0.2], [1], [0.2], [meca["XB"]+meca["L3"]*math.cos(meca["beta"])],
[meca["YB" ]+meca["L3"]*math.sin(meca["beta"])], [meca["theta"]], [meca["beta"]]])
print('q: ' + str(q))

# JACOBIANO
def jacob_Phiq(q,meca):
    #Inicializa a cero la matriz jacobiana
    Jacob = np.zeros((8,8))

    #Extraer coordenadas

    X1 = q[0]
    Y1 = q[1]
    X2 = q[2]
    Y2 = q[3]
    X3 = q[4]
    Y3 = q[5]
    theta = q[6]
    beta = q[7]

    #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[2,4] = -2*(meca["XB"]-X3)
```

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

    if (abs(math.cos(theta)) < 0.95 ):
        Jacob[4,6] = meca["L1"]*math.sin(theta)
        Jacob[4,0] = 1
    else:
        Jacob[4,6] = -meca["L1"]*math.cos(theta)
        Jacob[4,1] = 1


    if (abs(math.cos(beta)) < 0.95 ):
        Jacob[5,7] = meca["L3"]*math.sin(beta)
        Jacob[5,4] = 1
    else:
        Jacob[5,7] = -meca["L3"]*math.cos(beta)
        Jacob[5,5] = 1


    Jacob[6,6] = 1
    Jacob[7,7] = 1

    return Jacob


def Phi (q,meca):
    #Inicializa a cero Phi
    Phi = np.zeros((8,1))

    #Extraer coordenadas
    X1 = q[0]
    Y1 = q[1]
    X2 = q[2]
    Y2 = q[3]
    X3 = q[4]
    Y3 = q[5]
    theta = q[6]
    beta = q[7]

    Phi[0] = X1**2 + Y1**2 - meca["L1"]**2
    Phi[1] = (X2-X1)**2 + (Y2-Y1)**2 - meca["L2"]**2
    Phi[2] = (meca["XB"]-X3)**2 + (meca["YB"]-Y3)**2 -
    meca["L3"]**2 Phi[3] = (X3-X1)*(Y2-Y1) - (X2-X1)*(Y3-Y1)
    if (abs(math.cos(theta)) < 0.95):
        Phi[4] = X1-meca["L1"]*math.cos(theta)
    else:
        Phi[4] = Y1-meca["L1"]*math.sin(theta)


    if (abs(math.cos(beta)) < 0.95):
        Phi[5] = (X3-meca["XB"])-meca["L3"]*math.cos(beta)
    else:
        Phi[5] = (Y3-meca["YB"])-meca["L3"]*math.sin(beta)


    return Phi



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 ((8,1))
    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]
        X3 = q[4]
        Y3 = q[5]
        theta = q[6]
        beta = q[7]

        fi=Phi(q,meca)

        J = jacob_Phiq(q,meca)

        rango = np.linalg.matrix_rank(J, 1e-5)

        deltaQ = np.linalg.solve(J,-fi)
        q = q + deltaQ
        error = np.linalg.norm(deltaQ) # El error es el modulo del
        vector i=i+1
    if (error > tolerancia):
        raise Exception ('No se puede alcanzar la posición')

    return q

q = resuelve_prob_posicion(q,meca)
```

```
BIELA-MANIVELA INVERTIDA
============================
Introduce longitud L1:2
Introduce longitud L2:3
Introduce longitud L3: 1
Introduce angulo inicial theta:0.5
Introduce angulo inicial beta: 0.8
Introduce coordenada en x del punto B:2
q: [[0.1       ]
 [0.2       ]
 [1.        ]
 [0.2       ]
 [2.69670671]
 [0.71735609]
 [0.5       ]
 [0.8       ]]
```

PASO 2: PROBLEMA VELOCIDAD

Consiste en determinar las velocidades de todas las variables del mecanismo una vez sabemos su posición q y la velocidad de los grados de libertad.

Partimos de la ecuación:

$\Phi_q =0 $

Derivando se obtiene:

$\Phi_q \dot{q} + \Phi_t = 0$

Siendo $\dot{q}$ el vector velocidad, Φ_q el jacobiano y Φ_t la derivada parcial de las ecuaciones de restricción respecto al tiempo. Para las ecuaciones de sólido rígido el valor de esta derivada es 0. Solo tendría un valor no nulo la correspondiente al ángulo, que en ese caso tendría la velocidad que nosotros le indiquemos.

De este modo la expresión quedaría:

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

Este sistema de ecuaciones tiene infinitas soluciones y por tanto hay que ampliar añadiendo un dato conocido de velocidad, lo que se hace añadiendo una fila a la matriz de coeficientes del lado izquierdo y un dato a la columna del lado derecho de la ecuación por

cada grado de libertad.

De esta forma llegamos a un sistema de ecuaciones lineal matricial de la forma:

$\mathbf{A} \mathbf{x} = \mathbf{b}$

Habría que multiplicar en ambas partes de la igualdad por la A invertida en el lado izquierdo, del mismo modo que se hizo en el problema de posición. De esta manera quedaría:

$x = A^{-1} b$

In [2]:

```
#PASO 2

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 ((8,1))
#Velocidad del gdl. En una vuelta completa del angulo se cumple angulo=2*Pi*t
qp[6]=1
qp[7]=2
qp = resuelve_prob_velocidad (q,qp, meca)
qp
```

Out[2]:

```
array([[-0.95885108],
       [ 1.75516512],
       [-1.31819903],
       [ 0.35413793],
       [-1.43471218],
       [ 1.39341342],
        [ 1.        ],
        [ 2.        ]])
```

PASO 3: PROBLEMA ACELERACIÓN

El problema aceleración trata de determinar las aceleraciones de todas las variables del mecanismo, conociendo la posiciónq, la velocidad $\dot{q}$ y las aceleraciones de los grados de libertad.

Partimos la ecuación que se obtiene tras derivar la ecuación inicial para el problema de velocidad, es decir:

$\Phi_q \dot{q} + \Phi_t = 0$

Se deriva por segunda vez:

$\dot{\Phi}_q \dot{q} + \Phi_q \ddot{q} + \dot{\Phi}_t = 0$

Despejamos $\Phi_q \ddot{q}$:

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

Siendo Φ_q el jacobiano, $\ddot{q}$ el vector aceleración, $\dot{q}$ el vector velocidad, $\dot{\Phi}_q$ la derivada del jacobiano respecto al tiempo y $\dot{\Phi}_t$ es la derivada de las ecuaciones de restricción con respecto al tiempo, cuyo valor es nulo. Es decir, tendríamos:

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

Del mismo modo que en el problema velocidad, llamando b al conjunto formado por $\dot{\Phi}_q \dot{q}$ llegamos a un sistema de ecuaciones lineal matricial:

$ \mathbf{A} \mathbf{x} = \mathbf{b}$

Y despejando la x:

$\mathbf{x} = \mathbf{A}^{-1} \mathbf{b}$

El vector velocidad será:

$$\mathbf{q_p} = \begin{bmatrix} \dot{X}_1 \\ \dot{Y}_1 \\ \dot{X}_2 \\ \dot{Y}_2 \\ \dot{X}_3 \\ \dot{Y}_3 \\ \dot{\theta}\\ \dot{\beta} \\ \end{bmatrix}$$

Por otro lado, para calcular la derivada del jacobiano solo tenemos en cuenta las filas que hacen referencia a las ecuaciones de las coordenadas independientes, ya que la última que añadimos para poder realizar los cálculos era adicional. Teniendo en cuenta esto, la derivada del jacobiano sería:

$$\mathbf{\dot{\Phi}_q} = \begin{bmatrix} 2 \dot{X}_1 & 2 \dot{Y}_1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -2(\dot{X}_2 - \dot{X}_1) & -2(\dot{Y}_2 - \dot{Y}_1) & 2(\dot{X}_2 - \dot{X}_1) & 2(\dot{Y}_2 - \dot{Y}_1) & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 0 & 0 & 2\dot{X}_3 & 2\dot{Y}_3 \\ -(\dot{Y}_2 - \dot{Y}_1) + (\dot{Y}_3 - \dot{Y}_1) & (\dot{X}_2 - \dot{X}_1) - (\dot{X}_3 - \dot{X}_1) & -\dot{Y}_3 + \dot{Y}_1 & \dot{X}_3 - \dot{X}_1 & \dot{Y}_2 - \dot{Y}_1 & -(\dot{X}_2 - \dot{X}_1) & 0 & 0\\ \dot{\theta}_q(4) \\ \dot{\theta}_q(5)\\ \end{bmatrix}$$

Siendo:

1. $Si ~~ \cos (\theta) < {0.95} \rightarrow \dot{\Phi}_q(4) = \dot{\theta} · L_1 · \cos(\theta)$
2. $Si ~~ \cos (\theta) > {0.95} \rightarrow \dot{\Phi}_q(4) = \dot{\theta} · L_1 · \sin(\theta)$
3. $Si ~~ \cos (\beta) < {0.95} \rightarrow \dot{\Phi}_q(5) = \dot{\beta} · L_3 · \cos(\beta)$
4. $Si ~~ \cos (\beta) > {0.95} \rightarrow \dot{\Phi}_q(5) = \dot{\beta} · L_3 · \sin(\beta)$

Como ya tenemos $\dot{\Phi}_q$ y $\dot{q}$, podemos calcular b. Las dos últimas filas que añadimos son los valores de las aceleraciones angulares, datos que sabemos de antemano.

$$\mathbf{b} = \begin{bmatrix} 2\dot{X}_1^2 + 2\dot{Y}_1^2 & 0 & 0 & 0 & 0 & 0 & 0 \\ -2\dot{X}_1 (\dot{X}_2-\dot{X}_1) + -2\dot{Y}_1 (\dot{Y}_2-\dot{Y}_1) & 2\dot{X}_2(\dot{X}_2-\dot{X}_1) + 2\dot{Y}_2(\dot{Y}_2-\dot{Y}_1) & 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 2\dot{X}_3^2 + 2\dot{Y}_3^2 & 0 & 0 \\ -\dot{X}_1((\dot{Y}_2-\dot{Y}_1)+(\dot{Y}_3-\dot{Y}_1)) + \dot{Y}_1((\dot{X}_2-\dot{X}_1)-(\dot{X}_3-\dot{X}_1)) & \dot{X}_2(-\dot{Y}_3-\dot{Y}_1) + \dot{Y}_2(\dot{X}_3-\dot{X}_1) + \dot{X}_3(\dot{Y}_2-\dot{Y}_1) + \dot{Y}_3(-(\dot{X}_2-\dot{X}_1)) & 0 & 0\\ b(4) \\ b(5)\\ 1\\ 1\\ \end{bmatrix}$$

1. $Si ~~ \cos (\theta) < {0.95} \rightarrow b(4) = \dot{\theta}^2 · L_1 · \cos(\theta)$
2. $Si ~~ \cos (\theta) > {0.95} \rightarrow b(4) = \dot{\theta}^2 · L_1 · \sin(\theta)$
3. $Si ~~ \cos (\beta) < {0.95} \rightarrow b(5) = \dot{\beta}^2 · L_3 · \cos(\beta)$
4. $Si ~~ \cos (\beta) > {0.95} \rightarrow b(5) = \dot{\beta}^2 · L_3 · \sin(\beta)$

In [3]:

```
#PASO 3

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]
    beta = q[7]
    #Extraer las velocidades
    X1q = qp[0]
    Y1q = qp[1]
    X2q = qp[2]
    Y2q = qp[3]
    X3q = qp[4]
    Y3q = qp[5]
    thetaq = qp[6]
    betaq = qp[7]
    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*X3q**2 + 2*Y3q**2
    b[3] = -2*X1q*((Y2q-Y1q)+(Y3q-Y1q)) + Y1q*((X2q-X1q)-(X3q-X1q)) + X2q*(-Y3q-Y1q) + Y2q*(X3q-X1q)
+ X3q*(Y2q-Y1q) -Y3q*(X2q-X1q)

    if (abs(math.cos(theta)) < 0.95 ):
        b[4] = thetaq**2 * (meca["L1"] * math.cos(theta))
    else:
        b[4] = thetaq**2 * (meca["L1"] * math.sin(theta))

    if (abs(math.cos(beta)) < 0.95 ):
        b[5] = betaq**2 * (meca["L3"] * math.cos(beta))
    else:
        b[5] = betaq**2 * (meca["L3"] * math.sin(beta))
```

```
    b[6] = 1 #Aceleracion conocida
    b[7] = 1
    qpp = np.linalg.solve(-jacob_Phiq(q,meca),b)


    return qpp


qpp=np.zeros ((8,1))
qpp = resuelve_prob_aceleracion (q,qp,qpp,meca)
qpp
```

Out[3]:

```
array([[-0.79631405],
       [-2.7140162 ],
       [-2.73535687],
       [-9.3771278 ],
       [-2.06947075],
       [-3.56613107],
       [-1.        ],
       [-1.        ]])
```

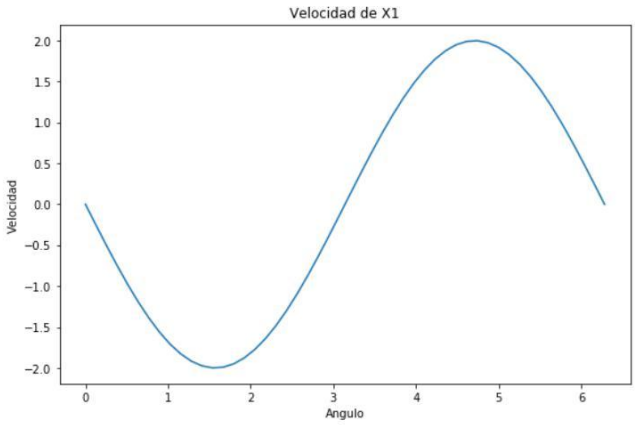
PASO 4: GRÁFICAS DE VELOCIDADES

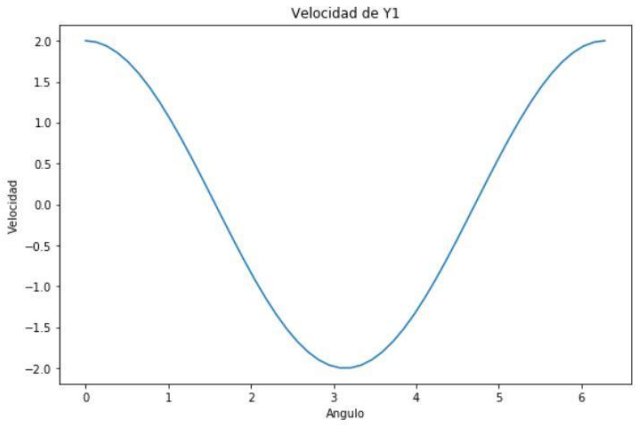
Vamos a representar por separado la gráfica de la velocidad en cada coordenada $(X_1, ~ Y_1, ~ X_2, ~ Y_2, ~ X_3 ~ e ~ Y_3)$.

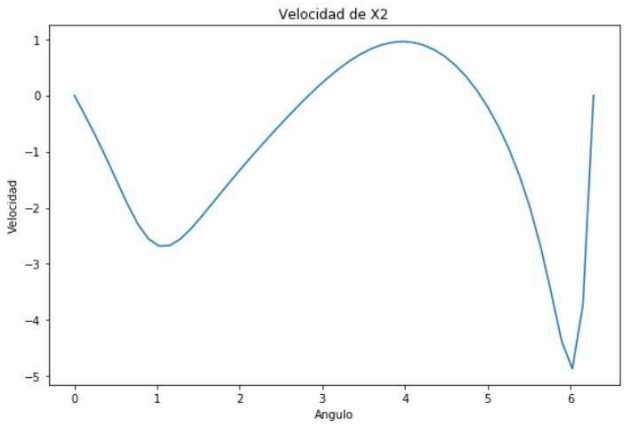
In [4]:

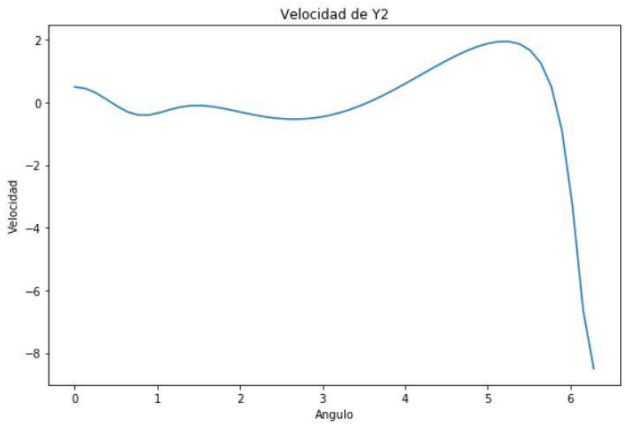
```
#PASO 4: GRÁFICAS DE VELOCIDADES
def grafica_velocidad(q,meca):

    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))
    VX3 = np.zeros((50,0))
    VY3 = np.zeros((50,0))

    i=0
    for t in th:

        omega1=1
        omega2=1.5
        q[6] = t*omega1
        q[7] = t*omega2

        q = resuelve_prob_posicion (q,
        meca) qp = np.zeros ((8,1))
        #Velocidad del gdl. En una vuelta completa del angulo se cumple
        angulo=2*Pi*t qp[6]=omega1
        qp[7]=omega2
        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)
```

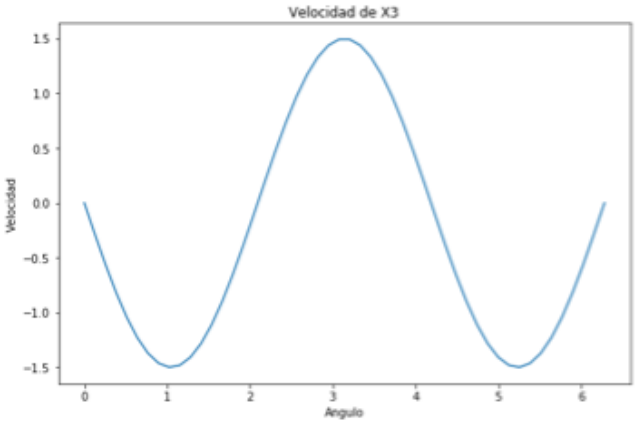
```
    plt.plot(th,VY1)
    plt.xlabel ('Angulo')
    plt.ylabel ('Velocidad')
    plt.title ('Velocidad de Y1')

    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)
```

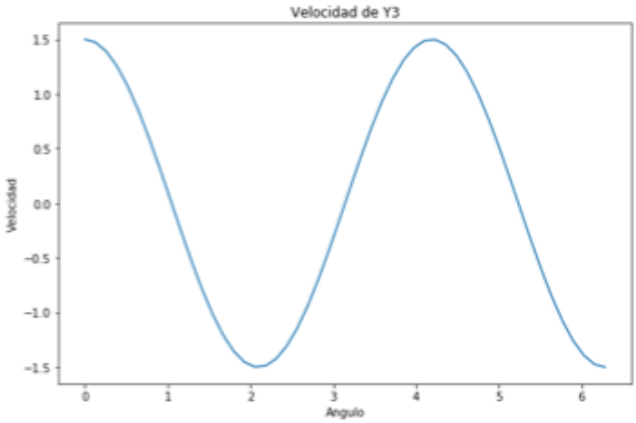












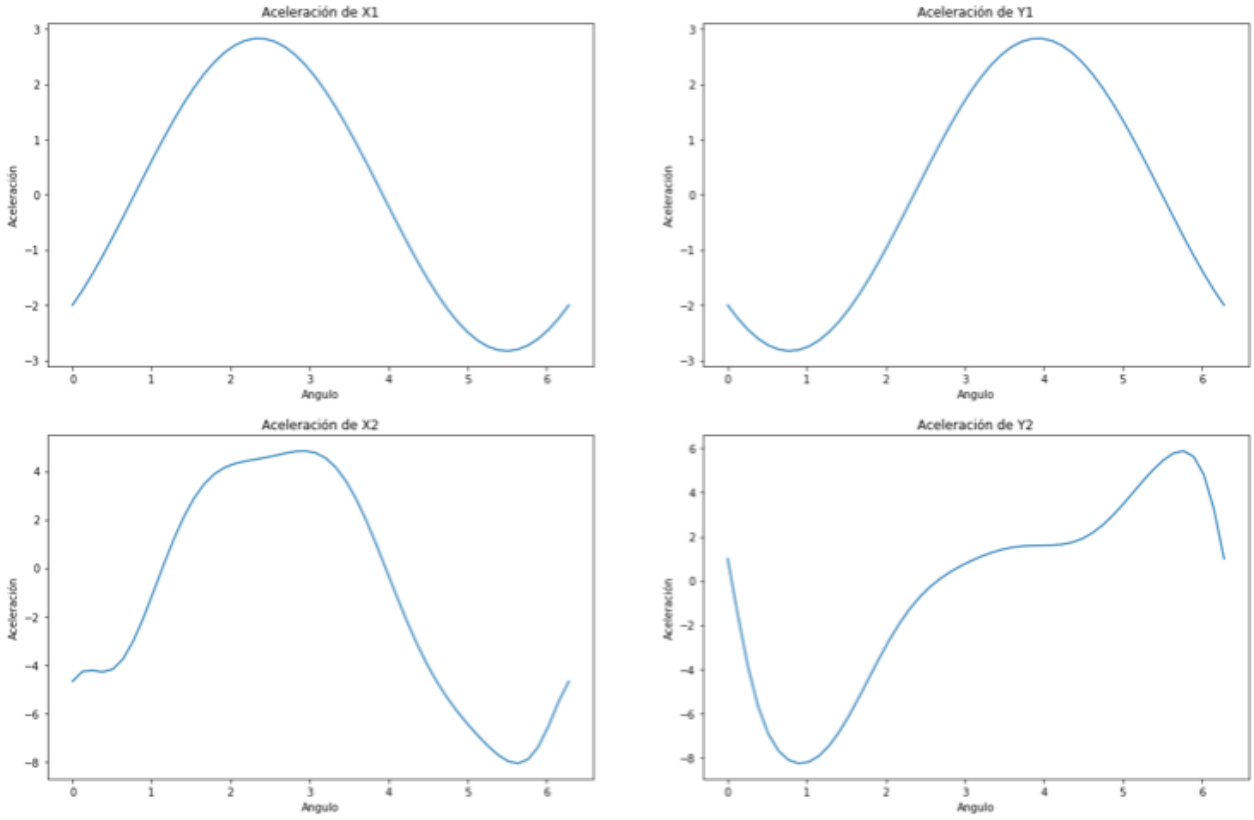
PASO 5: GRÁFICAS ACELERACIONES

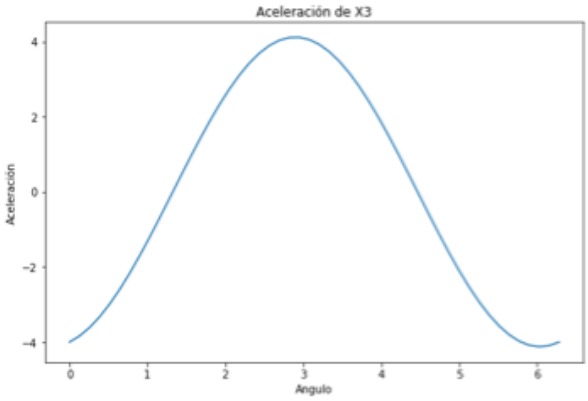
Haremos el mismo procedimiento que para la velocidad, representando en celdas separadas la aceleración de cada coordenada.

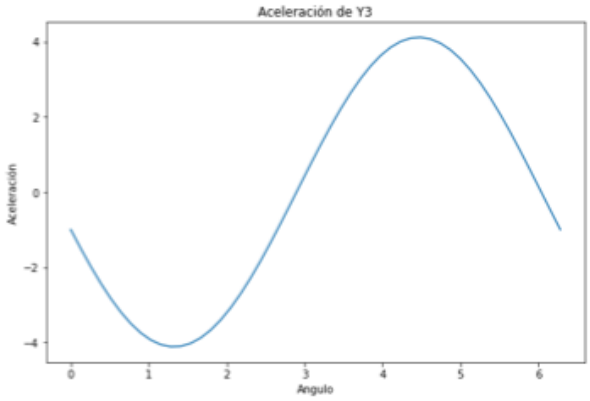
In [5]:

```
#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[7] = t
        q = resuelve_prob_posicion (q,meca)

        qp = np.zeros((8,1))
        qp[6] = 1 #inicializar qp en 0 con qp[6] = 1
        rad/s qp[7] = 2
        qp = resuelve_prob_velocidad(q, qp, meca)

        qpp = np.zeros((8,1))
        qpp[6] = 1 #inicializar qp en 0 con qpp[4] = 1
        rad/s**2 qpp[7] = 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)
```




Aceleración de X3
Aceleración
Angulo
4
2
0
−2
−4
0
1
2
3
4
5
6


Aceleración de Y3
Aceleración
Angulo
4
2
0
−2
−4
0
1
2
3
4
5
6

MECANISMO BIELA-MANIVELA INVERTIDA

ANIMACIÓN

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
#%matplotlib inline (Para notebook)



print ('BIELA-MANIVELA INVERTIDA')
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["L3"] = float (input ('Introduce longitud L3: '))
meca["theta"] = float (input ('Introduce angulo inicial theta:')) #En rad
meca["beta"] = float (input ('Introduce angulo inicial beta: '))
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 ([[0.1], [0.2], [1], [0.2], [meca["XB"]+meca["L3"]*math.cos(meca["beta"])],
[meca["YB" ]+meca["L3"]*math.sin(meca["beta"])], [meca["theta"]], [meca["beta"]]])
print('q: ' + str(q))

# JACOBIANO
def jacob_Phiq(q,meca):
    #Inicializa a cero la matriz jacobiana
    Jacob = np.zeros((8,8))

    #Extraer coordenadas

    X1 = q[0]
    Y1 = q[1]
    X2 = q[2]
    Y2 = q[3]
    X3 = q[4]
    Y3 = q[5]
    theta = q[6]
    beta = q[7]

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

```
    Jacob[3,5] = -(X2-X1)

    if (abs(math.cos(theta)) < 0.95 ):
        Jacob[4,6] = meca["L1"]*math.sin(theta)
        Jacob[4,0] = 1
    else:
        Jacob[4,6] = -meca["L1"]*math.cos(theta)
        Jacob[4,1] = 1


    if (abs(math.cos(beta)) < 0.95 ):
        Jacob[5,7] = meca["L3"]*math.sin(beta)
        Jacob[5,4] = 1
    else:
        Jacob[5,7] = -meca["L3"]*math.cos(beta)
        Jacob[5,5] = 1


    Jacob[6,6] = 1
    Jacob[7,7] = 1

    return Jacob


def Phi (q,meca):
    #Inicializa a cero Phi
    Phi = np.zeros((8,1))

    #Extraer coordenadas
    X1 = q[0]
    Y1 = q[1]
    X2 = q[2]
    Y2 = q[3]
    X3 = q[4]
    Y3 = q[5]
    theta = q[6]
    beta = q[7]


    Phi[0] = X1**2 + Y1**2 - meca["L1"]**2
    Phi[1] = (X2-X1)**2 + (Y2-Y1)**2 - meca["L2"]**2
    Phi[2] = (meca["XB"]-X3)**2 + (meca["YB"]-Y3)**2 -
    meca["L3"]**2 Phi[3] = (X3-X1)*(Y2-Y1) - (X2-X1)*(Y3-Y1)
    if (abs(math.cos(theta)) < 0.95):
        Phi[4] = X1-meca["L1"]*math.cos(theta)
    else:
        Phi[4] = Y1-meca["L1"]*math.sin(theta)

    if (abs(math.cos(beta)) < 0.95):
        Phi[5] = (X3-meca["XB"])-meca["L3"]*math.cos(beta)
    else:
        Phi[5] = (Y3-meca["YB"])-meca["L3"]*math.sin(beta)


    return Phi



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 ((8,1))
    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]
        X3 = q[4]
        Y3 = q[5]
        theta = q[6]
        beta = q[7]

        fi=Phi(q,meca)

        J = jacob_Phiq(q,meca)

        rango = np.linalg.matrix_rank(J, 1e-5)

        deltaQ = np.linalg.solve(J,-fi)
        q = q + deltaQ
        error = np.linalg.norm(deltaQ) # El error es el modulo del
        vector i=i+1
    if (error > tolerancia):
        raise Exception ('No se puede alcanzar la posición')
    return q

q = resuelve_prob_posicion(q,meca)
```

```
BIELA-MANIVELA INVERTIDA
============================
Introduce longitud L1:2
Introduce longitud L2:3
Introduce longitud L3: 1
Introduce angulo inicial theta:0.5
Introduce angulo inicial beta: 0.8
Introduce coordenada en x del punto B:2
Introduce coordenada en y del punto B:0
q: [[0.1       ]
 [0.2       ]
 [1.        ]
 [0.2       ]
 [2.69670671]
 [0.71735609]
 [0.5       ]
 [0.8       ]]
```

In [2]:

```
%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*3.14159/200 # vel. angular
    q[6] = i*omega
    q[7] = 2*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]
    X3 = q[4]
    Y3 = q[5]
    theta = q[6]
    beta = q[7]

    x=[meca["XA"], X1, X2, X3, meca["XB"]]
    y=[meca["YA"], Y1, Y2, Y3, meca["YB"]]



    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())
```

Out[2]: