UNIVERSIDAD DEL VALLE DE GUATEMALA
Facultad de Ingeniería

Desarrollo de simulador y analizador cinético de mecanismos de n barras de un grado de libertad

Trabajo de graduación presentado por Fernando José Lavarreda Urizar
para optar al grado académico de Licenciado en Ingeniería Mecánica

Guatemala,
2023

UNIVERSIDAD DEL VALLE DE GUATEMALA
Facultad de Ingeniería

Desarrollo de simulador y analizador cinético de mecanismos de n barras de un grado de libertad

Trabajo de graduación presentado por Fernando José Lavarreda Urizar
para optar al grado académico de Licenciado en Ingeniería Mecánica

Guatemala,
2023

Vo.Bo. Asesor

MBA Gustavo Adolfo Barrera Noriega

Vo.Bo. Terna Examinadora

MBA Gustavo Adolfo Barrera Noriega

Ing. Luis Diego Castañeda Fernández

Inga. Laura María Roldan Reyes

Fecha de aprobación de examen de graduación: Guatemala, 05 de diciembre 2023.

Índice

Lista de figuras

Lista de cuadros

Resumen

Se desarrollará un *software* que permitirá realizar un análisis cinético y cinemático de eslabones de geometría compleja para distintos mecanismos haciendo uso de ecuaciones de posición para mecanismos de n barras compuestos por lazos vectoriales de cuatro eslabones y de un grado de libertad; además, se encontrarán los esfuerzos de von Mises equivalentes a los cuales están sometidos los eslabones sin tomar en cuenta la concentración de esfuerzos haciendo un análisis por secciones de cada eslabón. De esta manera se espera crear una herramienta que facilite el proceso de diseño y estudio de máquinas proveyendo resultados de esfuerzos y simulaciones de trayectorias de eslabones reemplazando cálculos a mano por métodos numéricos en computadora. Esto permitirá dimensionar los componentes para evitar posibles colisiones y fallos por resistencia de materiales. Se busca proveer una alternativa gratuita, que requiera de pocos recursos de cómputo y de código abierto a los *softwares* disponibles actualmente en el mercado. Se creó una aplicación que permite el análisis de máquinas a través de una interfaz gráfica para la cual se codificó adicionalmente una herramienta para integrarla con Inventor®. Se utilizó el programa para calcular aceleraciones, fuerzas y esfuerzos equivalentes de von Mises a través de ecuaciones de mecánica de sólidos obteniedo un porcentaje de error menor al 10 % respecto a una simulación elaborada en ANSYS®.

CAPÍTULO 1

Introducción

El análisis de posición de mecanismos consiste en encontrar las posiciones, velocidades y aceleraciones de los eslabones que conforman un mecanismo. Esto se puede realizar a través de un método gráfico o un método analítico. La metodología gráfica involucra un proceso manual y lento ya que cada una de las soluciones es independiente la una de la otra; mientras que, la metodología analítica permite resolver n cantidad de posiciones de manera veloz al emplear una computadora. La desventaja de un análisis analítico se encuentra en la complejidad para plantear las ecuaciones que definen las posiciones del mecanismo.

Con esta información en mente se decidió resolver mecanismos de n barras compuestos por lazos vectoriales de cuatro barras al tratarse de un problema que ha sido estudiado de manera exhaustiva y al ser un mecanismo presente en una gran multitud de máquinas como pistones, bicicletas, bombas, entre otros. Al contar con mecanismos de cuatro barras las soluciones pueden concatenarse y resolverse con un único eslabón de entrada, de ahí que sea de un grado de libertad.

El análisis de esfuerzos es una herramienta curcial para el desarrollo de cualquier máquina para asegurarse de que esta no falle mediante el cálculo del factor de seguridad. Con los valores de esfuerzos el diseñador es capaz de seleccionar el material adecuado para satisfacer las necesidades de funcionamiento establecidas.

Existen múltiples metodologías propuestas para la solución de mecanismos de n barras. Estas incluyen la definición de los eslabones en el plano complejo resolviendo una serie de ecuaciones polinomiales (Wampler, 1999)y la resolución de matrices únicamente con aritmética real (Nielsen & Roth, 1999).

Mientras que, para obtener los esfuerzos de los eslabones también existen una serie de metodologías desarrolladas que van desde hallar los autovalores de una matriz (Imam et al., 1973) hasta expresiones analíticas concretas presentadas para mecanismos como el manivela corredera(Yang et al., 1981).

Existen herramientas disponibles en el mercado que cumplen las funciones propuestas;

no obstante, estas suelen ser costosas, requieren de muchos recursos computacionales y/o no tienen todas las funcionalidades propuestas. Algunos ejemplos son GeoGebra®, Working Model® y Solidworks®.

Al momento de desarrollar la aplicación se descubrió que su uso y adopción por estudiantes de Ingeniería Mecánica aumentaría si estuviese integrada con una herramienta que les es familiar como un *software* de modelado 3D, es por ello que se implementó una extensión que permite el trazo de eslabones en Inventor®.

Los mecanimos son dispositivos que transforman movimiento en un patrón deseable. Ejemplos comunes de mecanismos son los mecanismos de cuatro barras juntas de pasador y el mecanismo manivela corredera para los cuales existen fórmulas analíticas que permiten el cálculo de posición, velocidad, acerleración y fuerzas de sus eslabones.

La rotación de cuerpos en dos dimensiones está definida por la matriz de rotación cuya inversa es su transpuesta al ser una transformación ortogonal.

Los lenguajes de programación son notaciones que describen procesos computacionales facilitando la formulación de algoritmos. Existen diferentes tipos y clasificaciones de lenguajes de programación de acuerdo a su tipo de implementación y al modelo de computación que emplean.

Python es un lenguaje de programación interpretado, gratuito, de código abierto, multiparadigma y multipropósito. Este cuenta con la librería Matplotlib® la cual permite renderizar imágenes y animaciones a través de su interfaz de programación de aplicaciones (API) que consta de tres capas: *Renderer*, *Artist* y *FigureCanvas*.

CAPÍTULO 2

Antecedentes

La solución de mecanismos de n barras fue planteada por primera vez por Charles Wampler en el artículo de investigación Solving the Kinematics of Planar Mechanisms. En dicho artículo presenta un método general para el análisis de mecanismos planos conformados por eslabones rígidos con juntas de pin y de pasador. Los eslabones se deben definir como vectores en el plano complejo, luego se debe de formular una serie de ecuaciones polinomiales para determinar las ubicaciones de los eslabones al realizar el ensamblaje. Finalmente, el sistema de ecuaciones se reduce a un problema de autovalores generalizado o en ciertos casos a un único polinomio resultante (Wampler, 1999).

Trasladar las geometrías de los eslabones al plano comlejo permite que la posición de un punto al final de una serie de eslabones sea determinada por una suma. Luego se deben plantear los lazos vectoriales del mecanismo cuyas soluciones describen su movimiento. Posteriormente se selecciona una junta como la salida eliminando el resto de juntas de las ecuaciones de los lazos para obtener la ecuación de entrada/salida (Wampler, 1999).

Posteriormente, se obtienen las coordenadas isotrópicas basadas en las variables exponenciales y sus conjugados. De ahí se realiza una reducción a cuadráticas bilineales. Estas ecuaciones son posteriormente resueltas empleando una fórmula de determinante (Wampler, 1999).

En el mismo estudio se plantea un ejemplo práctico para el mecanismo de Stephenson. El autor termina concluyendo que formular las ecuaciones cinemáticas de mecanismos en el plano complejo a través de coordenadas isotrópicas producen ecuaciones sencillas de resolver empleando una fórmula basada en la identidad del determinante de Sylvester. Empleando dicha metodología el autor fue capaz de resolver problemas de hasta doce eslabones (Wampler, 1999).

Otra metodología para resolver mecanismos de n barras fue presentada por James Nielsen y Bernard Roth en el artículo Solving the Input/Output Problem for Planar Mechanisms. Dicho método también plantea la solución de mecanismos con juntas de pin y pasador a través de una modificación al método de la resultante de Dixon que es empleado para resolver

ecuaciones polinomiales (Nielsen & Roth, 1999). En este método se presenta únicamente aritmética real para construir una matriz cuyo determinante resuelve las posiciones del mecanismo. A lo largo del artículo se emplea el método en un mecanismo de doble mariposa. Primero se plantean las ecuaciones de lazos vectoriales, posteriomente se establece el eslabón de entrada y finalmente se procede con la reolución a través del determinante de Dixon (Nielsen & Roth, 1999).

Una vez hallada la respuesta esta cae dentro de uno de los cuatro casos propuestos. Después de seguir el procedimiento establecido por cada caso se aplican identidades trigonométricas (Nielsen & Roth, 1999).

Los autores concluyen con una serie de pasos que constitutyen el procedimiento: determinar las ecuaciones lineales de lazos vectoriales, suprimir uno de los ángulos desconocidos, calcular el determinante de Dixon, encontrar la matriz resultante y finalmente resolver hacia atrás. Resaltan la independencia del método de la existencia de lazos de cuatro barras en el mecanismo así como de no necesitar de una inversión cinemática (Nielsen & Roth, 1999).

El análisis de deflexión y esfuerzos de mecanismos ha sido parte de numerosos estudios como Deflection and Stress Analysis in High Speed Planar Mechanisms With Elastic Links. En este artículo se presenta un método de análisis de deflexión de mecanismos aplicable a todo tipo de mecanismo plano y de multi lazo (Imam et al., 1973).

El procedimiento se basa en la extensión del método del vector de permutaciones del análisis estructural al análisis cineto-elastodinámico de mecanismos, y mediante la introducción de la tasa de cambio de los autovalores con respecto al movimiento de los mecanismos. Para ello se requiere una descripción del mecanismo, una conversión de coordenadas, una construcción de matrices de rigidez y una solución de autovalores (Imam et al., 1973).

Se concluyó que el método diseñado es efectivo para hallar los esfuerzos dinámicos en mecanismos (Imam et al., 1973).

Otro estudio realizado sobre los esfuerzos en mecanismos es Stress Fluctuation in High Speed Mechanisms, donde se derivan expresiones analíticas que describen las fluctauaciones de esfuerzos en los eslabones de un mecanismo que opera a altas velocidades. Donde se usa como referencia un mecanismo de cuatro barras manivela-corredera (Yang et al., 1981).

La compañía de software de diseño en 2D y 3D Autodesk®desarrolló las aplicaciones ForceEffect y ForceEffect Motion. Estas aplicaciones permitían el análisis de cargas en cuerpos estáticos y el análisis cinemático de mecanismos.

Otras programas que permiten la animación de mecanismos son GeoGebra® y Working Model®. GeoGebra® requiere una definición del comportamiento de los mecanismos a través de relaciones geométricas y Working Model solo permite una simulación de los movimientos sin proveer información acerca de los esfuerzos en los mecanismos. Por último, Solidworks® permite tanto la definición de mecanismos como el análisis de esfuerzos a través de la metodología de elementos finitos.

CAPÍTULO 3

Justificación

Actualmente no existe disponibilidad de herramientas de software gratuitas, de bajos recursos computacionales y de código abierto que permitan el análisis de movimiento y esfuerzos de eslabones de geometrías complejas en mecanismos. Un antecedente del programa que se planea diseñar llamado Force Effect Motion fue descontinuado por Autodesk®. Este software permitía únicamente el análisis cinemático de mecanismos representados por segmentos de línea. Existen otros softwares similares como Linkages que no permiten el análisis de esfuerzos («Linkage mechanism designer and simulator – dave's blog», s.f.). Por otro lado, Solidworks® presenta la capacidad de realizar análisis cinéticos; sin embargo, es una herramienta cuyo costo se encuentra en los miles de dólares y no se tiene acceso al código fuente. Esto implica que realizar un análisis de esfuerzos para eslabones complejos requiere de cálculos a mano empleando métodos numéricos como puede ser por elementos finitos (Jonker, 1991) o pagar licensias de *software* propietario. Para realizar la simulación cinemática de mecanismos es posible emplear GeoGebra®; sin embargo, hay que establecer una serie de relaciones entre curvas para que esto llegue a funcionar.

Los mecanismos se encuentran en todo tipo de aplicaciones presentes en la industria y en la vida diaria. En el caso del mecanismo de cuatro barras este se encuentra presente en: bicicletas, suspensiones de vehículos, pinzas y válvulas para pozos petroleros («A novel classification of planar four-bar linkages and its application to the mechanical analysis of animal systems», 1996). Otros mecanismos como el de cinco barras son empleados ampliamente en robótica para diseñar impresoras 3d, cortadoras láser y otras máquinas. Mientras que, hay propuestas para emplear mecanismos de seis barras para diseñar prensas mecánicas (Hu et al., 2016).

El análisis cinemático de mecanismos es de suma importancia para diseñar y comprender el diseño de máquinas. Al momento de diseñar una máquina se busca que esta sea capaz de realizar un trabajo y que cumpla requisitos de desplazamiento y dimensionamiento (Barton, 2015). Antes de iniciar con el proceso de diseño de una máquina la movilidad del mecanismo debería ser determinada, luego el plan de propulsión y control pueden ser descritos. Por ende, el análisis correcto del movimiento de mecanismos es un prerrequisito para el diseño

y análisis estructural de máquinas (Zhao et al., 2014). Es importante realizar un análisis de esfuerzos en los eslabones de un mecanismo, puesto que el estudio de este fenómeno permite determinar las secciones sensibles del diseño y realizar las optimizaciones requeridas para aumentar la confiabilidad y reducir costos y peso. Realizar este examen permite establecer un factor de seguridad y los materiales adecuados a emplear («Llis», s.f.).

Los programas de computadora permiten simplificar tareas tediosas y repetitivas. De esta manera aumentan la productividad de sus usuarios. Programas como Autodesk Inventor® y Solidworks® se emplean para diseñar piezas mecánicas, ANSYS® para simulaciones, AutoCAD® para la elaboración de planos, Excel® para realizar hojas de cálculo y la lista podría continuar con un sinfín de herramientas que aceleran la generación de soluciones de problemas de ingeniería.

Se busca crear e implementar modelos de mecánica de sólidos para simular el comportamiento de mecanismos compuestos por eslabones complejos mediante el uso de métodos numéricos. Se tendrá como resultado un software que permita realizar un análisis cinético en 2D dando libertad al usuario para la definición de geometrías de los eslabones y sus conexiones. De esta manera se tendrá una herramienta que actualmente no se encuentra disponible en el mercado de forma gratuita, de código abierto y que requiera de pocos recursos de cómputo. Esta herramienta facilitará el diseño de máquinas y cumplirá fines didácticos para cursos de Mecanismos, Resistencia de Materiales y Diseño Mecánico. El software se desarrollará haciendo uso de información disponible sobre la cinemática y cinética de mecanismos de cuatro barras.

CAPÍTULO 4

Objetivos

4.1. Objetivo general

Crear e implementar modelos de mecánica de sólidos para simular y analizar el comportamiento de mecanismos de n barras de un grado de libertad, formados por lazos vectoriales de cuatro eslabones, de forma algorítmica.

4.2. Objetivos específicos

- Crear una interfaz gráfica que permita visualizar las posiciones de eslabones de formas complejas en mecanismos.
- Implementar un algoritmo para determinar las aceleraciones y fuerzas que sufren los eslabones del mecanismo.
- Determinar los esfuerzos equivalentes de von Mises mediante un algoritmo y presentárselos al usuario.

CAPÍTULO 5

Marco teórico

5.1. Sintesis de posición de mecanismos

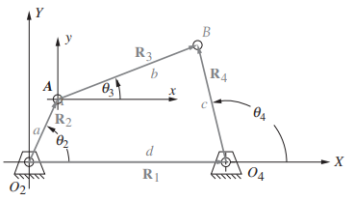
Un mecanismo es un dispositivo que transforma el movimiento en un patrón deseable, y por lo general desarrolla fuerzas muy bajas y transmite poca potencia.

El eslabonamiento más simple con un grado de libertad es el mecanismo de cuatro barras, el cual es un dispositivo extremadamente versátil y útil.

5.1.1. Solución de mecanismos de cuatro barras

A continuación se presenta la solución de un mecanismo de cuatro barras con juntas de pasador. Este mecanismo se encuentra presente en: bicicletas, suspensiones de vehículos, pinzas y válvulas para pozos petroleros.

Figura 1: Mecanismo de cuatro barras juntas de pasador



Fuente: (Norton, 2013)

$$A = cos(\theta_2) - \frac{d}{a} - \frac{d}{c}cos(\theta_2) + \frac{a^2 - b^2 + c^2 + d^2}{2ac}$$

$$B = -2sin(\theta_2)$$

$$C = \frac{d}{a} - (\frac{d}{c} + 1)(cos(\theta_2)) + \frac{a^2 - b^2 + c^2 + d^2}{2ac}$$

$$\theta_4 = 2arctan(\frac{-B \pm \sqrt{B^2 - 4AC}}{2A})$$

Donde:

a = longitud de manivela

b = longitud de acoplador

c = longitud de esabón de salida

d = longitud de bancada

θ_2 = angulo de manivela

θ_4 = angulo de eslabon de salida

La solución de un mecanismo de cuatro barras manivela-corredera está dada por las siguientes ecuaciones:

$$\theta_{3_1} = arcsin(\frac{asin(\theta_2) - c}{b})$$

$$\theta_{3_2} = arcsin(-\frac{asin(\theta_2) - c}{b}) + \pi$$

$$d = acos(\theta_2) - bcos(\theta_3)$$

Donde:

a = longitud de manivela

b = longitud de acoplador

c = descentrado

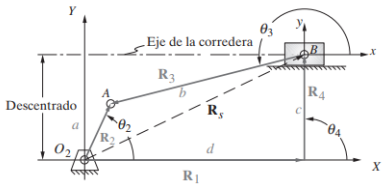
d = posición de la corredera

θ_2 = angulo de manivela

θ_3 = angulo del acoplador

(Norton, 2013)

Figura 2: Mecanismo de cuatro barras manivela-corredera



Fuente: (Norton, 2013)

5.1.2. Concatenación de mecanismos

Para la solución de mecanismos de más de cuatro barras un posible método de solución es analizar por separado los lazos vectoriales que conforman al mecanismo. Un lazo vectorial consiste en la represetanción por vectores de posición de los eslabones de un mecanismo de forma que su suma sea igual a cero. Se determinan las ecuaciones correspondientes y se resuelven en sucesión aplicando los resultados del primer lazo al segundo y así sucesivamente (Uicker et al., 2017).

Es decir, dentro de un mecanismo se plantean uno o más lazos vectoriales formados por cuatro eslabones. La salida de este mecanismo representa el ángulo de entrada al siguiente y así sucesivamente hasta encontrar las posiciones de todos los eslabones dentro del mecanismo (Uicker et al., 2017).

5.2. Cálculo de velocidades

Las velocidades angulares de los eslabones de un mecanismo de cuatro barras juntas de pasador están dadas por:

$$\omega_3 = \frac{a\omega_2}{b}\frac{sin(\theta_4-\theta_2)}{sin(\theta_3-\theta_4)}$$

$$\omega_4 = \frac{a\omega_2}{c}\frac{sin(\theta_2-\theta_3)}{sin(\theta_4-\theta_3)}$$

Donde:

ω_3 = velocidad angular del acolplador
ω_4 = velocidad angular de la salida

(Sclater & Chironis, 2007)

5.3. Cálculo de aceleraciones

Los eslabones de un mecanismo se encuentran sometidos a una aceleración tangencial y a una aceleración normal. Asumiendo una aceleración angular como entrada en la manivela las a aceleraciones angulares del acoplador y de la salida están dadas por:

$$\alpha_3 = \frac{CD-AF}{AE-BD}$$

$$\alpha_4 = \frac{CE-BF}{AE-BD}$$

$$A = csin(\theta_4)$$

$$B = bsin(\theta_3)$$

$$C = a\alpha_2 sin(\theta_2) + a\omega_2^2 cos(\theta_2) + b\omega_3^2 cos(\theta_3) - c\omega_4^2 cos(\theta_4)$$

$$D = dcos(\theta_4)$$

$$E = bcos(\theta_3)$$

$$F = a\alpha_2 cos(\theta_2) - a\omega_2^2 sin(\theta_2) - b\omega_3^2 sin(\theta_3) + c\omega_4^2 sin(\theta_4)$$

Donde:

α_3 = aceleración angular del acoplador
α_4 = aceleración angular de la salida

(Norton, 2013)

5.4. Cálculo de fuerzas

Las fuerzas de un mecanismo de cuatro barras están dadas el siguiente sistema de ecuaciones:

$$\begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ -R_{12_y} & R_{12_x} & -R_{32_y} & R_{32_x} & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & -1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & R_{23_y} & R_{23_x} & -R_{43_y} & R_{43_x} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & R_{34_y} & -R_{34_x} & -R_{14_y} & R_{14_x} & 0 \end{bmatrix} \times \begin{bmatrix} F_{12_x} \\ F_{12_y} \\ F_{32_x} \\ F_{32_y} \\ F_{43_x} \\ F_{43_y} \\ F_{14_x} \\ F_{14_y} \\ T_{12} \end{bmatrix} = \begin{bmatrix} m_2 a_{G_2 x} \\ m_2 a_{G_2 y} \\ I_{G_2}\alpha_2 \\ m_3 a_{G_3 x} \\ m_3 a_{G_3 y} \\ I_{G_3}\alpha_3 - T_3 \\ m_4 a_{G_4 x} \\ m_4 a_{G_4 y} \\ I_{G_4}\alpha_4 - T_4 \end{bmatrix}$$

Donde:

I = momento de inercia polar respecto al centroide de cada eslabón

a_G = aceleración respecto al sistema de coordenadas global

T_3 = momento externo aplicado al acoplador

T_4 = momento externo aplicado a la salida

(Norton, 2013)

5.5. Rotación de cuerpos en el espacio

Para rotar un vector alrededor del origen se tiene la siguiente matriz:

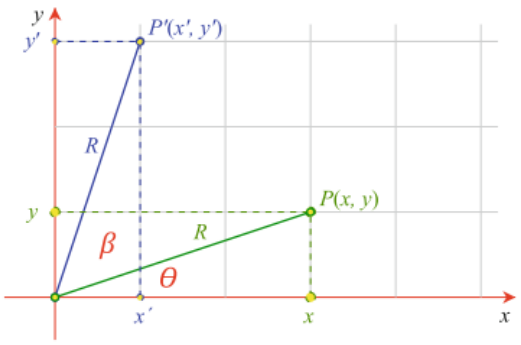
$$\begin{bmatrix} cos(\beta) & -sin(\beta) & 0 \\ sin(\beta) & cos(\beta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix}$$

Para realizar rotaciones alrededor de un punto arbitrario se tiene la siguiente matriz:

$$\begin{bmatrix} cos(\beta) & -sin(\beta)p_x(1-cos(\beta)) & p_y sin(\beta) \\ sin(\beta) & cos(\beta)p_y(1-cos(\beta)) & -p_x sin(\beta) \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix}$$

(Vince, 2017)

Figura 3: Rotación de cuerpos



Fuente: (Vince, 2017)

5.6. Esfuerzos

5.6.1. Teorías de falla

Esfuerzo cortante máximo

La fluencia comienza cuando el esfuerzo cortante máximo de cualquier elemento iguala al esfuerzo cortante máximo en una pieza de ensayo a tensión del mismo material cuando esa pieza comienza a fluir.

$$\tau = \frac{S_y}{2n}$$

Energía de distorsión

La falla por fluencia ocurre cuando la energía de deformación total por unidad de volumen alcanza o excede la energía de deformación por unidad de volumen correspondiente a la resistencia a la fluencia en tensión o en compresión del mismo material.

Teoría Esfuerzo normal máximo

La falla ocurre cuando uno de los tres esfuerzos principales es igual o excede la resistencia.

$$\sigma_1 > S_u t$$
$$\text{o}$$
$$\sigma_3 < -S_u t$$

5.6.2. Esfuerzo de von Mises

El valor del esfuerzo de von Mises está dado por:

$$\sigma' = \sqrt{\frac{(\sigma_x - \sigma_y)^2 + (\sigma_y - \sigma_z)^2 + (\sigma_z - \sigma_x)^2 + 6(\tau_{xy}^2 + \tau_{yz}^2 + \tau_{zx}^2)}{2}}$$

Mientras que para el esfuerzo plano su valor es:

$$\sigma' = \sqrt{\sigma_x^2 - \sigma_x \sigma_y + \sigma_y^2 + 3\tau_{xy}^2}$$

5.7. Cálculo de deflexiones de mecanismos

La síntesis y análisis cinético-elastodinámica es una herramienta para tomar en cuenta las deflexiones y distribución de masa en el comportamiento de los mecanismos. El enfoque

de elementos finitos analiza al mecanismo como una armadura en cada instante de análisis. Por lo tanto se tienen matrices de rigidez, de masa y de cuerpos rígidos que varían en cada posición. Se tiene un sistema de coordenadas global y local para cada miembro.

Se plantea la siguiente matriz con las ecuaciones de movimiento.

$$[M] \times [\delta] + [C] \times [\delta] + [K] \times [\delta] = [F]$$

Donde: M contiene la masa C es el amortiguamiento K es la rigidez F son las fuerzas inerciales y δ son las coordenadas generalizadas desconocidas

Para resolver las ecuaciones se requiere: determinar los autovalores y autovectores asumiendo que el amortiguamiento es 0 al igual que las fuerzas, transformar el sistema de ecuaciones y desarrollar la respuesta transitoria del sistema para una velocidad angular constante del eslabón de entrada del mecanismo. Para reducir el costo de cómputo de eigenvalores en cada instante del mecanismo se tienen ecuaciones que describen su tasa de cambio. (Imam et al., 1973)

5.8. Cálculo de esfuerzos en mecanismos

A altas velocidades de operación las cargas externas variables y las fuerzas de inercia internas producen esfuerzos variables en los eslabones de los mecanismos.

Para determinar los esfuerzos se emplean las deformaciones instantáneas obtenidas con el método de elementos finitos teniendo como resultado la ecuación de carga y el momento flector:

$$Intensidad_carga = m\omega_i^2 \Phi_i(x,t)$$

$$\frac{\partial^2 M_i(x,t)}{\partial x^2} = -m\omega_i^2 \Phi_i(x,t)$$

$$\Phi = -m\omega^2[B_1 x^3 + B_2 x^2 + B_3 x + B_4]$$

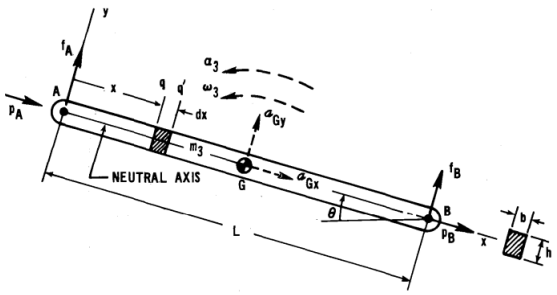
$$M(x,t) = -m\omega^2(\frac{B_1 x^5}{20} + \frac{B_2 x^4}{12} + \frac{B_3 x^3}{6} + \frac{B_4 x^2}{2})$$

A partir del momento flector se pueden calcular los esfuerzos por flexión y los esfuerzos axiales pueden determinarse por las deformaciones longitudinales de los elementos.(Imam et al., 1973)

5.8.1. Fluctuación de esfuerzos

Es necesario conocer las fluctuaciones de esfuerzos de los miembros para poder determinar la falla por fatiga. Se toma como ejemplo el acoplador de un mecanismo manivela corredera para demostrar el comportamiento de esfuerzos en los eslabones (Figura 3).

Figura 4: Diagrama de cuerpo libre del acoplador de un mecanismo manivela corredera



Fuente: (Yang et al., 1981)

Los componentes axiales y transversales del acoplador de un mecanismo manivela corredera están dados por:

$$p_a = -p_b + m_3 a_g x$$
$$f_a = \frac{m_3}{2} \times [a_g y - L/6\alpha_3]$$

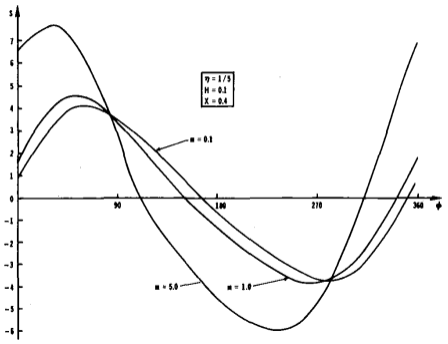
Los componentes axiales y transversales de la corredera del mismo mecanismo son:

$$p_b = -f_b tan(\theta) - \frac{m_4 a_4}{cos(\theta)}$$
$$f_b = \frac{m_3}{2} \times [a_g y + L/6\alpha_3]$$

Los esfuerzos alternantes en el acoplador se encuentran como en la siguiente ecuación. En la Figura 5 se muestran las fluctuaciones de los esfuerzos alternates para diferentes relaciones de masa entre acoplador y corredera.

$$\sigma_1 = [\frac{m_3 R \omega_2^2}{bh}] \times A_o + \sum_{n=1}^{3} A_n cos(n\phi) + B_n sin(n\phi)$$

Figura 5: Relaciones masa de acoplador y corredera y los efectos en los esfuerzos



Fuente: (Yang et al., 1981)

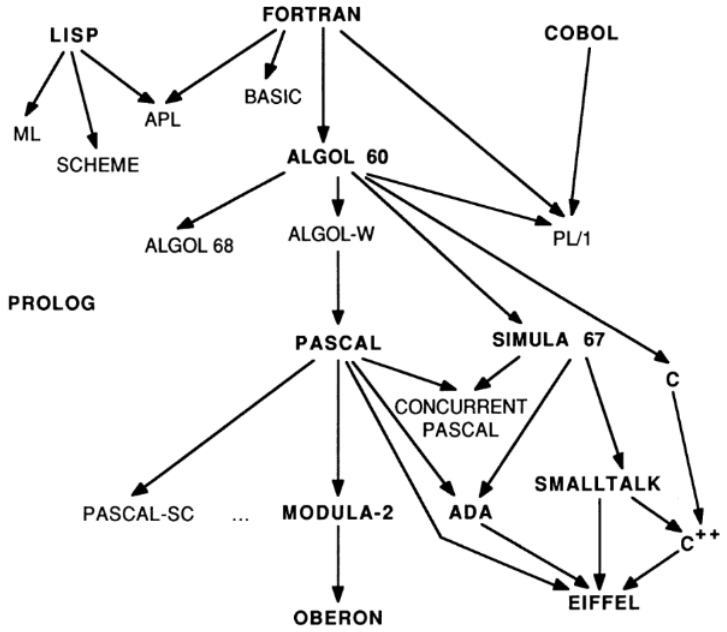
5.9. Lenguajes de programación

Un lenguaje de programación consiste en una notación sistemática a través de la cual se describen procesos computacionales. Los lenguajes de programación son herramientas que facilitan la formulación de algoritmos (Teufel, 1991).

5.9.1. Historia

El lenguaje de programación Torre de Babel representa el auge de lenguajes y dialectos en las etapas tempranas del desarrollo de lenguajes de programación. Existe una gran cantidad de lenguajes de programación; sin embargo, solo unos pocos son empleados por la mayoría de programadores. En la Figura 6 se presentan las relaciones entre lenguajes de programación más importantes (Teufel, 1991).

Figura 6: Relaciones entre lenguajes de programación



Fuente: (Teufel, 1991)

La historia de lenguajes de programación se compone de generaciones.

Primera generación:

Las computadoras operan a nivel binario. Los lenguajes de está generación consisten en código binario de máquina; es decir, unos y ceros. Son lenguajes que reflejan el *hardware* del sistema.

Segunda generación:

En esta generación se introdujeron los lenguajes de ensamblador que permitieron el uso de abreviaciones en forma de nombres simbólicos. También se introdujeron los conceptos de comandos y operadores.

Se desarrollaron sistemas de compilación llamados ensambladores para traducir los programas simbólicos a código de máquina. Aún se ve reflejado en ellos el *hardware* de la máquina a través de registros. Esto los hace dependientes de la máquina y difíciles de leer por personas que no sean los autores.

Tercera generación:

En la tercera generación de lenguajes de programación se encuentran los lenguajes de alto nivel. Estos lenguajes permiten el control de estructuras basadas en variables de tipos específicos. Proveen un nivel de abstracción que permite especificar información, procesos y control independiente de la computadora (Teufel, 1991).

5.9.2. Tipos de lenguajes

La primera clasificación de lenguajes de programación está dada por su tipo de implementación:

- Intérprete puro que ejecuta el código fuente
- Compilador nativo
- Transpilador que traduce un lenguaje de programación a otro
- Compilador de código intermedio que está acompañado de una máquina que lo interpreta

(Jeffery, 2021)

Los lenguajes de programación también pueden estar clasificados en base al modelo de computación que empleen. La clasificación más general basada en modelo de computación es declarativo o imperativo. Los lenguajes declarativos se enfocan en lo que la computadora ha de hacer; mientras que los lenguajes imperativos se enfocan en como lo debe hacer la computadora.

Por lo tanto, se puede interpretar como si los lenguajes declarativos estuvieran más relacionados con el punto de vista del programador. Sin embargo, los lenguajes imperativos son los lenguajes dominantes debido a contar con un desempeño superior.

Los lenguajes declarativos e imperativos se subdividen en subgrupos:

- Declarativos:
  - Funcionales: emplean un modelo basado en la definición recursiva de funciones. Su funcionamiento está inspirado en lambda calculus, que es un modelo computacional desarrollado por Alonzo Church en 1930. Un programa consiste en una función de entradas y salidas definido en términos de funciones más sencillas.
  - *Dataflow*: modelan la cumputación como un flujo de información entre nodos primitivos. Proveen un modelo inherentemente paralelo donde los nodos operan por la recepción de *tokens* y pueden operar de manera concurrente.
  - Lógicos: buscan valores que satisfacen ciertas relaciones, realizando búsquedas a través de reglas lógicas.
- Imperativos:
  - von Neumann: se basan en la modificación de variables. Constan de declaraciones y asignaciones que afectan las computaciones que le siguen cambiando el valor de la memoria.
  - Orientados a Objetos: están fuertemente relacionados a los lenguajes von Neumann, pero cuentan con un modelo más estructurado y distribuido tanto de computación como de memoria. Se modelan las operaciones computacionales como interacciones semiindependiente entre objetos. Cada uno de estos objetos tiene un estado y subrutinas que manejan dicho estado.
  - *Scripting*: son lenguajes caracterizados por coordinar componentes de los alrededores. Están enfatizados en un prototipado veloz con un sesgo en la facilidad de expresión por encima de la velocidad de ejecución.

(Scott, 2016)

5.9.3. Lenguajes populares

C++:

C++ es un lenguaje programación de propósito general cuya principal aplicación es el desarrollo de sistemas. Este lenguaje es utilizado en otras áreas siendo empleado en la programación de microcomputadoras hasta supercomputadoras. El lenguaje es un *superset* de C que provee facilidad y flexibilidad para la definición de nuevos tipos de variables a través de una técnica llamada abstracción de data.

El concepto clave en C++ es la clase. Este es un tipo definido por el usuario. Las clases proveen encapsulamiento de la información, inicialización de la información, conversión implícita de tipos, tipos dinámicos, control de memoria y sobrecarga de operadores.

El lenguaje proporciona mejores herramientas para determinar el tipo de variables y para expresar modularidad que C. Incluye constantes simbólicas, substitución de funciones

en línea, argumentos predeterminados, sobrecarga de nombres de funciones y referencias (Stroustrup, 2013).

Java:

Java es un lenguaje de programación orientado a objetos desarrollado por Sun Microsystems® para el desarrollo de aplicaciones de negocios y para aplicaciones de Internet.

Es un lenguaje de arquitectura neutral. Es decir, se puede emplear para desarrollar aplicaciones que puedan ser ejecutadas en cualquier dispositivo y sistema operativo. Esto es porque no se ejecuta directamente en una computadora sino que se ejecuta en la Java Virtual Machine.

Java permite el desarrollo de dos tipos principales de aplicaciones: Applets, que son aplicaciones en una página Web; y aplicaciones de Java, que son programas independientes que pueden ser con interfaces de caracteres en una consola o aplicaciones gráficas (Farrell, 2016).

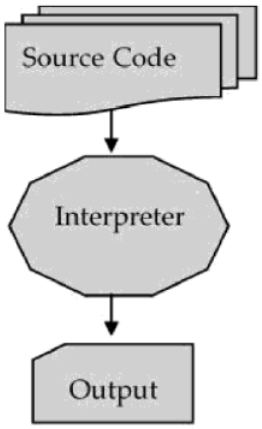
Python:

Python es un lenguaje de programación interpretado diseñado por Guido van Rossum en el año 1991. Al ser un lenguaje interpretado sigue un flujo de ejecución como el mostrado en la Figura 7.

Es un lenguaje gratuito y de código abierto. Además, es un lenguaje independiente de sistema. Por lo tanto puede ser ejecutado en diferentes sistemas operativos.

Python es un lenguaje multiparadigma y multipropósito. Puede ser empleado tanto para el desarrollo de *scripts* como para el desarrollo de aplicaciones. Permite un estilo de programación orientado a objetos, imperativo y funcional. Cuenta con un manejo automático de memoria (Mohbey & Bakariya, 2022).

Figura 7: Flujo de ejecución lenguaje interpretado



Fuente: (Mohbey & Bakariya, 2022).

C#:

C# es un lenguaje de programación general y multiparadigmaque combina programación dinámica, orientada a objetos, genérica, funcional y declarativa. Es la elección principal para el desarrollo de aplicaciones .NET para *desktop*, web y móvil.

El lenguaje fue diseñado para CLI (*Common Language Infrastructure*) que es una especificación desarrollada por Microsoft® y estandarizada por la International Standarization Organization (ISO) y Ecma International - European association for standardizing information and communication systems® que describe código ejecutable y el ambiente a emplear para distintas plataformas de computadora.

Su desarrollo comenzó en los años 90. Esta inspirado en Java, C++, Delphi y Smalltalk. La primera versión se hizo pública en 2002 y ha ido evolucionando desde entonces (Bancila et al., 2020).

Javascritpt:

Javascript es el lenguaje de programación del lado del cliente más usado. Permite agregar interacción, animaciones y efectos visuales a HTML. Es un lenguaje orientado a objetos, multiplataforma cuya intención es ser ejecutado en un ambiente como un navegador web.

Fue desarrollado en 1995 en Netscape®. Posteriormente, fue estandarizado por la Ecma International - European association for standardizing information and communication systems (Thanh, 2021).

Rust:

Rust es un lenguaje de programación de sistema que provee seguridad de memoria por defecto sin un recolector de basura, esto afecta su tiempo de ejecución. Es un lenguaje versátil que puede ser utilizado para: desarrollo web, motor de videojuegos o clientes web (Matzinger, 2019).

Es un lenguaje de código abierto desarrollado por Mozilla®. La primera versión estable fue lanzada en 2015. Esta inspirado en Cyclone, C++ y Haskell (Sharma et al., 2019).

Go:

Es un lenguaje de programación de código abierto cuyo desarrollo inició en 2007. Es similar a C; sin embargo, ha tomado prestadas y adaptado ideas de otros lenguajes. Cuenta con recolector de basura.

Es un lenguaje de programación general empleado para gráficos, aplicaciones móviles y aprendizaje automático. Esta especialmente desarrollado para el desarrollo de servidores y herramientas y sistemas para programadores (Donovan & Kernighan, 2015).

5.10. Gráficos por computadora

Figura 8: Representación de imagen como matriz con escala de color

Fuente: (Tanimoto, 2012)

Los gráficos por computadora involucran imágenes u objetos gráficos que son una colección de elementos discretos llamados pixeles. Un píxel es el elemento más pequeño que puede ser localizado en una pantalla, es la representación de una región que contiene una cantidad infinita de puntos. Un píxel puede ser controlado a través de su color e intensidad .

El proceso para determinar los píxeles que representan una imagen se denomina rasterizado. El proceso para representar una imagen continua como una colección de puntos discretos recibe el nombre de conversión de escan.

5.10.1. Librerías de gráficos por computadora

Un motor de juegos es una aplicación que permite desarrollar videojuegos desde la base. Las características entre motrores de juegos varían pero generalmente permiten importar modelos 3D, imágenes, audio y video para posteriormente manipularlos empleando un lenguaje de programación (Nicoll & Keogh, 2019).

OpenGL:

Open Graphics Library es una API (Application Programming Interface) que le proporciona a desarrolladores funciones para manipular imágenes y gráficos. Es la base para la mayoría de los juegos modernos, ya sean móviles, iOS o Android; consolas o cumputadoras de escritorio (Hussain, 2018).

Matplotlib:

Librería con el lenguaje de programación Python para realizar gráficos por computadora. La librería cuenta con dos interfaces: implícita y explícita. La interfaz explícita está basada en el modelo de programación orientada a objetos. La interfaz implícita realiza suposiciones sobre el objeto al cual el usuario desea cambiar.

La interfaz explícita es como está implementada la librería, personalizaciones y ajustes de gráficas se realizan a este nivel. Su funcionamiento se basa en instanciar una Figura a la cual se le aplica un método para generar uno o más ejes. Finalmente se llaman a métodos para dibujar sobre los ejes. Es una interfaz explícita puesto que se hace una referencia directa a cada uno de los objetos y cada uno de ellos es empleado para generar el siguiente objeto. Esto

permite una personalización de los objetos después de ser creados y antes de ser mostrados.

La interfaz implícita oculta los métodos de los ejes. Donde la creación de las Figuras y de los ejes se hace de forma automática. Útil para trabajo interactivo o para scripts sencillos. Una referencia a la Figura actual puede ser extraída llamando al método gcf y al eje actual empleando el método gca.

La API de matplotlib cuenta con tres capas: *FigureCanvas*, *Renderer* y *Artist*. *FigureCanvas* representa la región donde se dibuja la figura, Renderer es el objeto que sabe como dibujar sobre el *FigureCanvas* y el *Artist* es el objeto que sabe usar el *Renderer* para pintar sobre el lienzo. Esta última capa es la que será empleada por el usuario para administrar el texto, líneas y figuras a manipular (Hunter, 2007).

Unreal®:

Es un motor de juegos diseñado para C++. Emplea tanto C++ como Blueprint, este último es un lenguaje para desarrollar *scripts* de forma visual. Algunos de los videojuegos populares desarrollados con Unreal son: Fortnite®, Gears V® y Sea of Thieves®.

El motor de videojuegos fue desarrollado por Epic Games® y para poder interactuar con el se debe instalar un editor. Para trabajar con el editor se emplean plantillas y se establecen las configuraciones para las plataformas, la calidad de los gráficos y el rendimiento deseado (Fozi et al., 2020).

Unity®:

Es un motor de videojuegos propiedad de Unity Technologies®. Es de fácil uso y acceso, capaz de escalarse a la necesidad para desarrollar aplicaciones profesionales, amateur o industriales. La mitad de todos los videojuegos y proyectos de realidad virtual son desarrollados en Unity®.

Unity promueve un sistema de diseño orientado a componentes, descentraliza el papel del programador. Para el desarrollo de videojuegos se emplea el lenguaje C# (Nicoll & Keogh, 2019).

CAPÍTULO 6

Metodología

Para desarrollar el programa de análisis de mecanismos de n barras primero se determinó el tipo de mecanismos que iban a ser analizados. Para esto se decidió analizar los mecanismos de cuatro barras al estar presentes en múltiples diseños de máquinas.

Para cumplir con el análisis de mecanismos de n barras se decidió concatenar los mecanismos de cuatro barras con juntas de pasador y el mecamismo de cuatro barras con una salida de corredera. Para el análisis de esfuerzos se encontraron las aceleraciones y fuerzas dinámicas de los eslabones, seguido de un análisis por secciones identificando el punto del máximo esfuerzo de von Mises.

Como potenciales usuarios se plantearon estudiantes de ingeniería que preferiblemente hayan llevado el curso de Mecanismos y cursos de Cálculo y programación básicos. Adicionalmente se desarrolló una herramienta para poder integrar *Sketches* 2D de Autodesk Inventor® con la aplicación para facilitar su uso al público objetivo en base a la retroalimentación de potenciales usuarios.

6.1. Definición de requisitos

Cuadro 1: Identificación de códigos

ID	Significado
F	Funcionalidad
L	Limitantes
G	Interfaz gráfica

Cuadro 2: Requisitos

Código	Descripción
F01	El programa debe funcionar en Windows 10
F02	Permitir el análisis de eslabones en 2D de espesor constante
F03	El programa debe funcionar en una computadora con una RAM mínima de 8GB
F04	Mostrar resultados con 15 % de error o menos
F05	Integrar el programa con herramientas familiares para los usuarios potenciales
L01	Restringir el perímetro de los eslabones por curvas paramétricas simples (Que no se cruzan)
G01	Crear un módulo que permita construir y editar eslabones a través de texto y una interfaz gráfica
G02	Permitir al usuario construir y editar mecanismos a través de una interfaz gráfica
G03	Permitir al usuario usar botones para realizar los cálculos de esfuerzos de von Mises
G04	Mostrar al usuario resaltados los valores máximos de esfuerzos
G05	Mostrar al usuario las aceleraciones de los eslabones
G06	Mostrar al usuario las fuerzas a las que están sometidos los eslabones
G07	Mostrar al usuario una simulación cinemática para diferentes ángulos de entrada

Listado de recursos:

- Computadora con Windows 10, preferiblemente con tarjeta gráfica
- Acceso a Internet
- Python 3.10
- Editor de texto
- 2 meses para el desarrollo del software
- Autodesk Inventor® 2023

6.2. Diseño

6.2.1. Geometría

Para el proceso del diseño del *software* primero se creó un diagrama de Lenguaje Unificado de Modelado (UML) de la lógica relacionada a la rotación, desplazamiento, aceleraciones, fuerzas y esfuerzos de los componentes de las máquinas (Figura 8).

Como unidad base para las geometrías se seleccionó un vector en dos dimensiones, este tiene la capacidad de: rotar dado un ángulo en radianes o en grados, trasladarse, adicionarse y substraerse a otro vector. Puede ser copiado y se puede obtener su longitud. Además, es posible obtener la distancia y el ángulo entre dos vectores.

El siguiente componente base detrás de la lógica del programa son las funciones. Estas están definidas por un punto de inicio y punto final siendo todos los números entre ellos parte del dominio de la función. Las funciones puede sumarse y restarse entre sí, entiéndase esta operación como computar el valor de cada una de las funciones y sumarlos o restarlos según sea el caso. A las funciones es posible computarles el área y la ubicación de su centroide. Una concatenación de funciones está definida como la creación de una nueva función a partir de

dos o más funciones, el inicio de esta es el valor mínimo de los inicios y el final es el valor máximo del grupo de funciones que se desean concatenar, mientras que el espacio entre funciones no definido se sustituye por una interpolación lineal.

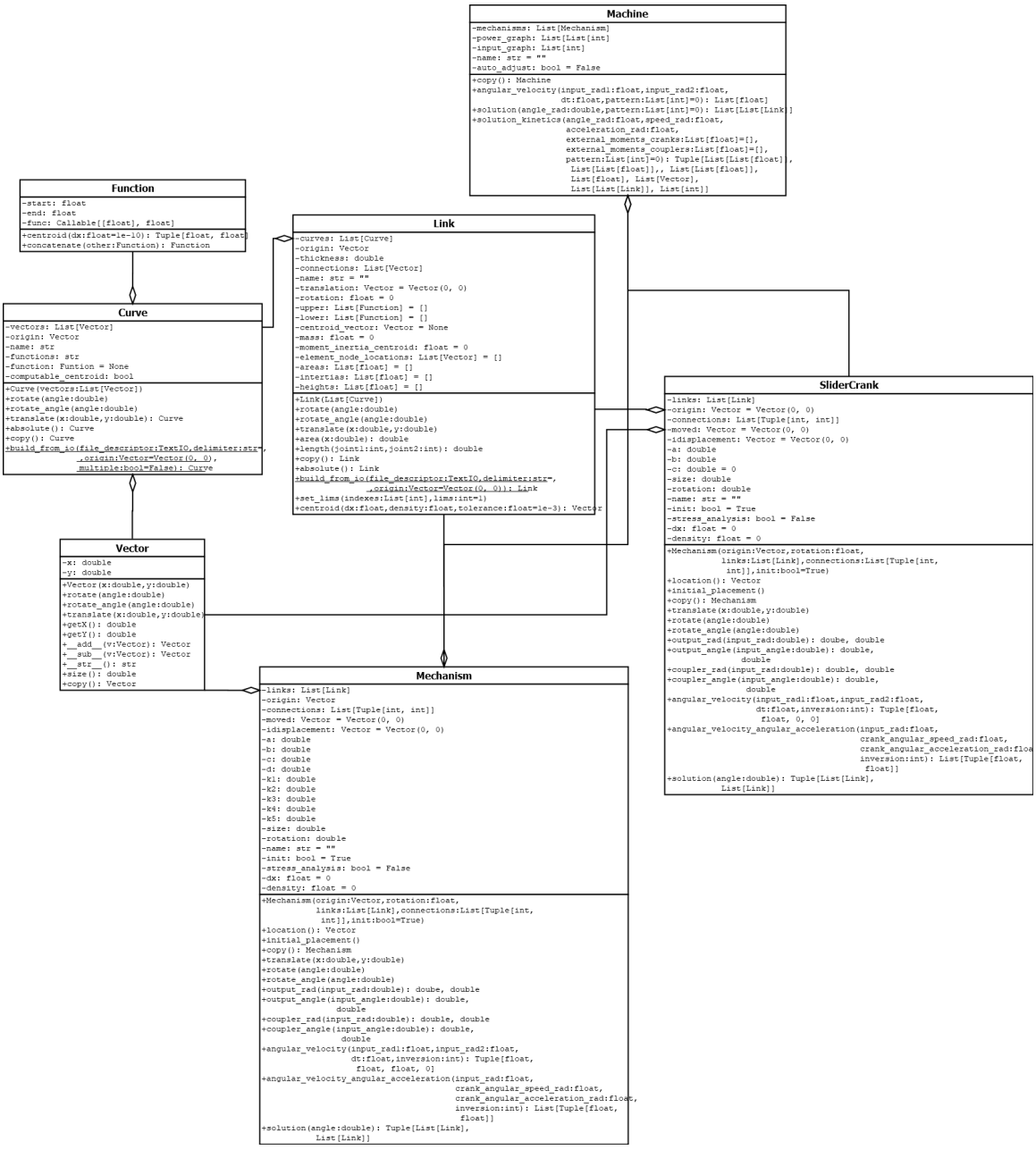
Una serie de vectores componen una curva, la cual también tiene un origen y un nombre para su identificación. De manera opcional es posible construir una curva con una o más funciones lo que permite realizar posteriormente el análisis de esfuerzos de los mecanismos. Adicionalmente, una curva se puede rotar y trasladar sin embargo esto deshabilita la opción de computar centroides.

Un eslabón está compuesto por una serie de curvas, un origen y una serie de puntos de conexión representados como un arreglo de vectores que es un sobrenombre para las juntas de un mecanismo. Adicionalmente, tiene un espesor constante y dos arreglos de funciones que definen su geometría para realizar un análisis de esfuerzos. Al definir las funciones que describen la geometría del mecanismo, su densidad y un diferencial es posible computar: su masa, centroide, momento de inercia polar, las áreas, inercias y alturas respecto a su eje neutro.

Una serie de eslabones forman un mecanismo que puede ser de cuatro barras con juntas de pasador o un mecanismo mananivela-corredera. Cada uno con un origen y nombre para su identificación además de poder rotarse, trasladarse y ser copiados. Los mecanismos poseen una serie de métodos para obtener el ángulo de salida en caso este sea un mecanismo juntas de pasador. Para un mecanismo manivela-corredera es posible computar la distancia de la corredera. En ambos casos se tiene como varible independiente el ángulo de rotación de la manivela en radianes. De la misma manera se puede obtener el ángulo de rotación del acoplador. La velocidad angular de cada uno de los eslabones es computable a través de dos métodos: empleando variables independientes dos ángulos de rotación de la manivela, un diferencial de tiempo y la configuración del mecanismo deseada (cruzada o abierta); el segundo método computa además aceleraciones angulares y empleando como variables independientes un único ángulo de rotación de la manivela, la velocidad y aceleración angular de la manivela y la configuración para la cual se desean las aceleraciones. Finalmente, el metodo de solución para los mecanismos devuelve las posiciones de cada uno de los eslabones para las dos configuraciones empleando como entrada el ángulo de rotación para la manivela.

Una máquina se definió como una serie de mecanismos con una relación de potencia entre ellos y un nombre para su identificación. La solución de una máquina es la posición de todos los eslabones de todos los mecanismos que la conforman para un ángulo de rotación para la manivela principal y para las configuraciones seleccionadas para cada uno de los mecanismos. Adicionalmente, es posible obtener las aceleraciones, fuerzas dinámicas y esfuerzos de cada uno de los eslabones del sistema al ingresar un ángulo de entrada, una velocidad angular inicial y una aceleración angular a la manivela principal así como una serie de momentos externos aplicados a los acopladores y a las salidas de los mecanismos.

Figura 8: UML lógica del programa

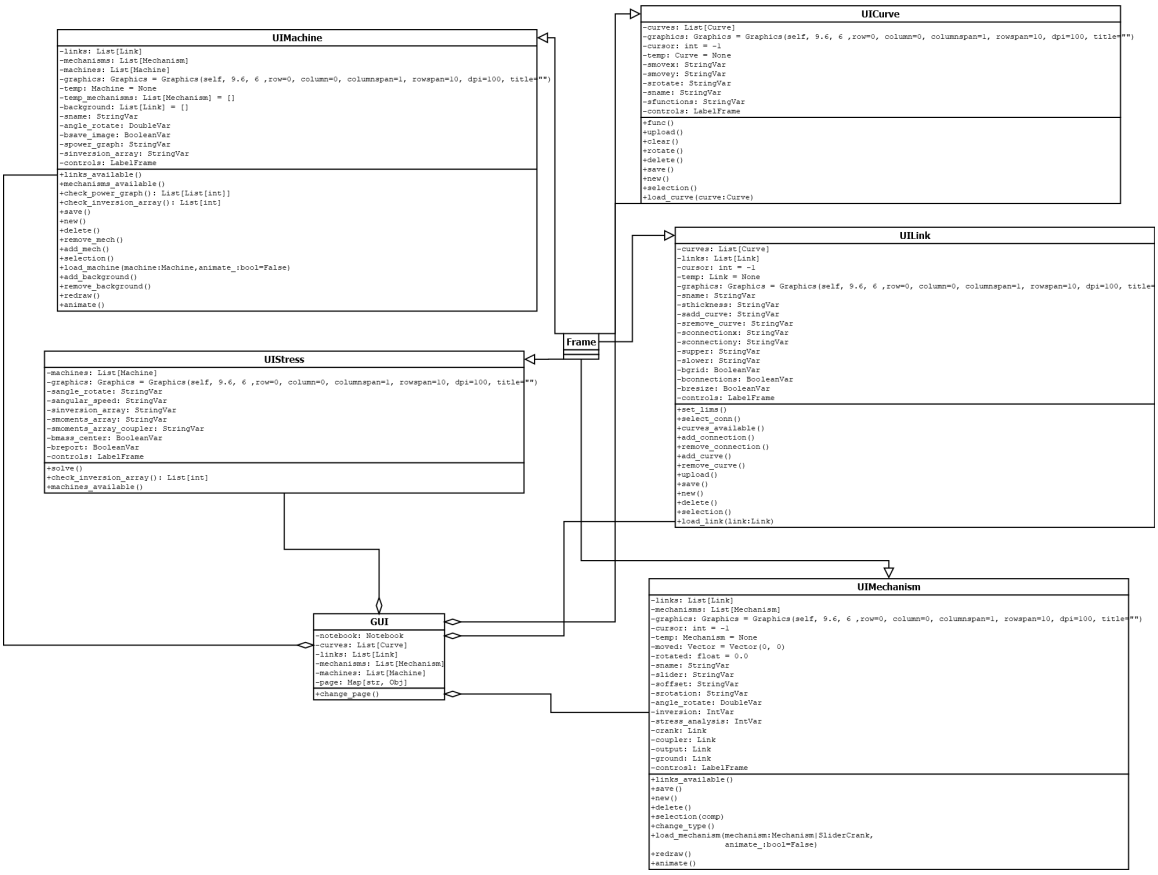


6.2.2. Interfaz gráfica

Posteriormente se procedió con el diseño de la interfaz gráfica para que los usuarios finales sean capaces de interactuar con el software creado. En la Figura 9 se muestra el UML para este módulo del programa.

El marco de trabajo diseñado consiste en una serie de ventanas donde los usuarios pueden contruir y editar los componentes que forman parte de la máquina. En la primera ventana se tiene acceso a la manipulación de curvas, seguidamente se tiene la manipulación de eslabones, mecanismos, máquinas y finalmente el análisis de esfuerzos.

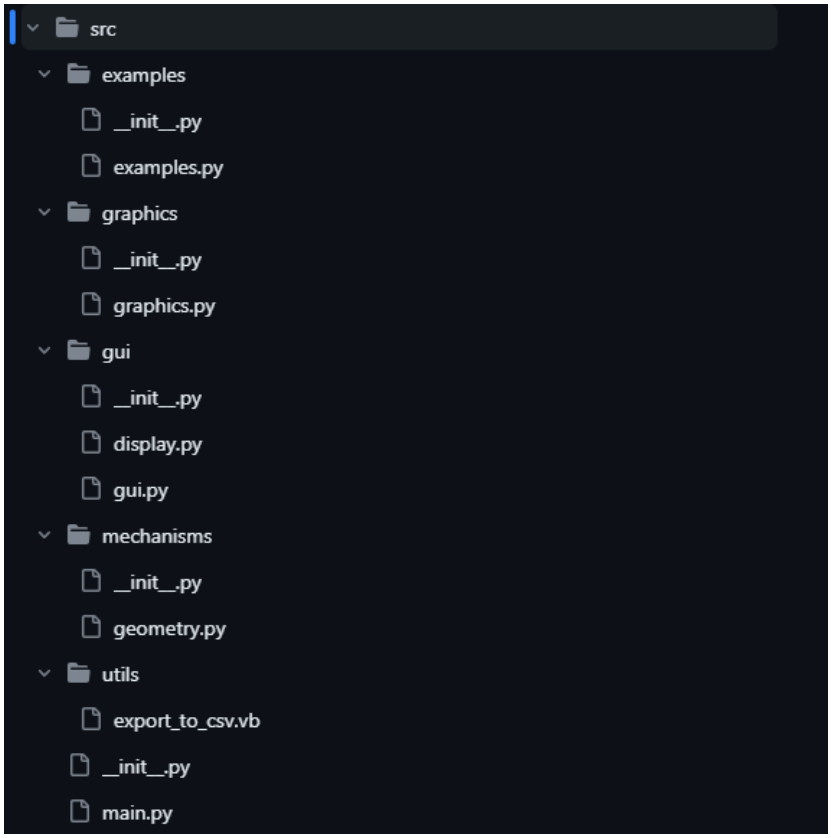
Figura 9: UML interfaz gráfica



6.2.3. Estructura del programa

La estructura del programa se muestra en la Figura 10. Esta consiste de cinco módulos: *examples*, *graphics*, *gui*, *mechanisms*, *utils*. En la sección *gui* se encuentra la implementación de la interfaz gráfica mostrada en la Figura 9. En la sección *mechanisms* se encuentra la implementación de la lógica principal del programa mostrada en la Figura 8. El módulo de *utils* cuenta con la herramienta diseñada para integrar Inventor® con el programa. La sección de examples cuenta con máquinas diseñadas con el programa. La sección de *graphics* cuenta con la implementación de los gráficos para mostrar las animaciones y las configuraciones del sistema para un único ángulo de entrada. Finalmente el archivo main.py representa el punto de entrada a la aplicación.

Figura 10: Estructura del programa



6.3. Cálculos

6.3.1. Cinemática

Posición de eslabones

Previo al análisis de posición de los mecanismos de una máquina es necesario ajustar los eslabones de cada uno de ellos. Para ello durante la inicialización de un mecanismo la bancada se rota al ángulo de inclinación del mecanismo y se desplaza al origen. En el caso del mecanismo cuatro barras juntas de pasador la salida se traslada desde la segunda junta hacia el origen y se rota de tal manera que el vector que va de la segunda junta a la primera junta sea paralelo a la bancada. Este procedimiento es replicado para la manivela y el acoplador solo que para dichos eslabones se debe trasladar la primera junta al origen y el vector formado desde la primera junta a la segunda junta debe ser paralelo a la bancada. En el caso del mecanismo manivela-corredera el acoplador se trata de manera similar a la salida del mecanismo juntas de pasador y la corredera se rota respecto a su única junta el ángulo de inclinación del mecanismo.

De esta manera al momento de obtener las soluciones del mecanismo primero se realiza una copia profunda de los eslabones. Luego, a cada una de las copias se le rotan los grados

Figura 11: Representación grafo de potencia

0{1}1{2,3,4,5,6}2{}3{}4{}5{}6{}

Figura 12: Representación grafo de entradas

[-1 0 1 1 1 1 1]

indicados por cada una de las configuraciones del mecanismo y finalmente se realiza una traslación de los eslabones. Donde la primera junta del acoplador se traslada hacia la segunda junta de la manivela. La corredera es traslada a la segunda junta del acoplador mientras que el eslabón de salida del mecanismo juntas de pasador tiene un desplazamiento de su segunda junta hacia la segunda junta de la bancada. Para el caso de una máquina compuesta por múltiples mecanismos se resuelve mediante un grafo de potencia.

Grafo de potencia

Un grafo de potencia es una estructura empleada para modelar la relación de entrada-salida entre los distintos mecanismos que componen una máquina. Es un grafo unidireccional donde se especifican las relaciones de impulso entre los mecanismos. Las salidas de los mecanismos son las encargadas de impulsar las manivelas de otros mecanismos. La única manivela que tiene la capacidad de impulsar más de un mecanismo es la manivela del primer mecanismo.

Esta estructura tiene múltiples representaciones, a nivel del usuario se representa por una serie de números y llaves como se muestra en la Figura 11. Esta expresión es interpretada de la siguiente manera: 0{1} indica que la manivela principal impulsa únicamente al mecanismo 1; mientras que, el mecanismo 1 impulsa al resto de mecanismos que conforman la máquina.

Para realizar el análisis de posición de una máquina primero se genera un grafo de entradas. El grafo de entradas correspondiente al grafo de potencia de la Figura 11 se muestra en la Figura 12. Este es un arreglo en el que en las posiciones de cada uno de los mecanismos se indica cual es su impulsor. La manivela principal está impulsada por el mecanismo -1 al ser la variable independiente del sistema. Con esta información se realiza un ordenamiento topológico del grafo de potencia para establecer el orden de solución de las posiciones de los mecanismos donde cada mecanismo emplea como entrada la salida del mecanismo impulsor usando como referencia el grafo de entradas.

6.3.2. Dinámica

Centroides

El cálculo de los centroides de los eslabones se realizó mediante métodos numéricos. Para ello se empleó la Ecuación 1 y la Ecuación 2. Donde $U(x)$ y $L(x)$ son funciones por partes que describen el contorno superior e inferior del eslabón respectivamente.

Figura 13: Torques externos

1, 0, 0

$$v_x = \frac{\int x(U(x) - L(x))\,dx}{\int U(x) - L(x)\,dx} \tag{1}$$

$$v_y = \frac{\int (U(x) - L(x))\frac{U(x)+L(x)}{2}\,dx}{\int U(x) - L(x)\,dx} \tag{2}$$

Masas

El cálculo de la masa de los eslabones se realizó con la Ecuación 3 donde t es el espesor del eslabón y d es la densidad del material.

$$m = td \int (U(x) - L(x))\,dx \tag{3}$$

Momentos de inercia

El primer paso para el cálculo del momento de inercia es calcular el momento polar del eslabón con la Ecuación 4. Donde $R(x, y)$ es la distancia del diferencial de área respecto al centroide del eslabón. Posteriormente, se emplea la Ecuación 5 para obtener el momento de inercia.

$$J = \int R(x, y)^2\,dA \tag{4}$$

$$I = Jtd \tag{5}$$

Aceleraciones y fuerzas

Las fuerzas y aceleraciones se determinaron con las ecuaciones presentadas en el marco teórico. La solución del sistema de ecuaciones lineales se hizo con la Ecuación 6. Se creó una interfaz para permitir a los usuarios ingresar torques externos que actúan en la salida y en el acoplador (T_3 y T_4 en las ecuaciones) de cada una de los mecanismos que conforman la máquina. Para ello se ingresan los valores en una lista separada por comas como se muestra en la Figura 13 que implica un torque de 1 N-m aplicado sobre la salida o acoplador del primer mecanismo de una máquina compuesta por tres mecanismos.

$$x = A^{-1}b \tag{6}$$

6.3.3. Esfuerzos

Una vez calculadas las fuerzas que actúan sobre las conexiones de cada uno de los eslabones se realizó una traslación de las mismas a lo largo del perfil de cada eslabón. Posteriormente, se trasnformaron a las coordenadas locales del eslabón mediante la matriz de rotación. A las fuerzas y pares equivalentes se les realizó el cálculo del esfuerzo de von Mises y se presentó el valor máximo al usuario. El esfuerzo normal a lo largo del eje transversal del eslabón es cero por lo que la ecuación simplificada del esfuerzo plano de von Mises se muestra en la Ecuación 7. Donde σ_x^2 es una combinación del esfuerzo normal provocado por la fuerza normal a la sección transversal y el momento flector. Este procedimiento se detalla en la Figura 14, Figura 15 y Figura 16. Para el esfuerzo cortante se usó el valor promedio que es simplemente $\frac{F}{A}$ para reducir el tiempo de cómputo. Además, sus efectos no suelen ser significativos al ser únicamente una fracción del valor de los esfuerzos normales.

Figura 14: Fuerzas en las juntas

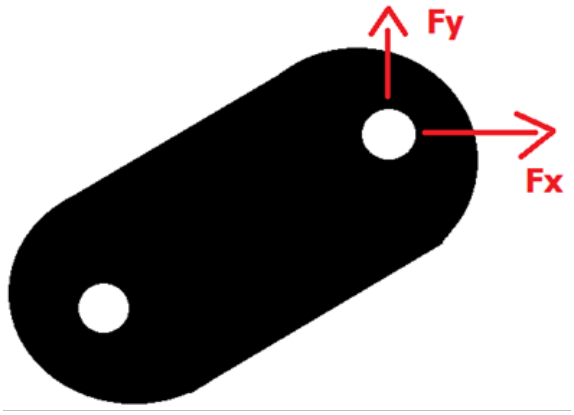


Figura 15: Traslación de fuerzas

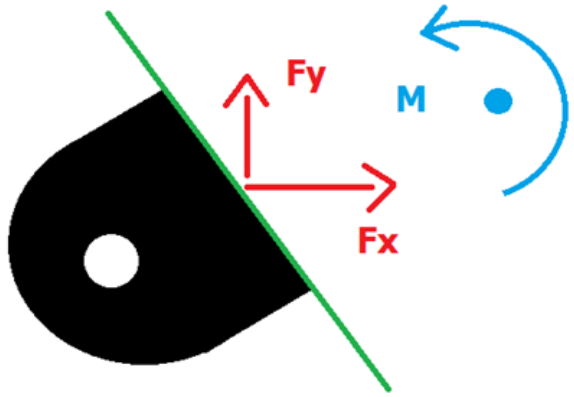
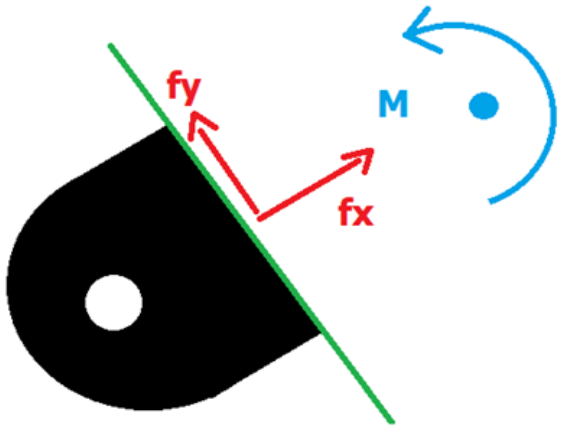


Figura 16: Conversión a coordenadas locales



$$\sigma' = \sqrt{\sigma_x^2 + 3\tau_{xy}^2} \tag{7}$$

6.3.4. Integración con Inventor®

Para integrar la aplicación con Inventor® se desarrolló un guión de código en VBA. Este código puede ser ejecutado como una regla de iLogic para trasladar un boceto cualquiera, preferiblemente en un plano paralelo al plano XY, de Inventor® a un archivo de texto que puede ser interpretado por la aplicación. La herramienta está diseñada para extraer: líneas (por ende cualquier polígono), arcos, círculos, líneas proyectadas, splines por interpolación y los puntos son interpretados como juntas del eslabón. No obstante, no es posible realizar un análisis de esfuerzos a un eslabón creado con esta metodología debido a que se interpretan las entidades únicamente como vectores que no están creados por una función. Esta herramienta se encuentra en la sección Utilidades de los Anexos.

6.3.5. Simulación en ANSYS®

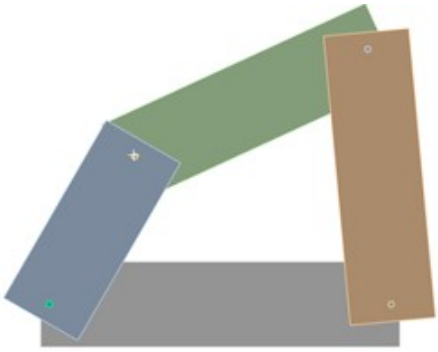
Para evaluar los resultados obtenidos con la herramienta desarrollada se decidió compararlos con un programa de elementos finitos. En el software ANSYS® se realizó un análisis estático estructural de un instante de un mecanismo de cuatro barras obteniendo el esfuerzo equivalente de von Mises y comparándolo directamente con los resultados calculados con el programa.

Geometría

Se decidió analizar un mecanismo sencillo compuesto de cuatro prismas rectangulares para: obtener resultados en el menor tiempo de cómputo posible, evitar discrepancias por concentraciones de esfuerzos y tener un mallado de alta calidad sin realizar mayor ajuste a las geometrías de los eslabones. El mecanismo se presenta en la Figura 17. La manivela es

de una longitud de 1.2m, el acoplador y salida de 1.7m y la bancada de 2.2 metros. Todos los prismas tienen una altura de 0.5m y un espesor de 0.1m. Las juntas son perforaciones de 20mm ubicadas a 0.25m de su base con distancias entre sí de: 1m, 1.5m, 1.5m y 2m para la manivela, acoplador, salida y bancada respectivamente. El ángulo de entrada evaluado es de 60°.

Figura 17: Mecanismo analizado en ANSYS®



Fuente: Imagen utilizada por cortesía de ANSYS, Inc.

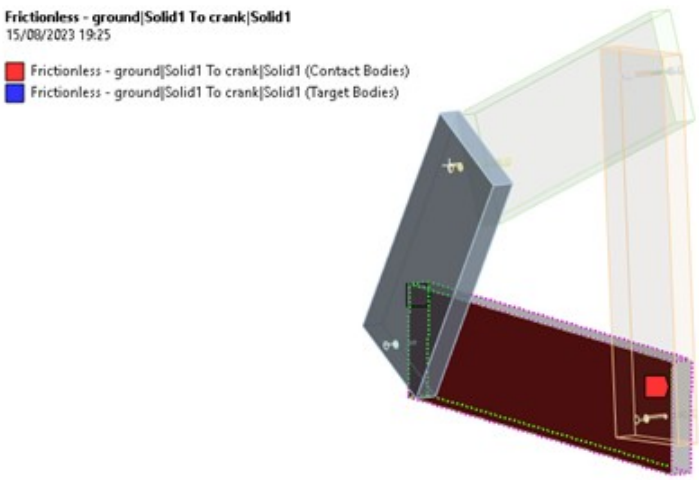
Material

Se asumió que el mecanismo está compuesto de acero estructural del material se requiere la densidad ($7850\frac{kg}{m_3}$) para que la aplicación se capaz de determinar las masas de los eslabones.

Contactos y soportes

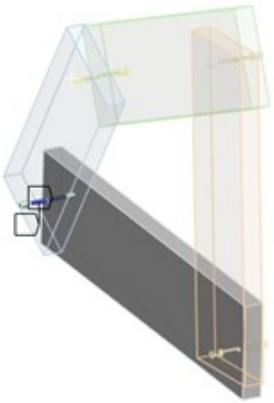
Los contactos entre las piezas se asumieron sin fricción; tanto entre los pines con los prismas como entre prismas, de esta manera se permite la rotación de los eslabones (Figura 18). Para el pin que conecta la bancada con la manivela se estableció un contacto *Bonded* en su superficie con respecto a los dos eslabones para simular el torque de respuesta que la bancada ejerce sobre la manivela para contrarrestar los torques externos (Figura 19). Se asume que la bancada está fija en el suelo por lo que se le asignó un soporte fijo en su base (Figura 20).

Figura 18: Contactos sin fricción



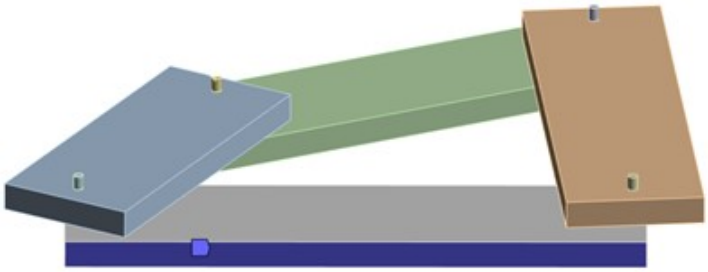
Fuente: Imagen utilizada por cortesía de ANSYS, Inc.

Figura 19: Contactos *bonded*



Fuente: Imagen utilizada por cortesía de ANSYS, Inc.

Figura 20: Soporte fijo

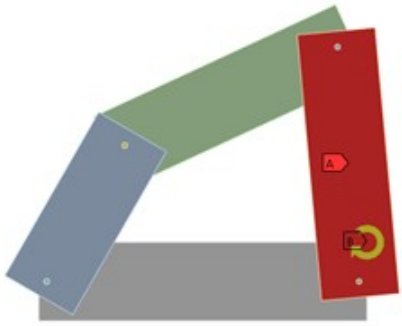


Fuente: Imagen utilizada por cortesía de ANSYS, Inc.

Cargas y velocidades

Para simular la operación del mecanismo se colocó un torque externo de 2N-m en el mecanismo de salida y una velocidad angular de -1$\frac{rad}{s}$ a la manivela (Figura 21).

Figura 21: Cargas y velocidades

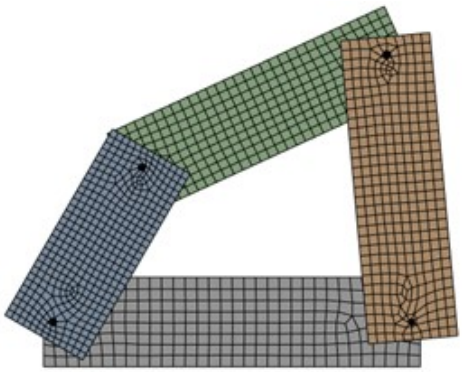


Fuente: Imagen utilizada por cortesía de ANSYS, Inc.

Mallado

Se aplicó un mallado general con una resolución de 4, esto da una asimetría promedio de 0.19315 (Figura 22).

Figura 22: Mallado



Fuente: Imagen utilizada por cortesía de ANSYS, Inc.

CAPÍTULO 7

Resultados

Cuadro 3: Listado de verificacion de requisitos

Código	Descripción	Estado
F01	El programa debe funcionar en Windows 10	✓
F02	Permitir el análisis de eslabones en 2D de espesor constante	✓
F03	El programa debe funcionar en una computadora con una RAM mínima de 8GB	✓
F04	Mostrar resultados con 15 % de error o menos	✓
F05	Integrar el programa con herramientas familiares para los usuarios potenciales	✓
L01	Restringir el perímetro de los eslabones por curvas paramétricas simples (Que no se cruzan)	✓
G01	Crear un módulo que permita construir y editar eslabones a través de texto y una interfaz gráfica	✓
G02	Permitir al usuario construir y editar mecanismos a través de una interfaz gráfica	✓
G03	Permitir al usuario usar botones para realizar los cálculos de esfuerzos de von Mises	✓
G04	Mostrar al usuario resaltados los valores máximos de esfuerzos	✓
G05	Mostrar al usuario las aceleraciones de los eslabones	✓
G06	Mostrar al usuario las fuerzas a las que están sometidos los eslabones	✓
G07	Mostrar al usuario una simulación cinemática para diferentes ángulos de entrada	✓

Figura 23: Interfaz gráfica para editar curvas

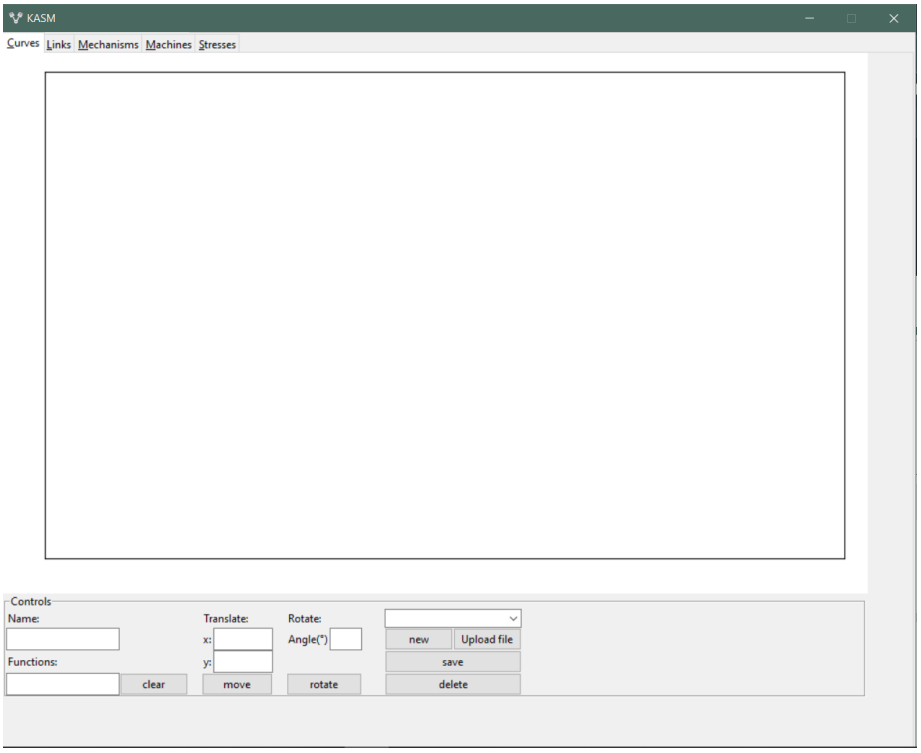


Figura 24: Interfaz gráfica para editar eslabones

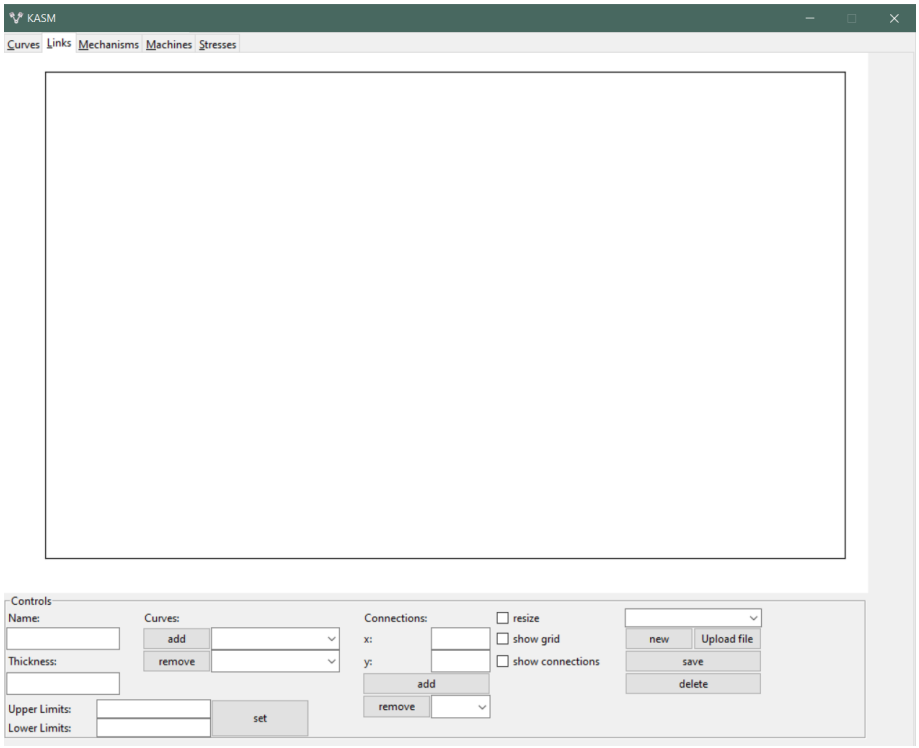


Figura 25: Interfaz gráfica para editar mecanismos

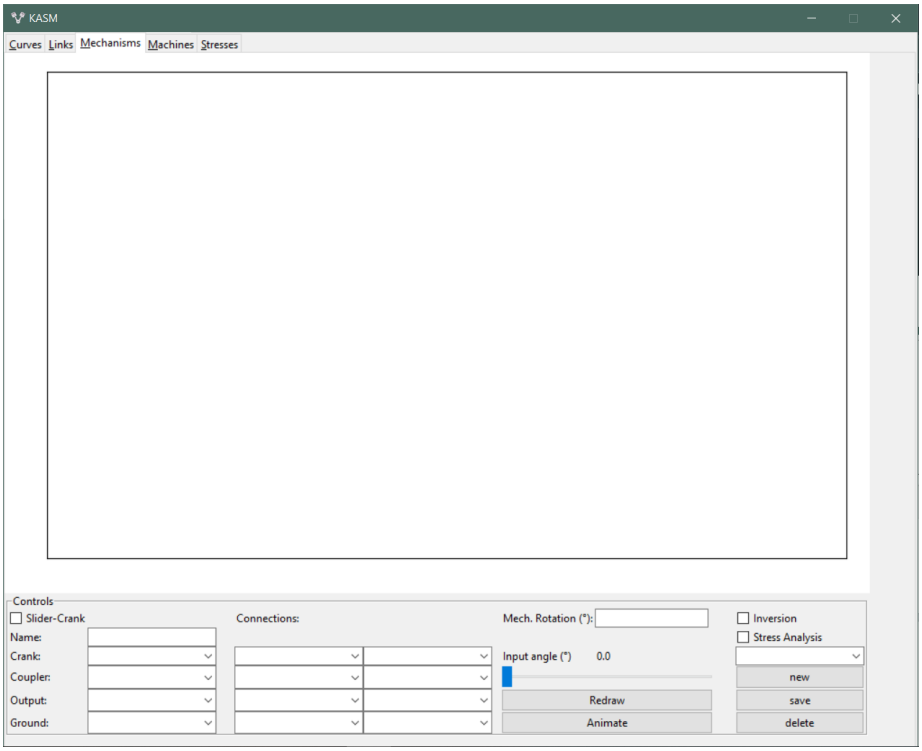


Figura 26: Interfaz gráfica para editar máquinas

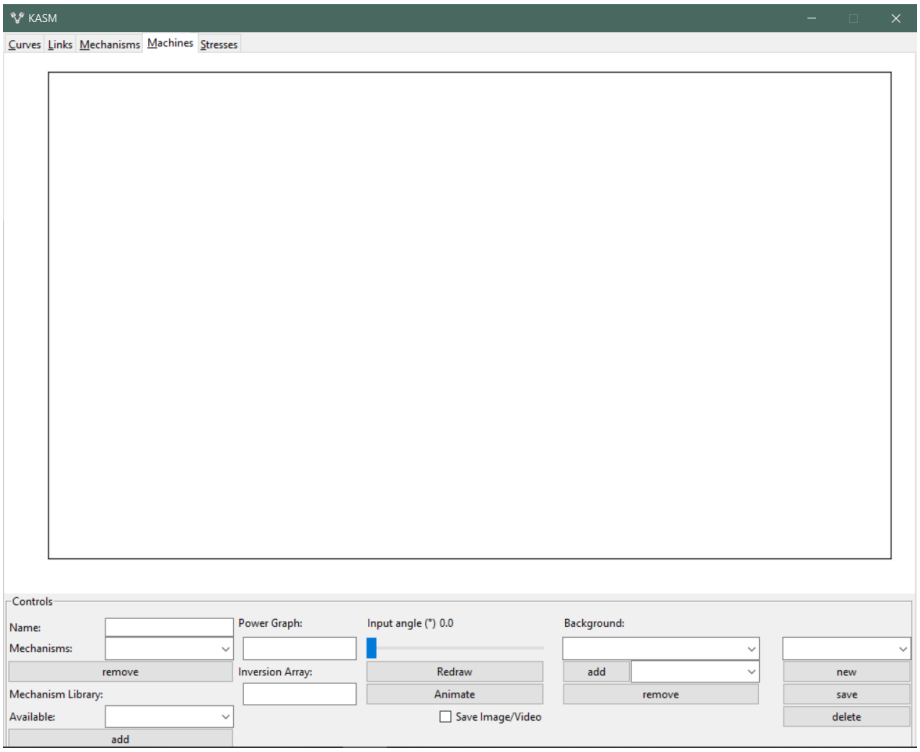


Figura 27: Interfaz gráfica para obtener esfuerzos

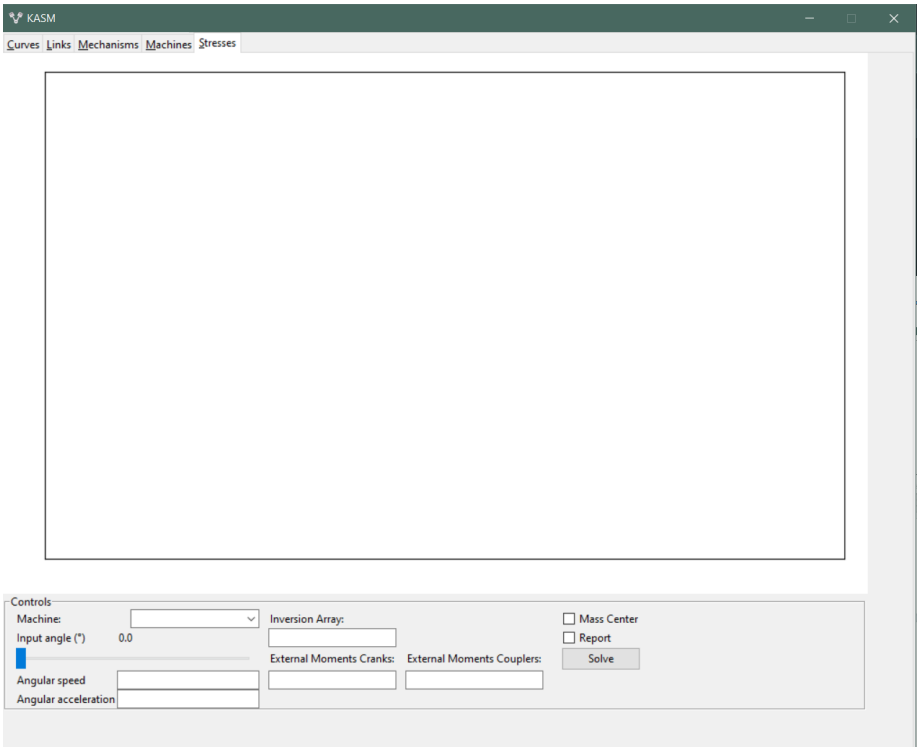


Figura 28: Simulación de mecanismo de cuatro barras

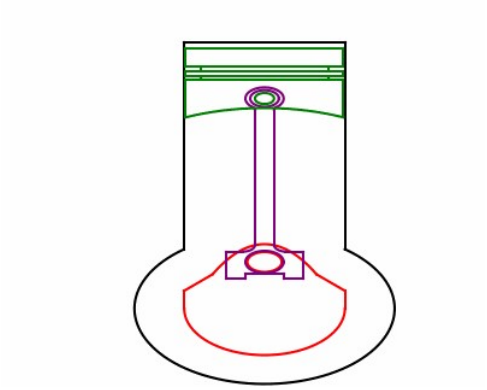


Figura 29: Simulación de mecanismo de 18 barras

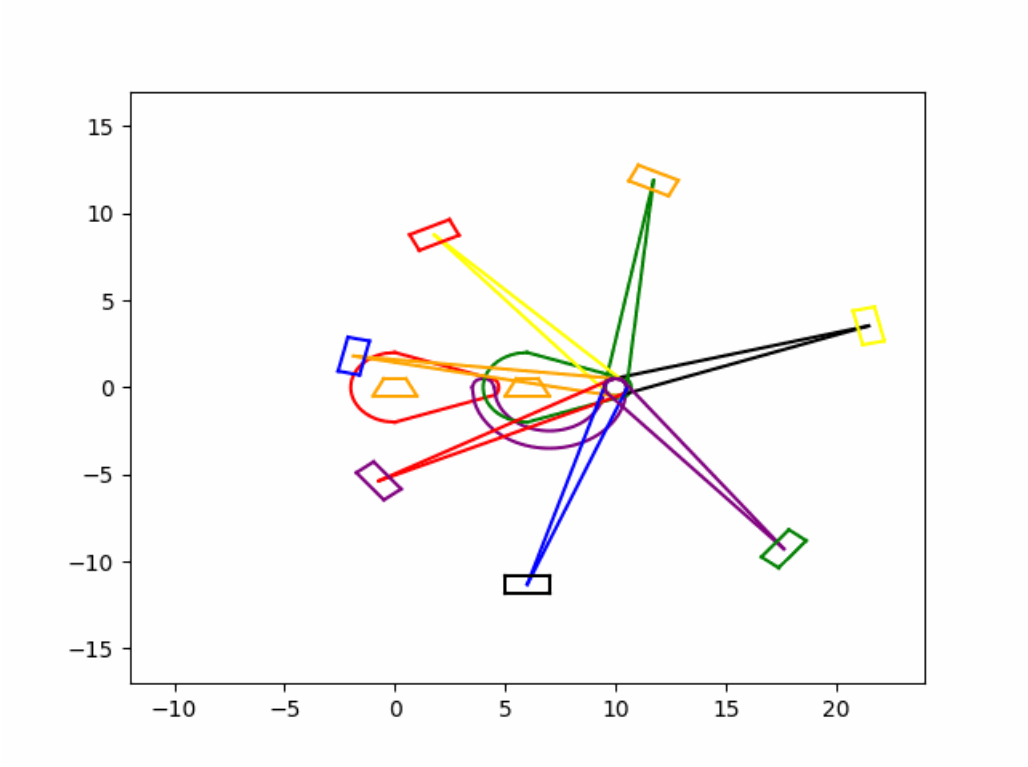


Figura 30: Máquina de 10 barras con centroides

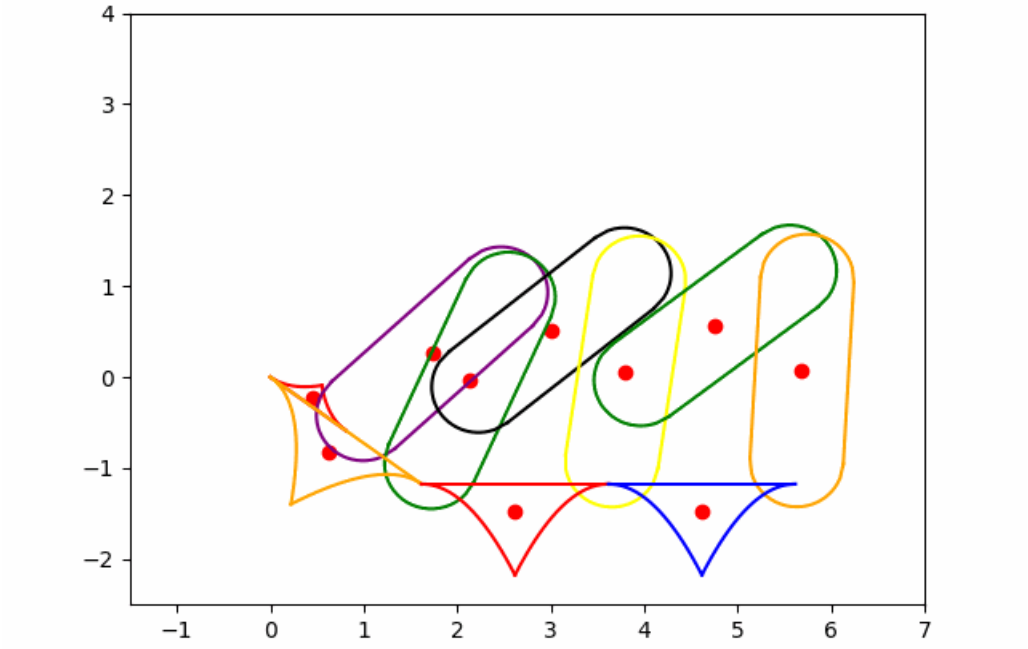
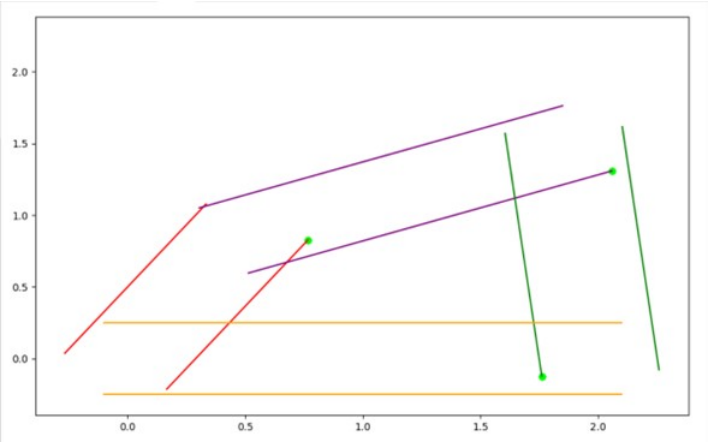
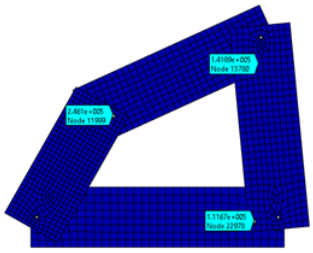


Figura 31: Esfuerzos máximos de von Mises calculados por la aplicación



manivela: 221072.90 Pa, acoplador: 152730.49 Pa, salida: 103490.20 Pa.

Figura 32: Esfuerzos máximos de von Mises calculados en ANSYS®



Fuente: Imagen utilizada por cortesía de ANSYS, Inc.
manivela: 246100 Pa, acoplador: 141890 Pa, salida: 111670 Pa.

Cuadro 4: Fuerzas

Conexión	X (N)	Y (N)
1	-871.02	-693.96
2	753.31	489.99
3	247.92	85.97
4	85.97	29.07

Cuadro 5: Aceleraciones

Eslabón	Ax ($\frac{m}{s^s}$)	Ay ($\frac{m}{s^s}$)	$\alpha(\frac{rad}{s^2})$
1	-0.25	-0.43	0.0
2	-0.75	-0.61	0.46
3	-0.51	-0.17	0.69

Cuadro 6: % de error esfuerzos de von Mises

Eslabón	% Error
1	10.17
2	7.64
3	7.33

Cuadro 7: Esfuerzos en la manivela

Corte (Pa)	Normal (Pa)	Flector (Pa)
8119	16024	204600

Cuadro 8: Esfuerzos en el acoplador

Corte (Pa)	Normal (Pa)	Flector (Pa)
483	5207	147520

Cuadro 9: Esfuerzos en la salida

Corte (Pa)	Normal (Pa)	Flector (Pa)
1887	387	103050

CAPÍTULO 8

Discusión

Se desarrolló un programa que cumple con los requerimientos planteados para analizar mecanismos de n barras de un grado de libertad formados por lazos vectoriales de cuatro eslabones. Las Figuras 24, 25, 26, 27 y 28 muestran la herramienta final con la cual los usuarios son capaces de construir máquinas haciendo uso únicamente de una interfaz gráfica y entradas de texto para definir geometrías, relaciones de entrada-salida y fuerzas externas que actúan sobre los cuerpos. Con esta herramienta fue posible construir las máquinas mostradas en las figuras 28, 29 y 30.

El algoritmo desarrollado para encontrar los esfuerzos equivalentes de von Mises reportó valores con una discrepancia menor al 11 % respecto a las mismas ubicaciones en el *software* de elementos finitos ANSYS®, siendo el valor más elevado en la manivela (Cuadro 6). Este error es producto de no tomar en cuenta las concentraciones de esfuerzos por orificios y filetes; por lo tanto, en máquinas cuyos eslabones cuenten con una mayor cantidad de geometrías que den lugar a concetraciones de esfuerzos se espera que aumente el porcentaje de error. En el caso particular estudiado en la Figura 31 y en la Figura 32 ANSYS® reporta valores de esfuerzo mayores en las juntas producidos por el cambio de geometría. Otra potencial causa de error son los contactos utilizados para modelar al mecanismo al asumir que el momento de respuesta se encuentra concentrado en la conexión *Bonded* entre la bancada y la manivela, cuando en realidad la manivela puede estar potenciada en otra ubicación. El mayor porcentaje de error se encuentra en el eslabón mas alejado del punto de aplicación de la fuerza que puede deberse a la propagación del error al no tomar en cuenta la deformación de los miembros.

Finalmente, en los cuadros 7, 8 y 9 es posible apreciar que el efecto de los esfuerzos cortantes es despreciable siendo el valor máximo en la manivela donde únicamente representa el 3.7 % de los esfuerzos normales.

CAPÍTULO 9

Conclusiones

Se desarrolló un algoritmo para simular y analizar el comportamiento de mecanismos de n barras de un grado de libertad, formados por lazos vectoriales de cuatro eslabones, a través de la implementación de modelos de mecánica de sólidos.

Se creó una interfaz gráfica basada en una serie de pantallas para que los usuarios finales de la aplicación sean capaces de crear máquinas definiendo geometrías, eslabones, mecanismos y finalmente una máquina.

Se implementó un algoritmo basado en la segunda ley de Newton para encontrar las fuerzas y aceleraciones a las que están sujetas los eslabones de una máquina a través de la resolución de una serie de sistemas de ecuaciones lineales.

Fue posible determinar los esfuerzos equivalentes de von Mises que experimentan los eslabones de una máquina que está sometida a momentos externos con un porcentaje de error menor al 15 %.

El programa creado puede ser empleado como una herramienta con fines didácticos para cursos de Dinámica, Diseño Mecánico y Mecanismos. Además, puede formar parte del proceso de diseño de máquinas. Funcionando como una alternativa gratuita y de código abierto a *softwares* pagados y que requieran de mayores recursos computacionales.

CAPÍTULO 10

Recomendaciones

Como puntos de mejora al *software* se propone la implementación de los efectos de la concentración de esfuerzos para obtener valores de los esfuerzos de von Mises con un menor porcentaje de error.

Se sugiere implementar más inversiones de mecanismos y más relaciones de entrada-salida (corredera-manivela, impulso basado en los acopladores) para poder modelar más maquinas.

Asímismo, se aconseja el desarrollo de un algoritmo que permita realizar el análisis de esfuerzos a geometrías definidas únicamente por coordenadas. Posiblemente a través de la implementación de una interpolación polinómica y/o el análisis de áreas definidas por una serie de curvas paramétricas.

Finalmente, se insta a mejorar el algortimo desarrollado para analizar secciones transversales circulares en los eslabones a través de la revolución de curvas. También es posible analizar secciones transversales más complejas al definir las geometrías como funciones multivariables y/o procesar archivos 3D (.ipt, .step, .XYZ).

CAPÍTULO 11

Bibliografía

Bancila, M., Rialdi, R., & Sharma, A. (2020). *Learn C# Programming: A guide to building a solid foundation in C# language for writing efficient programs* [Google-Books-ID: dCLhDwAAQBAJ]. Packt Publishing Ltd.

Barton, L. O. (2015). *Mechanism analysis: simplified graphical and analytical techniques* (Revised and updated edition) [OCLC: 1007644488]. CRC Press.

Donovan, A. A. A., & Kernighan, B. W. (2015). *The go programming language* [Google-Books-ID: SJHvCgAAQBAJ]. Addison-Wesley Professional.

Farrell, J. (2016). *Java programming* (Eighth edition) [OCLC: ocn899704322]. Cengage Learning.

Fozi, H., Marques, G., Pereira, D., & Sherry, D. (2020). *Game Development Projects with Unreal Engine: Learn to build your first games and bring your ideas to life using UE4 and C++* [Google-Books-ID: nV0MEAAAQBAJ]. Packt Publishing Ltd.

Hu, J., Sun, Y., & Cheng, Y. (2016). High mechanical advantage design of six-bar Stephenson mechanism for servo mechanical presses. *Advances in Mechanical Engineering*, *8*(7), 168781401665610. https://doi.org/10.1177/1687814016656108

Hunter, J. D. (2007). Matplotlib: A 2D graphics environment. *Computing in Science & Engineering*, *9*(3), 90-95. https://doi.org/10.1109/MCSE.2007.55

Hussain, F. (2018). *Learn OpenGL: Beginner's guide to 3D rendering and game development with OpenGL and C++* [Google-Books-ID: bvFsDwAAQBAJ]. Packt Publishing Ltd.

Imam, I., Sandor, G. N., & Kramer, S. N. (1973). Deflection and stress analysis in high speed planar mechanisms with elastic links. *Journal of Engineering for Industry*, *95*(2), 541-548. https://doi.org/10.1115/1.3438188

Jeffery, C. (2021). *Build your own programming language* (1st edition) [OCLC: 1291869073]. Packt Publishing.

Jonker, B. (1991). Linearization of dynamic equations of flexible mechanisms—a finite element approach. *International Journal for Numerical Methods in Engineering*, *31*(7), 1375-1392. https://doi.org/10.1002/nme.1620310710

Linkage mechanism designer and simulator – dave's blog. (s.f.). Consultado el 2 de agosto de 2023, desde https://blog.rectorsquid.com/linkage-mechanism-designer-and-simulator/

Llis. (s.f.). Consultado el 2 de agosto de 2023, desde https://llis.nasa.gov/lesson/819

Matzinger, C. (2019). *Rust Programming Cookbook: Explore the latest features of Rust 2018 for building fast and secure apps* [Google-Books-ID: QEm4DwAAQBAJ]. Packt Publishing Ltd.

Mohbey, K. K., & Bakariya, B. (2022). *An introduction to Python programming: a practical approach : using Python to solve complex problems with a burst of machine learning* [OCLC: 1331785986]. BPB Publications.

Nicoll, B., & Keogh, B. (2019). *The unity game engine and the circuits of cultural software* [Google-Books-ID: osKqDwAAQBAJ]. Springer Nature.

Nielsen, J., & Roth, B. (1999). Solving the input/output problem for planar mechanisms. *Journal of Mechanical Design*, *121*(2), 206-211. https://doi.org/10.1115/1.2829445

Norton, R. (2013). *Diseno de maquinaria sintesis y analisis* [OCLC: 1028800844]. MCGRAW HILL.

A novel classification of planar four-bar linkages and its application to the mechanical analysis of animal systems. (1996). *Philosophical Transactions of the Royal Society of London. Series B: Biological Sciences*, *351*(1340), 689-720. https://doi.org/10.1098/rstb.1996.0065

Sclater, N., & Chironis, N. P. (2007). *Mechanisms and mechanical devices sourcebook* (4th ed) [OCLC: ocm76828951]. McGraw-Hill.

Scott, M. L. (2016). *Programming language pragmatics* (Fourth edition) [OCLC: ocn933264678]. Morgan Kaufmann, an imprint of Elsevier.

Sharma, R., Kaihlavirta, V., & Matzinger, C. (2019). *The The Complete Rust Programming Reference Guide: Design, develop, and deploy effective software systems using the advanced constructs of Rust* [Google-Books-ID: kOiZDwAAQBAJ]. Packt Publishing Ltd.

Stroustrup, B. (2013). *The C++ programming language* (Fourth edition). Addison-Wesley.

Tanimoto, S. (2012). *An interdisciplinary introduction to image processing: pixels, numbers, and programs* [OCLC: ocn757133185]. MIT Press.

Teufel, B. (1991). *Organization of programming languages*. Springer.

Thanh, N. (2021). *Javascript programming: introduction to javascript, jquery, ajax, beginner to advanced, practical guide, tips and tricks, easy and comprehensive(Cover java 3)*. Neos Thanh.

Uicker, J. J., Pennock, G. R., & Shigley, J. E. (2017). *Theory of machines and mechanisms* (Fifth edition). Oxford University Press.

Vince, J. (2017). *Mathematics for computer graphics*. Springer Berlin Heidelberg.

Wampler, C. W. (1999). Solving the kinematics of planar mechanisms. *Journal of Mechanical Design*, *121*(3), 387-391. https://doi.org/10.1115/1.2829473

Yang, A. T., Pennock, G. R., & Hsia, L. M. (1981). Stress fluctuation in high speed mechanisms. *Journal of Mechanical Design*, *103*(4), 736-742. https://doi.org/10.1115/1.3254980

Zhao, J., Feng, Z., Chu, F., & Ma, N. (2014). *Advanced theory of constraint and motion analysis for robot mechanisms* [OCLC: ocn876381287]. Academic Press.

CAPÍTULO 12

Anexos

12.1. Manual

La aplicación consiste en una serie de pantallas donde se puede ir construyendo los distintos componentes que forman una máquina.

12.1.1. Instalación

Asegurarse de tener instalado en la computadora el programa Git y Python (preferiblemente de la versión 3.10 en adelante). Acceder a la línea de comandos (*Powershell* para *Windows*, *Bash* para *Linux* y *Terminal* para *Mac*). Seguidamente, ingresar los siguientes comandos:

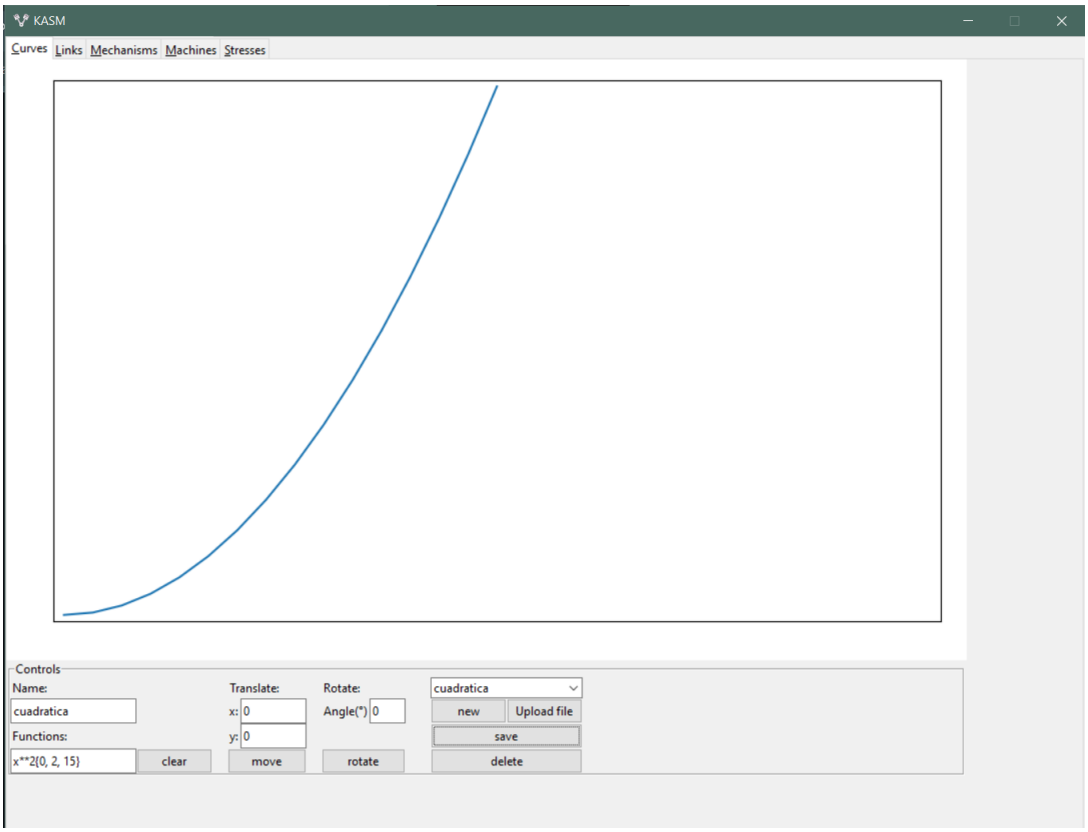
```
$ git clone https://github.com/FernandoLavarreda/KSAM
$ cd KSAM
$ python -m pip install -r requirements.txt
$ cd src
$ python main.py
```

Esto instalará el código en un directorio llamado KSAM en la ubicación donde hayan sido ejecutados los comandos. Un instalador para tener el programa como ejecutable también está disponible. Esto hace que no sea necesario tener Python ni Git instalados en la computadora, pero hace que editar el código en un futuro sea más complicado. Esta disponible únicamente para *Windows* KSAM y se recomienda si únicamente se desea ejecutar el código.

12.1.2. Uso

Pantalla 1: *Curves*

Esta pantalla tiene la utilidad de crear funciones o una serie de vectores que serán empleados para construir eslabones. Para crear una nueva curva el primer paso es presionar el botón ***new***. Esto llenará la mayoría de los campos. El siguiente paso es darle un nombre a la función, por ejemplo cuadrática. En la casilla de ***Functions*** ingresar la siguiente expresión: **x**2{0, 2, 15}**. Luego presionar ***Enter***. Esto dibujará una curva cuadrática $\boldsymbol{x^2}$ que inicia en 0 y finaliza en 2 con 15 entradas linealmente espaciadas $\boldsymbol{dx = (2-0)/15}$ (f(2) siempre será añadida). Posteriormente presionar ***save***, con esto se tiene la primera función. Ahora para crear una línea vertical seguir el mismo procedimiento. Primero presionar ***new*** en el campo de ***Functions*** ingresar *0{0, 4, 2}*. Luego, escribir 90 en el ***Angle(°)*** y presionar ***rotate***. Finalmente, presionar guardar. Luego crear una línea horizontal con la siguiente función: *4{0, 2, 2}*. La creación de funciones acepta: números enteros y decimales, x, sin, cos y tan. Se pueden definir múltiples funciones como parte de una curva al ingresar otro valor en la casilla de ***Functions*** y presionar ***Enter*** una vez más. Todas las funciones estarán conectadas entre sí por una línea que va desde el último vector de la última función ingresada hasta el primer vector de la siguiente función agregada. Para borra la pantalla simplemente presionar ***clear***. Importante notar que la rotación (que siempre se da alrededor de (0, 0)) y/o una traslación de una función invalidará a la función para el análisis de esfuerzos. Presionar ***delete*** eliminará la curva seleccionada en el menú de cascada. La funcionalidad de cargar un archivo se explicará en la siguiente sección.

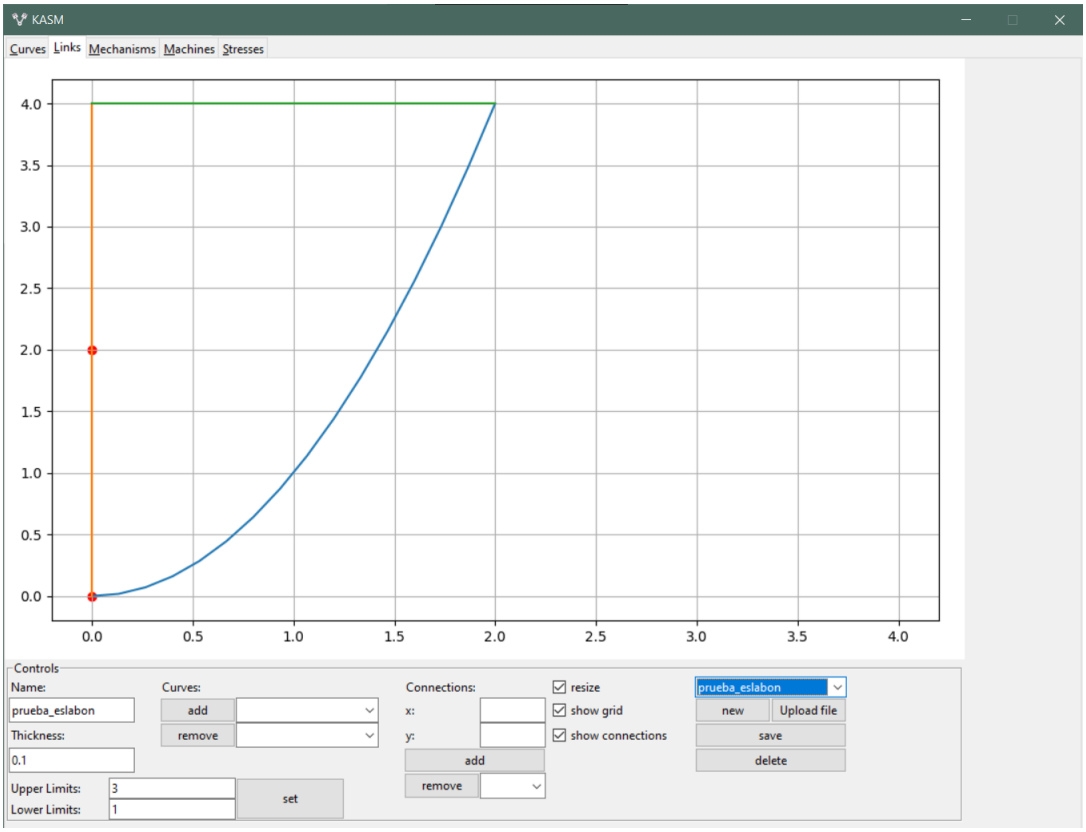


Pantalla 2: *Links*

Esta pantalla está diseñada para construir los eslabones de un mecanismo. Si se presiona el menú de cascada en el extremo derecho una serie de eslabones están predefinidos. Seleccionarlos permite tener una vista previa de ellos. Para crear un nuevo eslabón presionar ***new***. Renombrarlo a *prueba_ eslabon*. En la sección ***Curves*** seleccionar una de las curvas creadas anteriormente y agregarla presionando ***Add***. ***Thickness*** indica el espesor del eslabón (importante trabajar con metros en mente si se desean resultados en pascales, esto igualmente aplica para las dimensiones de las curvas). ***Upper limits*** y ***Lower limits*** se emplean para indicar que funciones definen el perfil de un eslabón si es que se desea un análisis de esfuerzos. Ingresar 3 para límites superiores y 1 para límites inferiores ya que se desea que la función cuadrática represente el límite inferior de la geometría y la línea horizontal represente el límite superior del eslabón. Importante notar que esto asume que al momento de crear el eslabón primero se agrego la función cuadrática, luego la línea vertical y finalmente la línea horizontal. Presionar ***Set***. Múltiples curvas pueden ser empleadas para definir los límites superior e inferior de los eslabones. Asumiendo que se emplean dos funciones para definir el límite superior de un eslabón (f1 y f2) con los dominios [0, 4] y [6, 10]. El programa interpretará una función con un dominio de [0, 10] donde la región (4, 6) será una interpolación lineal entre f1(4) y f2(6). Al añadir varias funciones es importante notar que el programa seleccionará de primero siempre aquella cuyo dominio inicie en el valor más pequeño y cualquier intersección de dominios siempre le dará prioridad a la función menor. Es decir, si f1 tuviera un dominio de [0, 10] y f2 un dominio de [4, 6] la función resultante será idéntica a f1 con un dominio de [0, 10] si en su lugar f2 tuviese un dominio [6, 12] la función se comportará como f1 en [0,10] y como f2 en (10, 12]. El siguiente paso es guardar el eslabón creado (Importante que las curvas inferior y superior tengan el mismo dominio sino resultados incorrectos se presentarán). Las conexiones son las juntas del eslabón. Para que sea más sencillo colocarlas seleccionar todas las *checkboxes* y volver a seleccionar el eslabón *prueba_ eslabon* en el menú de cascada. Ingresar x:0, y:0 y luego presionar ***Enter***, luego ingresar las x:0, y:2. Presionar ***save*** y volver a seleccionar el eslabón en el menú de cascada. La lista entre las conexiones es la longitud efectiva de un eslabón.

Cargar un archivo

Para simplificar la construcción de mecanismos complejos el *softaware* permite cargar archivos csv. En el folder ***utils*** existe un programa desarrollado para integrarlo con iLogic de Inventor®. Esta herramienta espera un bosquejo en el plano XY. Al momento de correrla solicita una ubicación para guardar el archivo y el número del bosquejo que se ha de procesar para ello se pueden emplear números negarivos para hacer referencia al último bosquejo del .ipt/.iam. Para crear herramientas que se integren con KSAM es necesario que los archivos sigan la siguiente convención: la primera línea es ignorada, luego las entidades que crean una curva/curvas deben estar mencionadas seguidas de coordenadas x, y. La entidad puede tener cualquier nombre siempre y cuando esté escrita de la siguiente manera: ***New + cualquier_ - nombre***. Después de esto las coordenadas x, y deben estar presentes (un par de coordenadas por línea). Los puntos no tienen la palabra ***New*** y deben estar nombrados como ***Point*** estos siempre seran interpretados como puntos de conexión para los eslabones. Cargar un archivo

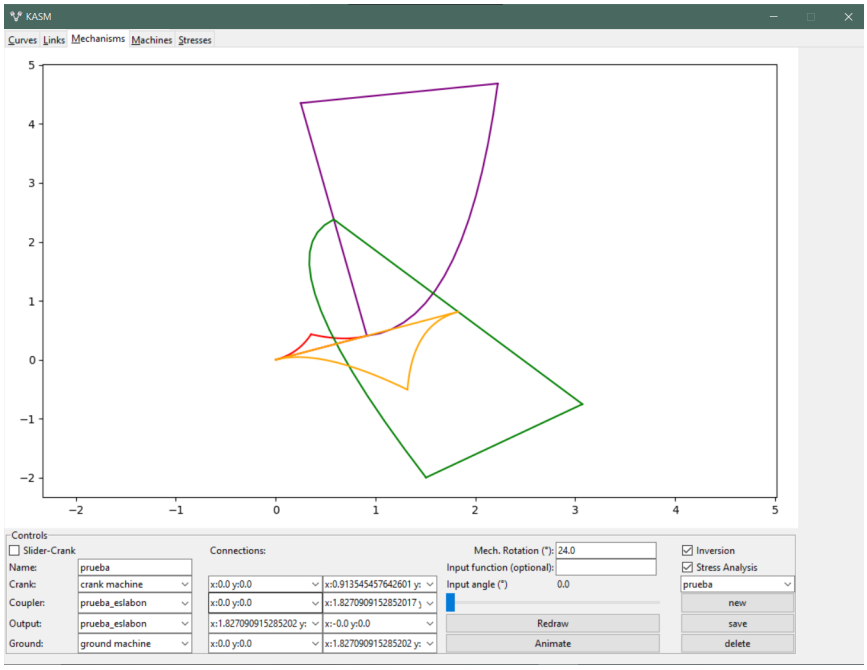


en la pantalla ***Links*** creará una eslabón con varias curvas, pero estas no estarán disponibles en la sección ***Curves***. Los puntos serán los posibles puntos de conexión y las curvas que conforman al eslabón pueden ser removidas. Si el archivo se carga en la pantalla de ***Curves*** se puede cargar a todas las entidades como una única curva (esto no se recomienda) y estarán conectadas entre sí por una línea entre sí según aparezcan en el archivo por lo que el resultado puede no ser el deseado. De lo contrario cada entidad será interpretada como su propia curva con el nombre del archivo y un contador iniciando desde 1. Los puntos serán ignorados en este caso. Esto permite crear mecanismos como la Figura 20.

Pantalla 3: *Mechanisms*

En esta pantalla se pueden crear mecanismos de cuatro barras, ya sea juntas de pasador o manivela-corredera. Existen algunos predefinidos en el menú de cascada que pueden ser visualizados al seleccionarlos. Comenzando por seleccionar el piston 1 y presionar ***Animate*** en este caso inversión es empleado para referirse a las configuraciones abierta/cruzada (no a la inversión en sí). Seleccionar la *checkbox* y volver a animar el mecanismo. ***Redraw*** es empleada para volver a dibujar el mecanismo en la configuración deseada basada en el ángulo de entrada (***Input angle(°)***). Creemos un nuevo mecanismo presionando ***new***. En la casilla de nombre colocar prueba. Seleccionar los eslabones que conformaran el mecanismo como manivela seleccionar ***crank machine*** para el acoplador y la salida seleccionar el eslabón ***prueba_ eslabon*** para la bancada seleccionar ***ground machine***. Luego escoger las juntas empleadas para cada uno de los eslabones. Las juntas están representadas por su

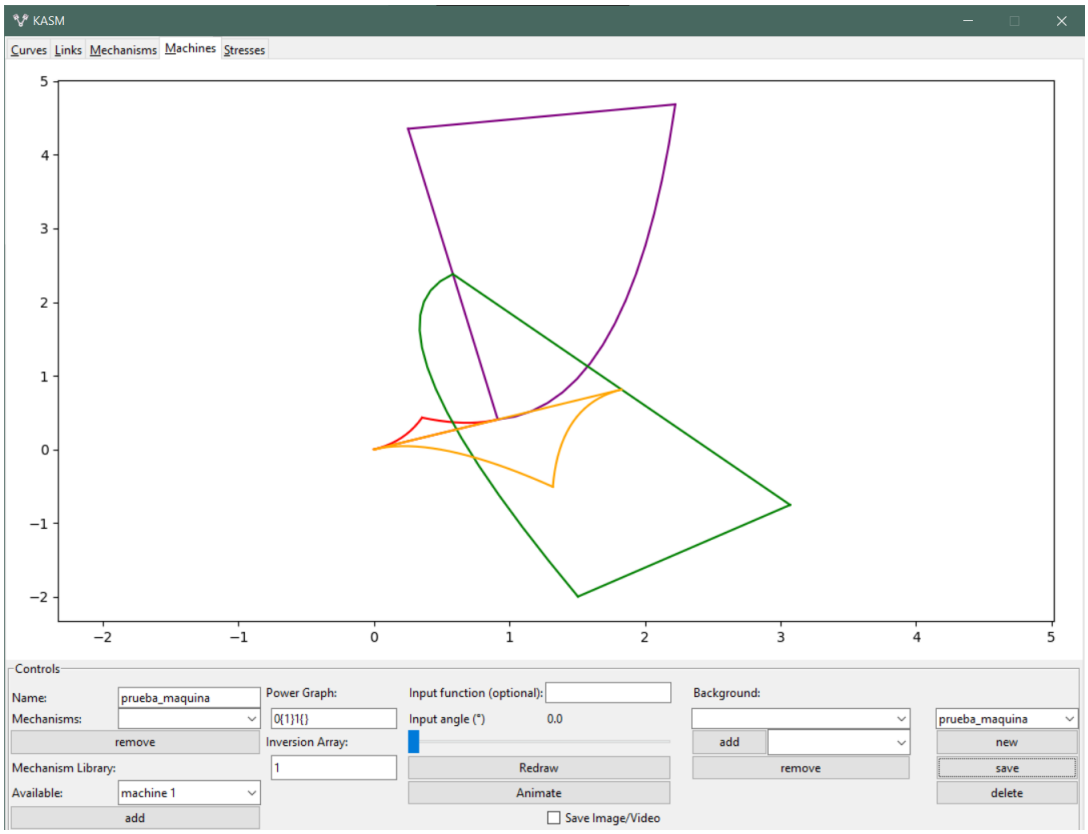
ubicación en el plano. La primera conexión seleccionada para la bancada y para la manivela están conectadas a (0, 0). La primera conexión del acoplador está conectada a la manivela y la primera conexión de la salida está conectada al acoplador. Esto mismo sucede para un mecanismo manivela-corredera que no tiene bancada, pero si un desplazamiento de la corredera con respecto a la primera conexión de la manivela. ***Mech. Rotation(°)*** se refiere a la rotación de la bancada respecto al (0, 0). Para este caso ingresar 60°(Esto significa que el ángulo real del mecanismo será de 24° ya que la bancada que se tomó prestada tiene un rotación inicial de -36°, así que la rotación seleccionada en este campo se adiciona a la rotación de la bancada seleccionda). La opción ***Stress Analysis*** permite encontrar los esfuerzos de un mecanismo para esto se tienen que tener definidos límites superiores e inferiores para cada eslabón (como se hizo anteriormente). Seleccionar la *checkbox* para marcar la opción (así como *Inversion*)y posteriormente presionar ***save***. Esto lanzará dos nuevas ventanas solicitando información primero los diferenciales para realizar el análisis para esto se recomienda 1e-4 para obtener resultados relativamente veloces y precisos. El siguiente paso es ingresar la densidad del material colocar 7850 para simular acero estructural, nuevamente trabajando con $\frac{kg}{m^3}$ para obtener resultados de esfuerzos en pascales. Esto crea el mecanismo después de cierto tiempo. Una función para la entrada a la manivela puede ser ingresada. Por defecto la simulación son 100 valores linealmente espaciados en el rango $[\mathbf{0, 2\pi}]$. Esto puede ser alterado ingresando información al campo ***Input function(°)*** donde se emplea como variable *time* en vez de *x*. Por ejemplo para el piston 1 se puede colocar la siguiente expresión: *3.14159/2*sin(time){0, 6.28, 120}*.



Pantalla 4: ***Machines***

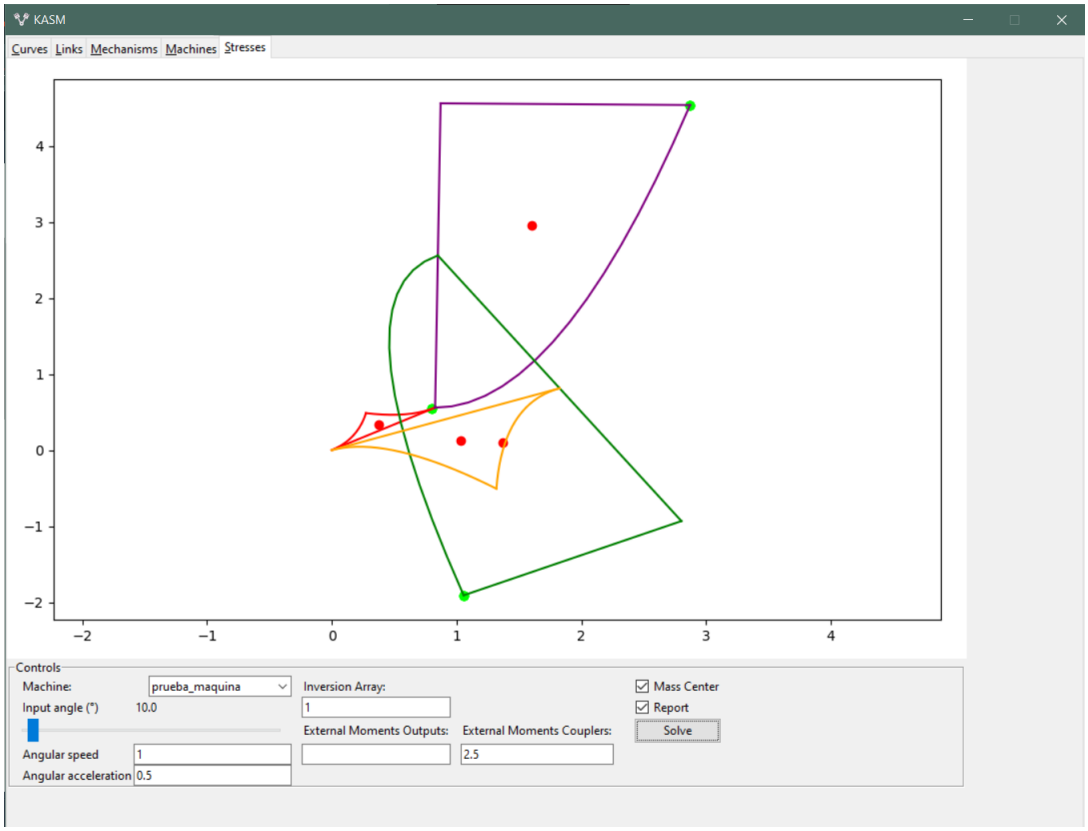
Una máquina está definida como uno o más mecanismos que interactúan entre sí. Igual que en el resto de pantallas hay una serie de ejemplos predefinidos para manipular. Presionar

new para crear una nueva máquina y llamarla *pureba_maquina* en la sección ***Available*** seleccionar prueba y presionar ***Add***. Luego seleccionar ***machine 1*** y agregarla. En la opción de ***power graph*** ingresar *0{1, 2}*. Esto indica que la manivela del primer mecanismo (denomida 0) impulsa a los dos mecanismos del sistema. Los mecanismos son referenciados según su orden de adición a la máquina (similar al caso de las curvas de un eslabón) 1 hace referencia a prueba y 2 a *machine 1*. En la sección de ***Inversion Array*** se puede colocar una serie de 1s y 0s para indicar la configuración deseada de cada uno de los mecanismos. Si se coloca un único 1 o 0 se asumirá la misma configuración para todos los mecanismos. Sino se debe proporcionar la configuración de cada uno de ellos. Colocar como prueba 01 (no dejar espacio entre los números). La sección ***Background*** permite colocar una serie de eslabones que no serán animados solo para mejorar la calidad de la animación (como la cámara de combustión de la Figura 20). Primero presionar ***save*** luego ***Animate*** la manivela de *machine 1* no aparecerá en la animación ya que el mecanismo está impulsado por la salida del mecanismo *prueba*. Los acopladores, bancadas y salidas siempre estarán presentes para cualquier mecanismo. Ahora remover *machine 1*, ingresar 1 en ***Inversion Array*** y 0{1} en ***Power graph***. (Importante notar que el mecanismo 1 siempre debe estar impulsado por su manivela). Luego presionar ***save***. Una animación puede ser guardada seleccionando la *checkbox* ***Save Image/Video*** antes de presionar ***Redraw/Animate***.



Pantalla 5: *Stresses*

Esta pantalla determina el esfuerzo máximo equivalente de von Mises para cada eslabón. Escoger *prueba_maquina* un ángulo de entrada de 10°, una velocidad angular de 1 (trabajando con $\frac{rad}{s}$ para obtener resultados en Pa). Una aceleración angular de 0.5 ($\frac{rad}{s^2}$ para obtener resultados en Pa). Una configuración de 1, momento externo en el acoplador de 2.5N-m. Dejando las salidas vacías. Seleccionar ambas ***checkboxes*** y presionar ***Solve***. Esto solicitará una ubicación para guardar el archivo y solo es de esperar a que se resuelva el mecanismo. Este reporte puede ser abierto en el navegador. Los puntos verdes indican las ubicaciones de máximo esfuerzo y los puntos rojos los centros de masa para los cuales se muestran las aceleraciones. Los valores del reporte están en N, Pa, $\frac{m}{s^2}$ y $\frac{rad}{s^2}$ siempre y cuando se hayan seguido las convenciones presentadas para el resto de entradas.



12.2. Programa principal

```
#Fernando Lavarreda

import os
from gui.gui import GUI
from examples import examples
```

```
icon = os.path.abspath(os.path.join(os.path.dirname(__file__), ".."))+"\\Icon\\pistons.ico"
compresor = examples.build_compresor(3)
machine = examples.build_machine()
vline = examples.build_vline()
power_comp = examples.build_double_crank(5)
gui = GUI(links=compresor.mechanisms[0].links[:]+machine.mechanisms[0].links[:], mechanisms=compresor.mechanisms[:]+machine.mechanisms[:], machines=[compresor, machine, vline, power_comp], icon=icon)
gui.mainloop()
```

12.3. Solución de mecanismos

```
#Fernando Jose Lavarreda Urizar
#Program to analyze Mechanisms, Graduation Project UVG

import csv
import numpy as np
import numpy.linalg as linalg
from typing import List, Tuple, Mapping, Callable
from math import sin, cos, pi, sqrt, atan, atan2, asin


def angle_rad(angle:float):
    return angle*pi/180


def rad_angle(rad:float):
    return rad*180/pi


class Vector:
    def __init__(self, x, y):
        """Representation of 2D-Vector"""
        self.x = x
        self.y = y


    def rotate(self, angle:float):
        self.x, self.y = self.x*cos(angle)-self.y*sin(angle), self.x*sin(angle)+self.y*cos(angle)


    def rotate_angle(self, angle:float):
        self.rotate(angle_rad(angle))
```

```
    def translate(self, x, y):
        self.y+=y
        self.x+=x


    def __add__(self, other):
        return Vector(self.x+other.x, self.y+other.y)


    def __sub__(self, other):
        return Vector(self.x-other.x, self.y-other.y)


    def __str__(self):
        return f"Vector: \nx: {self.x}\ny: {self.y}\n"


    def size(self):
        return sqrt(self.x*self.x+self.y*self.y)


    def copy(self):
        return Vector(self.x, self.y)



def vector_length(a:Vector, b:Vector):
    """Return the distance between two vectors"""
    return sqrt((b.x-a.x)**2+(b.y-a.y)**2)


def vector_angle(a:Vector, b:Vector):
    """Get the angle between two position"""
    return atan2(b.y-a.y, b.x-a.x)


def build_callable_function(string:str):
    """Build function from string"""
    def function(x:float):
        call = string.replace("x", str(x))
        return eval(call)
    return function


class Function:
    def __init__(self, start:float, end:float, process:Callable[[
       float], float]=None, string_function=""):
```

```
    assert start<end, f"Start must be smaller than end, {start
        }, {end}"
    assert (process != None) ^ (string_function != ""), "
        Either process xor string_function must be valid"
    self.start = start
    self.end = end
    if process:
        self.func = process
    else:
        self.func = build_callable_function(string_function)


def __call__(self, value:float):
    if value < self.start or value > self.end:
        raise ValueError(f"{value} out of bounds, limits: [{
            self.start}, {self.end}]")
    return self.func(value)


def centroid(self, dx:float=1e-10):
    nvalue = self.start+dx
    area = 0
    xcoord_numerator = 0
    ycoord_numerator = 0
    while nvalue < self.end:
        compute = abs(self(nvalue)*dx)
        xcoord_numerator+= compute*nvalue
        ycoord_numerator+= compute*self(nvalue)/2
        area+=compute
    centroid_ = Vector(xcoord_numerator/area, ycoord_numerator
        /area)
    return centroid_, area


def __sub__(self, other):
    """Create new Function from substracting one from another
        """
    assert type(other) == type(self), "Can only substract self
        from other Function"
    if self.start < other.start:
        start = other.start
    else:
        start = self.start
    if self.end > other.end:
        end = other.end
    else:
        end = self.end
```

```
    def call_(x:float):
        return self(x)-other(x)

    return Function(start, end, call_)


def __add__(self, other):
    """Create new Function from adding one from another"""
    assert type(other) == type(self), "Can only substract self
        from other Function"
    if self.start < other.start:
        start = other.start
    else:
        start = self.start
    if self.end > other.end:
        end = other.end
    else:
        end = self.end

    def call_(x:float):
        return self(x)+other(x)

    return Function(start, end, call_)


def concatenate(self, other):
    """Create new Function from putting one next to the other,
        smallest start
       Function has precedence. Undefined behaviour is out of
          bounds
       for both functions (space in between) and therefore
          replaced with linear interpolation
       between the last and first value of each one of the
          funcitons"""
    if self.start <= other.start:
        if self.end <  other.end:
            if self.end < other.start:
                m = other(other.start)/self(self.end)
                def function(x:float):
                    if x <= self.end:
                        return self.func(x)
                    elif x < other.start:
                        return m*(x-self.end)+self(self.end)
                    else:
                        return other.func(x)
            else:
                def function(x:float):
                    if x <= self.end:
```

```
                        return self.func(x)
                    else:
                        return other.func(x)
                return Function(self.start, other.end, function)
            return Function(self.start, self.end, self.func)

        other.concatenate(self)



def concatenate_functions(functions:List[Function]):
    ordered = sorted(functions, key=lambda x: x.start)
    fn = ordered[0]
    for f in ordered[1:]:
        fn = fn.concatenate(f)
    return fn



class Curve:
    def __init__(self, origin:Vector, vectors:List[Vector], name="
       ", functions="", function:Function=None):
        """Series of Vectors to create a profile
           origin: Absolute position where the start of the Curve
              is located"""
        self.vectors = vectors
        self.origin = origin
        self.name = name
        self.functions = functions
        self.function = function
        if self.function:
            self.computable_centroid = True
        else:
            self.computable_centroid = False


    @staticmethod
    def build_from_io(file_descriptor, delimiter=",", origin=
       Vector(0, 0), multiple=False):
        reader = csv.reader(file_descriptor, delimiter=delimiter)
        header = next(reader)
        vectors = []
        for line in reader:
            if "New" in line[0]:
                if multiple:
                    vectors.append([])
            elif "Point" in line[0]:
                next(reader)
```

```
            else:
                try:
                    x = float(line[0])
                    y = float(line[1])
                except ValueError:
                    raise ValueError("Unable to process token: ",
                        line[0])
                else:
                    if multiple:
                        vectors[-1].append(Vector(x, y))
                    else:
                        vectors.append(Vector(x, y))
        if multiple:
            curves = []
            for v in vectors:
                curves.append(Curve(origin, v))
            return curves
        return Curve(origin, vectors)


    def rotate(self, angle:float):
        for v in self.vectors:
            v.rotate(angle)
        self.computable_centroid = False


    def rotate_angle(self, angle:float):
        self.rotate(angle_rad(angle))


    def translate(self, x:float, y:float)->Vector:
        for v in self.vectors:
            v.translate(x, y)
        self.computable_centroid = False


    def absolute(self):
        return Curve(self.origin, [v+self.origin for v in self.
            vectors])


    def copy(self):
        curve = Curve(self.origin.copy(), [v.copy() for v in self.
            vectors], self.name, self.functions, self.function)
        curve.computable_centroid = self.computable_centroid
        return curve
```

```
class Link:
    def __init__(self, origin:Vector, connections:List[Vector],
        curves:List[Curve], thickness:float, name="", upper:List[
        Function]=[], lower:List[Function]=[], centroid=None):
         """A link is a segment of a mechanism for which the
            position and stress are desired
            origin: Absolute position where the start of the Link
               is located
            connections: Vectors where another link may be
               connected to create a mechanism
            curves: groups of curve that define the link's shape
            thickness: depth of the link, important only if
               stresses are to be calculated
            name: helps as identifier of the link
            upper: list of Function that define the upper bound of
               the link,
                   only important to define when calculating
                      stresses
            lower: list of Function that define the lower bound of
               the link,
                   only important to define when calculating
                      stresses"""

         self.origin = origin
         self.connections = connections
         self.curves = curves
         self.thickness = thickness
         self.name = name
         self.translation = Vector(0, 0)
         self.rotation = 0
         self.upper = upper
         self.lower = lower
         self.centroid_vector = centroid
         self.mass = 0
         self.moment_inertia_centroid  = 0
         self.element_node_locations = [] #List[Vector] indicating
            the centroid for each element for Finite Element
            Analysis
         self.areas = []
         self.inertias = []
         self.heights = [] #Heights for each node being analyzed,
            required for stresses due to momentum (Mc/I )


    @staticmethod
    def build_from_io(file_descriptor, delimiter=",", origin=
        Vector(0, 0)):
```

```
    reader = csv.reader(file_descriptor, delimiter=delimiter)
    header = next(reader)
    vectors = []
    connections = []
    for line in reader:
        if "New" in line[0]:
            vectors.append([])
        elif "Point" in line[0]:
            advance = next(reader)
            connections.append(Vector(float(advance[0]), float
                (advance[1])))
        else:
            try:
                x = float(line[0])
                y = float(line[1])
            except ValueError:
                raise ValueError("Un able to process token: ",
                    line[0])
            else:
                vectors[-1].append(Vector(x, y))

    curves = []
    for v in vectors:
        curves.append(Curve(origin, v))

    return Link(origin, connections, curves, 0.1)


def rotate(self, angle:float):
    for curve in self.curves:
        curve.rotate(angle)

    for conn in self.connections:
        conn.rotate(angle)

    if self.centroid_vector:
        self.centroid_vector.rotate(angle)
        for v in self.element_node_locations:
            v.rotate(angle)

    self.rotation+=angle


def rotate_angle(self, angle:float):
    self.rotate(angle_rad(angle))


def translate(self, x:float, y:float):
```

```
        for curve in self.curves:
            curve.translate(x, y)

        for conn in self.connections:
            conn.translate(x, y)

        if self.centroid_vector:
            self.centroid_vector.translate(x, y)
            for v in self.element_node_locations:
                v.translate(x, y)

        self.translation+=Vector(x, y)


    def absolute(self):
        return Link(self.origin, [conn+self.origin for conn in
            self.connections], [c.absolute() for c in self.curves],
             thickness, name, upper=self.upper[:], lower=self.lower
            [:])


    def length(self, a, b):
        """Get the length between two connection points in the
            Link, 0 based"""
        assert a < len(self.connections), "a argument is out of
            bounds"
        assert b < len(self.connections), "b argument is out of
            bounds"
        return vector_length(self.connections[a], self.connections
            [b])


    def set_lims(self, indexes:List[int], lims=1):
        """Set functions defining limits of the link to calculate
            stresses"""
        functions_ = []
        for v in indexes:
            if v >= len(self.curves) or v < 0:
                raise ValueError(f"{v} is out of bounds, Curves is
                    only size {len(self.curves)}")
            if self.curves[v].function == None or not self.curves[
                v].computable_centroid:
                raise ValueError(f"Curve {self.curves[v]} with
                    index {v} has no function defining it or has
                    been rotated, therefore is not valid")
            functions_.append(self.curves[v].function)

        if lims:
```

```
            self.upper = functions_
        else:
            self.lower = functions_


    def centroid(self, dx:float, density:float, tolerance:float=1e
        -3):
        """To speed up calculation intesection between curves is
            not evaluated,
            make sure upper curve is above lower curve throughout
                the domain
            centroid must be computed before translating and/or
                rotating the Link
            proceed with the calculation of mass and moment of
                intertia around centroid"""
        if not self.upper or not self.lower:
            raise ValueError("Either lower or upper limits are
                missing")

        up_function = concatenate_functions(self.upper)
        lo_function = concatenate_functions(self.lower)
        if abs(up_function.start-lo_function.start)>tolerance or
            abs(up_function.end-lo_function.end)>tolerance:
            raise ValueError("End and start of functions defining
                the upper and lower bounds is greater than
                tolerance accepeted")
        self.element_node_locations = []
        self.areas = []
        self.inertias = []
        self.heights = []
        area = 0
        position = dx/2+up_function.start
        ycoord_numerator = 0
        xcoord_numerator = 0 #Repeating similar process to
            Function computation of centroid because the y
            coordinate will not be accurate when substracting the
            upper and lower bounds of the link
        while position < up_function.end:
            evaluate_high = up_function(position)
            evaluate_low = lo_function(position)
            compute = dx*(evaluate_high-evaluate_low)
            area+=compute
            y_centroid = (evaluate_high-evaluate_low)/2+
                evaluate_low
            xcoord_numerator+=position*compute
            ycoord_numerator+=y_centroid*compute
            self.heights.append((evaluate_high-evaluate_low)/2)
            self.element_node_locations.append(Vector(position,
```

```
            y_centroid))
        self.areas.append(self.thickness*(evaluate_high-
            evaluate_low))
        self.inertias.append(self.thickness/12*(evaluate_high-
            evaluate_low)*(evaluate_high-evaluate_low)*(
            evaluate_high-evaluate_low)) #Second moment of area
        position+=dx



    self.centroid_vector = Vector(xcoord_numerator/area,
        ycoord_numerator/area)
    self.area = area
    position = dx/2+up_function.start
    dy = dx
    darea = dx*dy
    xo = self.centroid_vector.x
    yo = self.centroid_vector.y
    polar_moment_area = 0
    while position < up_function.end:
        evaluate_high = up_function(position)
        evaluate_low = lo_function(position)
        compute = (evaluate_high-evaluate_low)
        positiony = evaluate_low+dy/2

        precom_pute = (position-xo)*(position-xo)
        while positiony < evaluate_high:
            polar_moment_area+=darea*(precom_pute+(positiony-
                yo)*(positiony-yo))
            positiony+=dy

        position+=dx
    self.mass = self.thickness*area*density
    self.moment_inertia_centroid = self.thickness*
        polar_moment_area*density
    return self.centroid_vector


def copy(self):
    link = Link(self.origin.copy(), [conn.copy() for conn in
        self.connections], [c.copy() for c in self.curves],
        self.thickness, self.name, upper=self.upper[:], lower=
        self.lower[:])
    if self.centroid_vector:
        link.centroid_vector = self.centroid_vector.copy()
        link.mass = self.mass
        link.moment_inertia_centroid = self.
            moment_inertia_centroid
```

```
            link.element_node_locations = [i.copy() for i in self.
                element_node_locations]
            link.areas = self.areas[:]
            link.inertias = self.inertias[:]
            link.heights = self.heights[:]
        return link



class Mechanism:
    def __init__(self, origin:Vector, rotation:float, links:List[
        Link], connections:List[Tuple[int, int]], init=True, name="
        ", stress_analysis:bool=False, dx:float=0, density:float=0)
        :
        """Representation of 4 bar mechanism, must be ordered as
            follows: a (crank), b (coupler), c (output), d (bench/
            ground)
            connections: numbers indicating the connection points
                from the links to use"""
        assert len(connections) == len(links), "Missmatch between
            connections and links"
        assert len(links) == 4, "Can only solve for 4 bar
            mechanism"
        self.name = name
        self.origin = origin
        self.links = [l.copy() for l in links]
        self.connections = connections
        self.moved = Vector(0,0)
        self.idisplacement = Vector(0, 0)
        self.a = links[0].length(connections[0][0], connections
            [0][1])
        self.b = links[1].length(connections[1][0], connections
            [1][1])
        self.c = links[2].length(connections[2][0], connections
            [2][1])
        self.d = links[3].length(connections[3][0], connections
            [3][1])
        self.k1 = self.d/self.a
        self.k2 = self.d/self.c
        self.k3 = (self.a*self.a-self.b*self.b+self.c*self.c+self.
            d*self.d)/(2*self.a*self.c)
        self.k4 = self.d/self.b
        self.k5 = (self.c*self.c-self.d*self.d-self.a*self.a-self.
            b*self.b)/(2*self.a*self.b)
        self.size = (self.a+self.b)*1.1
        self.stress_analysis = stress_analysis

        if stress_analysis:
```

```
            assert dx>0, "Requiring a dx>0 for stress analysis"
            assert density>0, "Requiring a density>0 for stress
               analysis"
            for link in self.links:
                link.centroid(dx, density)

        if rotation > 2*pi:
            self.rotation = angle_rad(rotation)
        else:
            self.rotation = rotation


        if init:
            self.initial_placement()


    def location(self):
        return self.origin+self.moved



    def initial_placement(self):
        """This method allows to change the rotation and position
           of a mechanism that has already been created
           Useful when copying a mechanism and desiring to change
              rotation and location of the new one"""
        self.links[3].rotate(self.rotation)
        self.idisplacement = self.origin-self.links[3].connections
           [self.connections[3][0]]
        self.links[3].translate(self.idisplacement.x, self.
           idisplacement.y)

        for i in range(3):

            if i == 2:
                self.links[i].translate(-self.links[i].connections
                   [self.connections[i][1]].x, -self.links[i].
                   connections[self.connections[i][1]].y)
                self.links[i].rotate(vector_angle(self.links[i].
                   connections[self.connections[i][1]], self.links
                   [i].connections[self.connections[i][0]])+self.
                   rotation+pi)
                #Do not know why but this fixed the error
                if abs(vector_angle(self.links[i].connections[self
                   .connections[i][1]], self.links[i].connections[
                   self.connections[i][0]])+self.rotation+pi-pi*2)
                   <1e-10:
                    self.links[i].rotate(pi)
```

```
            else:
                self.links[i].translate(-self.links[i].connections
                    [self.connections[i][0]].x, -self.links[i].
                    connections[self.connections[i][0]].y)
                self.links[i].rotate(-vector_angle(self.links[i].
                    connections[self.connections[i][0]], self.links
                    [i].connections[self.connections[i][1]])+self.
                    rotation)


    def copy(self):
        mechanism = Mechanism(self.origin.copy(), self.rotation,
            links=self.links[:], connections=[i for i in self.
            connections], init=False, name=self.name)
        mechanism.stress_analysis = self.stress_analysis
        mechanism.translate(self.moved.x, self.moved.y)
        return mechanism


    def translate(self, x:float, y:float):
        self.moved += Vector(x, y)


    def rotate(self, angle:float):
        movement_ground = self.idisplacement + self.moved
        self.links[3].translate(-movement_ground.x, -
            movement_ground.y)
        self.links[3].rotate(angle)
        for i in range(3):
            self.links[i].rotate(angle)
        self.links[3].translate(movement_ground.x, movement_ground
            .y)
        self.rotation+=angle


    def rotate_angle(self, angle:float):
        self.rotate(angle_rad(angle))


    def output_rad(self, input_rad:float):
        """input_rad: crank angle relative to ground in rads
        raise a ValueError if the crank can't be placed in that
            position"""
        A = cos(input_rad)-self.k1-self.k2*cos(input_rad)+self.k3
        B = -2*sin(input_rad)
        C = self.k1-(self.k2+1)*cos(input_rad)+self.k3

        try:
```

```
            solution_a = 2*atan((-B+sqrt(B*B-4*A*C))/2/A)
            solution_b = 2*atan((-B-sqrt(B*B-4*A*C))/2/A)
        except ValueError:
            if (B*B-4*A*C) < -1e-15:
                raise ValueError("Crank can't be put in that
                    position")
            else:
                solution_a = 2*atan((-B)/2/A)
                solution_b = solution_a
        except ZeroDivisionError:
            if (-B+sqrt(B*B-4*A*C)) < 0:
                solution_a = -pi
            else:
                solution_a = pi
            if (-B-sqrt(B*B-4*A*C)) < 0:
                solution_b = -pi
            else:
                solution_b = pi
        return solution_a, solution_b


    def output_angle(self, input_angle:float):
        """input_angle: crank angle relative to ground in degrees
        raise a ValueError if the crank can't be placed in that
            position"""
        a, b = self.output_rad(angle_rad(input_angle))
        return rad_angle(a), rad_angle(b)


    def coupler_rad(self, input_rad:float):
        """input_rad: crank angle relative to ground in rads
        raise a ValueError if the crank can't be placed in that
            position"""
        D = cos(input_rad)-self.k1+self.k4*cos(input_rad)+self.k5
        E = -2*sin(input_rad)
        F = self.k1+(self.k4-1)*cos(input_rad)+self.k5

        try:
            solution_a = 2*atan((-E+sqrt(E*E-4*D*F))/2/D)
            solution_b = 2*atan((-E-sqrt(E*E-4*D*F))/2/D)
        except ValueError:
            raise ValueError("Crank can't be put in that position"
                )

        return solution_a, solution_b


    def coupler_angle(self, input_angle:float):
```

```
        """input_angle: crank angle relative to ground in degrees
        raise a ValueError if the crank can't be placed in that
            position"""
        a, b = coupler_rad(angle_rad(input_angle))
        return rad_angle(a), rad_angle(b)


    def angular_velocity(self, input_rad1:float, input_rad2:float,
        dt:float, inversion:int):
        """input_rad1: crank angle relative to ground in rads
            input_rad2: second crank angle relative to ground in
                rads
            dt: time between the two positions
            Ground has an angular speeds of 0
        raise a ValueError if the crank can't be placed in that
            position"""
        assert inversion == 1 or inversion == 0, "Inversion can
            only by 0 or 1"
        w_crank = (input_rad2-input_rad1)/dt
        w_coupler = (self.coupler_rad(input_rad2)[inversion]-self.
            coupler_rad(input_rad1)[inversion])/dt
        w_output = (self.output_rad(input_rad2)[inversion]-self.
            output_rad(input_rad1)[inversion])/dt
        return w_crank, w_coupler, w_output, 0


    def angular_velocity_angular_acceleration(self, input_rad:
        float, crank_angular_speed_rad:float,
        crank_angular_acceleration_rad:float, inversion:int):
        """Obtain tuples with angular speed and acceleration of
            each link"""
        coupler_rad = self.coupler_rad(input_rad)[inversion]
        output_rad = self.output_rad(input_rad)[inversion]
        coupler_speed  = crank_angular_speed_rad*self.a/self.b*sin
            (output_rad-input_rad)/sin(coupler_rad-output_rad)
        output_speed = crank_angular_speed_rad*self.a/self.c*sin(
            input_rad-coupler_rad)/sin(output_rad-coupler_rad)

        A = self.c*sin(output_rad)
        B = self.b*sin(coupler_rad)
        C = self.a*crank_angular_acceleration_rad*sin(input_rad)+
            self.a*crank_angular_speed_rad**2*cos(input_rad)+self.b
            *coupler_speed**2*cos(coupler_rad)-self.c*output_speed
            **2*cos(output_rad)
        D = self.c*cos(output_rad)
        E = self.b*cos(coupler_rad)
        F = self.a*crank_angular_acceleration_rad*cos(input_rad)-
            self.a*crank_angular_speed_rad**2*sin(input_rad)-self.b
```

```
            *coupler_speed**2*sin(coupler_rad)+self.c*output_speed
            **2*sin(output_rad)

        coupler_acceleration = (C*D-A*F)/(A*E-B*D)
        output_acceleration = (C*E-B*F)/(A*E-B*D)
        ground_speed = 0
        ground_acceleration = 0
        return [(crank_angular_speed_rad,
            crank_angular_acceleration_rad), (coupler_speed,
            coupler_acceleration), (output_speed,
            output_acceleration), (ground_speed,
            ground_acceleration)]


    def solution(self, angle_rad:float):
        """Absolute position given an input angle for crank in
            radians
            Return value might be: one solution repeated twice, two
                different solutions
            or ValueError if this position is not possible"""
        ca, cb = self.coupler_rad(angle_rad)
        oa, ob = self.output_rad(angle_rad)

        crank = self.links[0].copy()
        coupler = self.links[1].copy()
        coupler2 = self.links[1].copy()
        output = self.links[2].copy()
        output2 = self.links[2].copy()
        ground = self.links[3].copy()


        crank.rotate(angle_rad)
        coupler.rotate(ca)
        coupler2.rotate(cb)
        output.rotate(oa)
        output2.rotate(ob)

        crank.translate(self.origin.x, self.origin.y)
        crank.translate(self.moved.x, self.moved.y)
        ground.translate(self.moved.x, self.moved.y)
        coupler.translate(crank.connections[self.connections
            [0][1]].x, crank.connections[self.connections[0][1]].y)
        coupler2.translate(crank.connections[self.connections
            [0][1]].x, crank.connections[self.connections[0][1]].y)
        output.translate(ground.connections[self.connections
            [3][1]].x, ground.connections[self.connections[3][1]].y
            )
        output2.translate(ground.connections[self.connections
```

```
            [3][1]].x, ground.connections[self.connections[3][1]].y
            )


        return [crank, coupler, output, ground], [crank, coupler2,
            output2, ground]



class SliderCrank:
    def __init__(self, origin:Vector, rotation:float, links:List[
        Link], connections:List[Tuple[int, int]], offset:float=0,
        init=True, name="", stress_analysis:bool=False, dx:float=0,
         density:float=0):
        """Representation of 4 bar mechanism, must be ordered as
            follows: a (crank), b (coupler), c (output), d (bench/
            ground)
            Ground is an optional addition to the links
            connections: numbers indicating the connection points
                from the links to use"""
        assert len(connections) == len(links), "Missmatch between
            connections and links"
        assert len(links) == 3 or len(links) == 4, "Can only solve
            for 4 bar mechanism, ground link is optional"

        self.name = name
        self.origin = origin
        self.links = [l.copy() for l in links]
        self.connections = connections
        self.moved = Vector(0,0)
        self.idisplacement = Vector(0, 0)
        self.a = links[0].length(connections[0][0], connections
            [0][1])
        self.b = links[1].length(connections[1][0], connections
            [1][1])
        self.c = offset
        self.size = (self.a+self.b)*1.1
        self.stress_analysis = stress_analysis

        if stress_analysis:
            assert dx>0, "Requiring a dx>0 for stress analysis"
            assert density>0, "Requiring a density>0 for stress
                analysis"
            for link in self.links:
                link.centroid()

        #Ground optional
        if len(links) == 3:
```

```
            gg = Curve(Vector(0, 0), [Vector(0,0,),])
            self.links.append(Link(Vector(0, 0), [Vector(0, 0)], [
                gg,], 0.0))
            self.connections.append([0,])

        if rotation > 2*pi:
            self.rotation = angle_rad(rotation)
        else:
            self.rotation = rotation


        if init:
            self.initial_placement()


    def location(self):
        return self.origin+self.moved


    def initial_placement(self):
        """This method allows to change the rotation and position
            of a mechanism that has already been created
            Useful when copying a mechanism and desiring to change
                rotation and location of the new one"""
        self.links[3].rotate(self.rotation)
        self.idisplacement = self.origin-self.links[3].connections
            [self.connections[3][0]]
        self.links[3].translate(self.idisplacement.x, self.
            idisplacement.y)

        for i in range(3):

            if i == 2:
                self.links[i].translate(-self.links[i].connections
                    [self.connections[i][0]].x, -self.links[i].
                    connections[self.connections[i][0]].y)
                self.links[i].rotate(self.rotation)
            elif i == 1:
                self.links[i].translate(-self.links[i].connections
                    [self.connections[i][1]].x, -self.links[i].
                    connections[self.connections[i][1]].y)
                self.links[i].rotate(-vector_angle(self.links[i].
                    connections[self.connections[i][0]], self.links
                    [i].connections[self.connections[i][1]])+self.
                    rotation+pi)
            else:
                self.links[i].translate(-self.links[i].connections
                    [self.connections[i][0]].x, -self.links[i].
```

```
                connections[self.connections[i][0]].y)
            self.links[i].rotate(-vector_angle(self.links[i].
                connections[self.connections[i][0]], self.links
                [i].connections[self.connections[i][1]])+self.
                rotation)


def copy(self):
    slider_crank = SliderCrank(self.origin.copy(), self.
        rotation, links=self.links, connections=[i for i in
        self.connections], offset=self.c, init=False, name=self
        .name)
    slider_crank.stress_analysis = self.stress_analysis
    slider_crank.translate(self.moved.x, self.moved.y)
    return slider_crank


def translate(self, x:float, y:float):
    self.moved += Vector(x, y)


def rotate(self, angle:float):
    movement_ground = self.idisplacement + self.moved
    self.links[3].translate(-movement_ground.x, -
        movement_ground.y)
    self.links[3].rotate(angle)
    for i in range(3):
        self.links[i].rotate(angle)
    self.links[3].translate(movement_ground.x, movement_ground
        .y)
    self.rotation+=angle


def rotate_angle(self, angle:float):
    self.rotate(angle_rad(angle))


def output_rad(self, input_rad:float, coupler_rotation:float):
    """input_rad: crank angle relative to ground in rads
        coupler_rotation: rotation of the coupler in rads
        Returns a distance"""
    d = self.a*cos(input_rad)-self.b*cos(coupler_rotation)
    return d


def output_angle(self, input_angle:float, coupler_rotation:
    float):
    """input_rad: crank angle relative to ground in degrees
```

```
            coupler_rotation: rotation of the coupler in degrees
            Returns a distance"""
        d = self.output_rad(angle_rad(input_angle), angle_rad(
            coupler_rotation))
        return d


    def coupler_rad(self, input_rad:float):
        """input_rad: crank angle relative to ground in rads
        raise a ValueError if the crank can't be placed in that
            position"""
        try:
            solution_a = asin((self.a*sin(input_rad)-self.c)/self.
                b)
            solution_b = asin(-(self.a*sin(input_rad)-self.c)/self
                .b)+pi
        except ValueError:
            raise ValueError("Crank can't be put in that position"
                )

        return solution_a, solution_b


    def coupler_angle(self, input_angle:float):
        """input_angle: crank angle relative to ground in degrees
        raise a ValueError if the crank can't be placed in that
            position"""
        a, b = self.coupler_rad(angle_rad(input_angle))
        return rad_angle(a), rad_angle(b)


    def angular_velocity(self, input_rad1:float, input_rad2:float,
        dt:float, inversion:int):
        """input_rad1: crank angle relative to ground in rads
            input_rad2: second crank angle relative to ground in
                rads
            dt: time between the two positions.
            Ground and output have angular speeds of 0
        raise a ValueError if the crank can't be placed in that
            position"""
        assert inversion == 1 or inversion == 0, "Inversion can
            only by 0 or 1"
        w_crank = (input_rad2-input_rad1)/dt
        w_coupler = (self.coupler_rad(input_rad2)[inversion]-self.
            coupler_rad(input_rad1)[inversion])/dt
        return w_crank, w_coupler, 0, 0
```

```
def angular_velocity_angular_acceleration(self, input_rad:
    float, crank_angular_speed_rad:float,
    crank_angular_acceleration_rad:float, inversion:int):
    """Obtain tuples with angular speed and acceleration of
       each link
       For output it is linear velocity and acceleration since
           the slider has no rotation"""
    coupler_rad = self.coupler_rad(input_rad)[inversion]
    coupler_speed  = crank_angular_speed_rad*self.a/self.b*cos
       (input_rad)/cos(coupler_rad)
    output_speed = -self.a*crank_angular_speed_rad*sin(
       input_rad)+self.b*coupler_speed*sin(coupler_rad)

    coupler_acceleration = (self.a*
       crank_angular_acceleration_rad*cos(input_rad)-self.a*
       crank_angular_speed_rad**2*sin(input_rad)+self.b*
       coupler_speed**2*sin(coupler_rad))/(self.b*cos(
       coupler_rad))
    output_acceleration = -self.a*
       crank_angular_acceleration_rad*sin(input_rad)-self.a*
       crank_angular_speed_rad**2*cos(input_rad)+self.b*
       coupler_acceleration*sin(coupler_rad)+self.b*
       coupler_speed**2*cos(coupler_rad)
    ground_speed = 0
    ground_acceleration = 0
    return [(crank_angular_speed_rad,
       crank_angular_acceleration_rad), (coupler_speed,
       coupler_acceleration), (output_speed,
       output_acceleration), (ground_speed,
       ground_acceleration)]


def solution(self, angle_rad:float):
    """Absolute position given an input angle for crank in
       radians
       Return value might be: one solution repeated twice, two
           different solutions
       or ValueError if this position is not possible"""
    ca, cb = self.coupler_rad(angle_rad)
    oa = self.output_rad(angle_rad, ca)
    ob = self.output_rad(angle_rad, cb)

    crank = self.links[0].copy()
    coupler = self.links[1].copy()
    coupler2 = self.links[1].copy()
    output = self.links[2].copy()
    output2 = self.links[2].copy()
    ground = self.links[3].copy()
```

```
        crank.rotate(angle_rad)
        coupler.rotate(ca)
        coupler2.rotate(cb)

        crank.translate(self.origin.x, self.origin.y)
        crank.translate(self.moved.x, self.moved.y)
        ground.translate(self.moved.x, self.moved.y)
        mv_coupler = crank.connections[self.connections[0][1]] -
            coupler.connections[self.connections[1][0]]
        mv_coupler2 = crank.connections[self.connections[0][1]] -
            coupler2.connections[self.connections[1][0]]
        coupler.translate(mv_coupler.x, mv_coupler.y)
        coupler2.translate(mv_coupler2.x, mv_coupler2.y)
        mv_out = coupler.connections[self.connections[1][1]] -
            output.connections[self.connections[2][0]]
        mv_out2 = coupler2.connections[self.connections[1][1]] -
            output2.connections[self.connections[2][0]]
        output.translate(mv_out.x, mv_out.y)
        output2.translate(mv_out2.x, mv_out2.y)


        return [crank, coupler, output, ground], [crank, coupler2,
             output2, ground]



class Machine:
    def __init__(self, mechanisms:List[Mechanism], power_graph:
        List[List[int]], auto_adjust:bool=False, name:str=""):
        """A machine is defined here as one or more 4-bar
            mechanisms connected
            - First Mechanism must be the input mechanism
            - power_graph: indicates what a mechanism powers. It is
                a list of lists where list 0 represents the input
                crank from the first mechanism
              so mechanism '0' is represented in list as '1'.
                Example: [[1, 3], [], [], [2]] in this example the
                 input crank powers mechanism 1 and 3 and
                mechanism 3 powers mechanism 2
              if a mechanism doesn't power anything place an empty
                list. Bear in mind mechanism 2 will only be
                powered by the output of mechanism 3.
            - auto_adjust: indicates if mechanisms should be
                translated to the corresponding output
            - Important to consider that a mechanism cannot be
                powered by more than one output, if this happens
```

```
        only the last one will be taken as power
    - SliderCrank cannot power any mechanism
"""
assert all([type(mechanisms[i]) != SliderCrank or len(
    power_graph[i+1]) == 0 for i in range(len(mechanisms))
    ]), "Slider cannot power another mechanism"
assert 1 in power_graph[0], "Mechanism 1 must be powered
    by its own crank"

self.name = name
self.mechanisms = mechanisms[:]
self.power_graph = power_graph
self.input_graph = [-1 for i in range(len(power_graph))]
for power in range(len(power_graph)):
    for receiver in power_graph[power]:
        self.input_graph[receiver] = power

if -1 in self.input_graph[1:]:
    raise ValueError("One or more mechanisms are not
        powered")

if auto_adjust:
    for m in range(1, len(self.input_graph)):
        powered_mech_connection = self.mechanisms[m-1].
            connections[-1][0] #Ground connection of slave
            mechanism Index
        if self.input_graph[m]:
            powering_mech = self.input_graph[m]-1
            powering_mech_connection_to_attach = self.
                mechanisms[powering_mech].connections
                [-1][1] #Ground connection of commander
                mechanism Index
            displacement = self.mechanisms[powering_mech].
                links[-1].connections[
                powering_mech_connection_to_attach]-self.
                mechanisms[m-1].links[-1].connections[
                powered_mech_connection]
            displacement+= self.mechanisms[powering_mech].
                moved
        else:
            powering_mech_connection_to_attach = self.
                mechanisms[0].connections[0][0] #Ground
                connection of commander mechanism Index
            displacement = self.mechanisms[0].links[0].
                connections[
                powering_mech_connection_to_attach]-self.
                mechanisms[m-1].links[0].connections[
                powered_mech_connection]
```

```
                self.mechanisms[m-1].translate(displacement.x,
                    displacement.y)


    def copy(self):
        return Machine(mechanisms=[m.copy() for m in self.
            mechanisms], power_graph=self.power_graph, name=self.
            name)


    def angular_velocity(self, input_rad1:float, input_rad2:float,
        dt:float, pattern:list=0):
        """input_rad1: main crank angle relative to ground in rads
            input_rad2: second main crank angle relative to ground
                in rads
            dt: time between the two positions.
            pattern: indicates the inversions for each mechanism
            angular velocities are returned for each one of the
                mechanisms in the order of creation of the Machine
        raise a ValueError if the crank can't be placed in that
            position"""
        sorting = topological_sort(self.power_graph)[1:]
        snapshot_1 = self.solution(input_rad1, pattern)
        snapshot_2 = self.solution(input_rad2, pattern)
        velocities = [0 for i in range(len(self.mechanisms))]
        for i in range(len(snapshot_1)):
            velocities_single_mechanism = [(snapshot_2[i][x].
                rotation-snapshot_1[i][x].rotation)/dt for x in
                range(len(snapshot_1[i])) ]
            velocities[sorting[i]-1] = velocities_single_mechanism
        return velocities


    def solution(self, angle_rad:float, pattern:list=0)->List[List
        [Link]]:
        inversions = None
        solutions = [0 for i in range(len(self.mechanisms)+1)]
        if type(pattern) == list:
            if len(pattern) != len(self.mechanisms):
                raise ValueError("Incorrect inversion, must be a
                    single 1 or 0 or a list of 1s and 0s for each
                    mechanism")
            inversions = pattern
        elif pattern:
            inversions = [1 for i in range(len(self.mechanisms))]
        else:
            inversions = [0 for i in range(len(self.mechanisms))]
```

```
        snapshot = []

        sorting = topological_sort(self.power_graph)
        solutions[0] = self.mechanisms[0].rotation+angle_rad
        counter = 1
        for mechanism_ in sorting[1:]:
            n_solution = self.mechanisms[mechanism_-1].solution(
                solutions[self.input_graph[mechanism_]]-self.
                mechanisms[mechanism_-1].rotation)[inversions[
                mechanism_-1]]
            snapshot.append(n_solution)
            if self.power_graph[mechanism_]:
                solutions[mechanism_] = self.mechanisms[mechanism_
                    -1].output_rad(solutions[self.input_graph[
                    mechanism_]]-self.mechanisms[mechanism_-1].
                    rotation)[inversions[mechanism_-1]]+self.
                    mechanisms[mechanism_-1].rotation

        return snapshot


    def solution_kinetics(self, angle_rad:float, speed_rad:float,
        acceleration_rad:float, external_moments_cranks:List[float
        ]=[], external_moments_couplers:List[float]=[], pattern:
        list=0):
        assert all([m.stress_analysis for m in self.mechanisms]),
            "All mechanisms must have a centroid and mass to find
            the forces"
        assert len(external_moments_cranks) == len(self.mechanisms
            ) or len(external_moments_cranks) == 1 or len(
            external_moments_cranks) == 0, "Must provide an
            external moment for each mechanism or just for the last
            one"
        assert len(external_moments_couplers) == len(self.
            mechanisms) or len(external_moments_couplers) == 1 or
            len(external_moments_couplers) == 0, "Must provide an
            external moment for each mechanism or just for the last
            one"
        assert all([type(m) == Mechanism for m in self.mechanisms
            ]), "Stresses have only been implemented to four link-
            pin mechanisms. No SliderCrank, nor any other type"

        if len(external_moments_couplers) == 0 and len(
            external_moments_cranks) == 0:
            raise ValueError("No external torque is being applied"
                )

        inversions = None
```

```
solutions = [0 for i in range(len(self.mechanisms)+1)]
accelerations = [[] for i in range(len(self.mechanisms)+1)
    ]
if type(pattern) == list:
    if len(pattern) != len(self.mechanisms):
        raise ValueError("Incorrect inversion, must be a
            single 1 or 0 or a list of 1s and 0s for each
            mechanism")
    inversions = pattern
elif pattern:
    inversions = [1 for i in range(len(self.mechanisms))]
else:
    inversions = [0 for i in range(len(self.mechanisms))]

snapshot = []
sorting = topological_sort(self.power_graph)
solutions[0] = self.mechanisms[0].rotation+angle_rad
accelerations[0] = ((speed_rad, acceleration_rad), (
    speed_rad, acceleration_rad), (speed_rad,
    acceleration_rad), (speed_rad, acceleration_rad))
linear_and_angular_accelerations = []

# Saved because external moment is applied to last
    mechanism's output
# So its reaction is applied to output of the mechanism
    that dirves it
solution_forces_ = [[],]
solution_accelerations_ = [[],]
final_forces_ = []
#
#Important to save rotations of links so that stresses can
    be determined
absolute_rotations = []
#
for mechanism_ in sorting[1:]:
    n_solution = self.mechanisms[mechanism_-1].solution(
        solutions[self.input_graph[mechanism_]]-self.
        mechanisms[mechanism_-1].rotation)[inversions[
        mechanism_-1]]
    n_acceleration = self.mechanisms[mechanism_-1].
        angular_velocity_angular_acceleration(solutions[
        self.input_graph[mechanism_]]-self.mechanisms[
        mechanism_-1].rotation,\
```

```
accelerations[mechanism_] = n_acceleration
absolute_rotations.append((solutions[self.input_graph[
    mechanism_]]+self.mechanisms[mechanism_-1].rotation
    ,\
                                        self.mechanisms[
                                            mechanism_-1].
                                            coupler_rad(
                                            solutions[self.
                                            input_graph[
                                            mechanism_]]-self.
                                            mechanisms[
                                            mechanism_-1].
                                            rotation)[
                                            inversions[
                                            mechanism_-1]]+self
                                            .mechanisms[
                                            mechanism_-1].
                                            rotation,\
                                        self.mechanisms[
                                            mechanism_-1].
                                            output_rad(
                                            solutions[self.
                                            input_graph[
                                            mechanism_]]-self.
                                            mechanisms[
                                            mechanism_-1].
                                            rotation)[
                                            inversions[
                                            mechanism_-1]]+self
                                            .mechanisms[
```

```
                                                mechanism_-1].
                                                rotation))
        snapshot.append(n_solution)
        if self.power_graph[mechanism_]:
            solutions[mechanism_] = self.mechanisms[mechanism_
                -1].output_rad(solutions[self.input_graph[
                mechanism_]]-self.mechanisms[mechanism_-1].
                rotation)[inversions[mechanism_-1]]+self.
                mechanisms[mechanism_-1].rotation


        #Linear accelerations
        mechanism_analyzed = self.mechanisms[mechanism_-1]
        #Crank linear accelerations
        radius_connection_to_centroid = n_solution[0].
            connections[mechanism_analyzed.connections[0][0]]-
            n_solution[0].centroid_vector
        w2 = n_acceleration[0][0]*n_acceleration[0][0]
        ax = radius_connection_to_centroid.x*w2
        ay = radius_connection_to_centroid.y*w2

        if n_acceleration[0][1]>0:
            radius_connection_to_centroid.rotate(-pi/2)
        else:
            radius_connection_to_centroid.rotate(pi/2)
        ax+=radius_connection_to_centroid.x*abs(n_acceleration
            [0][1])
        ay+=radius_connection_to_centroid.y*abs(n_acceleration
            [0][1])


        #Coupler and output linear accelerations Mechanism
        if type(mechanism_analyzed) == Mechanism:
            radius_connection_to_centroid = n_solution[0].
                connections[mechanism_analyzed.connections
                [0][1]]-n_solution[1].centroid_vector
            w2_cop = n_acceleration[1][0]*n_acceleration[1][0]
            ax_cop = radius_connection_to_centroid.x*w2_cop
            ay_cop = radius_connection_to_centroid.y*w2_cop

            if n_acceleration[1][1]>0:
                radius_connection_to_centroid.rotate(-pi/2)
            else:
                radius_connection_to_centroid.rotate(pi/2)
            ax_cop+=radius_connection_to_centroid.x*abs(
                n_acceleration[1][1])
            ay_cop+=radius_connection_to_centroid.y*abs(
                n_acceleration[1][1])
```

```
        # Translate relative velocity into absolute
        # The relative velocity is in terms of the
            connection point from crank not from the
            centroid, so acceleration for this point needs
            to be computed
        radius_connection_to_centroid = n_solution[0].
            connections[mechanism_analyzed.connections
            [0][0]] - n_solution[0].connections[
            mechanism_analyzed.connections[0][1]]
        ax_conn = radius_connection_to_centroid.x*w2
        ay_conn = radius_connection_to_centroid.y*w2

        if n_acceleration[0][1]>0:
            radius_connection_to_centroid.rotate(-pi/2)
        else:
            radius_connection_to_centroid.rotate(pi/2)
        ax_conn+=radius_connection_to_centroid.x*abs(
            n_acceleration[0][1])
        ay_conn+=radius_connection_to_centroid.y*abs(
            n_acceleration[0][1])

        ax_cop+=ax_conn
        ay_cop+=ay_conn

        #Output
        radius_connection_to_centroid = n_solution[3].
            connections[mechanism_analyzed.connections
            [3][1]] - n_solution[2].centroid_vector
        w2_out = n_acceleration[2][0]*n_acceleration[2][0]
        ax_out = radius_connection_to_centroid.x*w2_out
        ay_out = radius_connection_to_centroid.y*w2_out

        if n_acceleration[2][1]>0:
            radius_connection_to_centroid.rotate(-pi/2)
        else:
            radius_connection_to_centroid.rotate(pi/2)
        ax_out+=radius_connection_to_centroid.x*abs(
            n_acceleration[2][1])
        ay_out+=radius_connection_to_centroid.y*abs(
            n_acceleration[2][1])



    #Coupler and output linear accelerations SliderCrank
    elif type(mechanism_analyzed) == SliderCrank:
        radius_connection_to_centroid = n_solution[1].
            connections[mechanism_analyzed.connections
```

```
        [1][1]] - n_solution[1].centroid_vector
    w2_cop = n_acceleration[1][0]*n_acceleration[1][0]
    ax_cop = radius_connection_to_centroid*w2_cop
    ay_cop = radius_connection_to_centroid*w2_cop

    if n_acceleration[1][1]>0:
        radius_connection_to_centroid.rotate(-pi/2)
    else:
        radius_connection_to_centroid.rotate(pi/2)
    ax_cop+=radius_connection_to_centroid.x*abs(
        n_acceleration[1][1])
    ay_cop+=radius_connection_to_centroid.y*abs(
        n_acceleration[1][1])

    # Translate relative velocity into absolute
    # Use as reference velocity from output since
        slider has to be naturally computed and
        corresponds to the reference point for the
        SliderCrank coupler
    direction = n_solution[3].connections[
        mechanism_analyzed.connections[3][1]] -
        n_solution[3].connections[mechanism_analyzed.
        connections[3][0]]
    normalize_x = direction.x/((direction.x**2+
        direction.y**2)**0.5)
    normalize_y = direction.y/((direction.x**2+
        direction.y**2)**0.5)

    ax_out = normalize_x*n_acceleration[2][1]
    ay_out = normalize_y*n_acceleration[2][1]
    ax_cop+=ax_out
    ay_cop+=ay_out

else:
    raise ValueError(f"type '{type(mechanism_analyzed)
        }' has not defined linear accelerations")

#ax, ay and angular acceleration for each link in each
    mechanism
linear_and_angular_accelerations.append([[ax, ay,
    n_acceleration[0][1]], [ax_cop, ay_cop,
    n_acceleration[1][1]], [ax_out, ay_out,
    n_acceleration[2][1]], [0, 0, 0]])


force_matrix = np.array([\
                        [1, 0, 1, 0, 0, 0, 0, 0,
                            0],\
```

```
    [0, 1, 0, 1, 0, 0, 0, 0,
        0],\
    [-(n_solution[0].connections
        [mechanism_analyzed.
        connections[0][0]] -
        n_solution[0].
        centroid_vector).y, (
        n_solution[0].connections
        [mechanism_analyzed.
        connections[0][0]] -
        n_solution[0].
        centroid_vector).x, -(
        n_solution[0].connections
        [mechanism_analyzed.
        connections[0][1]] -
        n_solution[0].
        centroid_vector).y, (
        n_solution[0].connections
        [mechanism_analyzed.
        connections[0][1]] -
        n_solution[0].
        centroid_vector).x, 0, 0,
         0, 0, 1],\
    [0, 0, -1, 0, 1, 0, 0, 0,
        0],\
    [0, 0, 0, -1, 0, 1, 0, 0,
        0],\
    [0, 0, (n_solution[0].
        connections[
        mechanism_analyzed.
        connections[0][1]] -
        n_solution[1].
        centroid_vector).y, -(
        n_solution[0].connections
        [mechanism_analyzed.
        connections[0][1]] -
        n_solution[1].
        centroid_vector).x, -(
        n_solution[1].connections
        [mechanism_analyzed.
        connections[1][1]] -
        n_solution[1].
        centroid_vector).y, (
        n_solution[1].connections
        [mechanism_analyzed.
        connections[1][1]] -
        n_solution[1].
        centroid_vector).x, 0, 0,
```

```
                    0],\
                [0, 0, 0, 0, -1, 0, 1, 0,
                    0],\
                [0, 0, 0, 0, 0, -1, 0, 1,
                    0],\
                [0, 0, 0, 0, (n_solution[1].
                    connections[
                    mechanism_analyzed.
                    connections[1][1]] -
                    n_solution[2].
                    centroid_vector).y, -(
                    n_solution[1].connections
                    [mechanism_analyzed.
                    connections[1][1]] -
                    n_solution[2].
                    centroid_vector).x, -(
                    n_solution[2].connections
                    [mechanism_analyzed.
                    connections[2][1]] -
                    n_solution[2].
                    centroid_vector).y, (
                    n_solution[2].connections
                    [mechanism_analyzed.
                    connections[2][1]] -
                    n_solution[2].
                    centroid_vector).x, 0]\
            ])

mass_x_accelerations = np.array([\
                    [n_solution[0].mass*
                        ax],\
                    [n_solution[0].mass*
                        ay],\
                    [n_solution[0].
                        moment_inertia_centroid
                        *n_acceleration
                        [0][1]],\
                    [n_solution[1].mass*
                        ax_cop],\
                    [n_solution[1].mass*
                        ay_cop],\
                    [n_solution[1].
                        moment_inertia_centroid
                        *n_acceleration
                        [1][1]],\
                    [n_solution[2].mass*
                        ax_out],\
                    [n_solution[2].mass*
```

```
                                                    ay_out],\
                                                [n_solution[2].
                                                    moment_inertia_centroid
                                                    *n_acceleration
                                                    [2][1]]\
                                               ])

        solution_accelerations_.append(mass_x_accelerations)
        solution_forces_.append(force_matrix)
        #print(linalg.solve(force_matrix, mass_x_accelerations
            ))

    external_moments_crank = []
    external_moments_coupler = []

    for i in range(len(self.mechanisms)):
            external_moments_crank.append(0)

    if len(external_moments_cranks) == 1:
        external_moments_crank[-1] = external_moments_cranks
            [0]
    elif len(external_moments_cranks) > 1:
        external_moments_crank = external_moments_cranks


    for i in range(len(self.mechanisms)):
            external_moments_coupler.append(0)

    if len(external_moments_couplers) == 1:
        external_moments_coupler[-1] =
            external_moments_couplers[0]
    elif len(external_moments_couplers) > 1:
        external_moments_coupler = external_moments_couplers



    for matrix in sorting[::-1][:-1]:
        solution_accelerations_[matrix][-1][0] -=
            external_moments_crank[matrix-1]
        solution_accelerations_[matrix][-4][0] -=
            external_moments_coupler[matrix-1]
        solution = linalg.solve(solution_forces_[matrix],
            solution_accelerations_[matrix])
        final_forces_.append(solution)
        if self.input_graph[matrix] != 0:
            solution_accelerations_[self.input_graph[matrix
                ]][-1][0]-=solution[-1][0]
```

```
final_forces_ = final_forces_[::-1]
final_stresses_ = []
vonMises_ = []
locations_ = []
#Stresses only for the crank of the first mechanism, all
    others are powered by the output
include_crank = 0
current = 0
for mechanism in snapshot:
    link_ = 0+include_crank
    stresses = []
    location = None
    for link in mechanism[include_crank:-1]:
        max_vonMises = 0
        for i in range(len(link.element_node_locations)):
            if link.areas[i]<=1e-4:
                continue
            forcex = final_forces_[current][link_*2+2][0]
            forcey = final_forces_[current][link_*2+3][0]
            move_force = link.element_node_locations[i]-
                self.mechanisms[current].links[link_].
                connections[self.mechanisms[current].
                connections[link_][0]]

            moment = forcex*move_force.y-forcey*move_force
                .x+final_forces_[current][-1][0]
            c = cos(absolute_rotations[current][link_])
            s = sin(absolute_rotations[current][link_])
            local_x = forcex*c+forcey*s
            local_y = -forcex*s+forcey*c

            moment_stress = moment*link.heights[i]/link.
                inertias[i]
            shear_stress = local_y/link.areas[i]
            normal_stress = local_x/link.areas[i]
            total_normal_stress = abs(normal_stress)+abs(
                moment_stress)
            vonMises = (total_normal_stress*
                total_normal_stress+3*shear_stress*
                shear_stress)**0.5
            if vonMises>max_vonMises:
                max_vonMises = vonMises
                stresses = [shear_stress, normal_stress,
                    moment_stress]
                if (moment_stress > 0 and normal_stress >
                    0) or (moment_stress < 0 and
                    normal_stress < 0):
                    location = Vector(s*link.heights[i], -
```

```
                                    c*link.heights[i])+link.
                                    element_node_locations[i]
                            else:
                                location = Vector(-s*link.heights[i],
                                    c*link.heights[i])+link.
                                    element_node_locations[i]
                        link_+=1
                        final_stresses_.append(stresses)
                        vonMises_.append(max_vonMises)
                        locations_.append(location)
                    include_crank = 1
                    current+=1
            return linear_and_angular_accelerations, final_forces_,
                final_stresses_, vonMises_, locations_, snapshot,
                sorting



def topological_sort(mechanisms:List[List[int]]):
    found = []
    def topo_sort(mechanisms:List[List[int]], node_start:int=0):
        """Find the dependencies of mechanisms in a machine"""
        if node_start in found:
            return
        found.append(node_start)
        for node in mechanisms[node_start]:
            topo_sort(mechanisms, node)
        for nnode in range(len(mechanisms)):
            topo_sort(mechanisms, nnode)
    topo_sort(mechanisms)
    return found



if __name__ == "__main__":
    a, b = Vector(1, 0), Vector(0, 1)
    print(a)
    a.rotate_angle(180)
    print(a+b)
    print(topological_sort([[1, 2, 3], [4], [5], [], [], []]))
```

12.4. Interfaz gráfica

12.4.1. Animación

```
#Fernando Jose Lavarreda Urizar
#Program to analyze Mechanisms, Graduation Project UVG
```

```
from time import sleep
from typing import List
import matplotlib.pyplot as plt
from matplotlib.axes import Axes
from matplotlib.figure import Figure
from matplotlib.animation import FuncAnimation


if __name__ == "__main__":
    import os
    import sys
    parent = os.path.abspath(os.path.join(os.path.dirname(__file__
        ), ".."))
    sys.path.append(parent)
    from examples import examples

from mechanisms import geometry as gm


def plot_link(link:gm.Link, axes:Axes=None, show_connections:bool=
    False, grid:bool=False, resize:bool=False, color=""):
    if axes:
        ax = axes
    else:
        fig = plt.figure()
        ax = fig.add_subplot(111)

    if color:
        for curve in link.curves:
            ax.plot([vector.x for vector in curve.vectors], [
                vector.y for vector in curve.vectors], color=color)
    else:
        for curve in link.curves:
            ax.plot([vector.x for vector in curve.vectors], [
                vector.y for vector in curve.vectors])

    if show_connections:
        ax.scatter([vector.x for vector in link.connections], [
            vector.y for vector in link.connections], color="red")

    if resize:
        miny, maxy = ax.get_ylim()
        minx, maxx = ax.get_xlim()

        minimun = min([miny, minx])
        maximum = max([maxy, maxx])
```

```
        ax.set_xlim([minimun, maximum])
        ax.set_ylim([minimun, maximum])
    ax.grid(visible=grid)

    if not grid:
        ax.set_xticklabels([])
        ax.set_xticks([])
        ax.set_yticks([])
        ax.set_yticklabels([])

    if not axes:
        plt.show()



def plot_mechanism(crank:gm.Link, coupler:gm.Link, output:gm.Link,
    ground:gm.Link, axes:Axes=None, colors:List[str]=["red", "
   purple", "green", "orange", "blue", "black", "yellow"]):
    links = [crank, coupler, output, ground]

    if axes:
        ax = axes
    else:
        fig = plt.figure()
        ax = fig.add_subplot(111)

    color=0
    for link in links:
        for curve in link.curves:
            ax.plot([vector.x for vector in curve.vectors], [
                vector.y for vector in curve.vectors], color=colors
                [color%len(colors)])
        color+=1
    miny, maxy = ax.get_ylim()
    minx, maxx = ax.get_xlim()

    minimun = min([miny, minx])
    maximum = max([maxy, maxx])

    ax.set_xlim([minimun, maximum])
    ax.set_ylim([minimun, maximum])

    if not axes:
        plt.show()


def plot_machine(mechanisms:List[List[gm.Link]], axes:Axes=None,
   colors:List[str]=["red", "purple", "green", "orange", "blue", "
```

```
    black", "yellow"], mass_center:bool=False, max_stress:List[gm.
    Vector]=[]):
     if mass_center:
         for jindex, m in enumerate(mechanisms):
             for index, l in enumerate(m):
                 assert l.centroid_vector, f"Missing centroid for
                     link {index} in mechanism {jindex}"

     if axes:
         ax = axes
     else:
         fig = plt.figure()
         ax = fig.add_subplot(111)

     color = 0
     omit_crank = 0
     for mechanism in mechanisms:
         for link in mechanism[omit_crank:]:
             for curve in link.curves:
                 ax.plot([vector.x for vector in curve.vectors], [
                     vector.y for vector in curve.vectors], color=
                     colors[color%len(colors)])
             color+=1
             if mass_center:
                 ax.scatter([link.centroid_vector.x,], [link.
                     centroid_vector.y], color='red')
             if max_stress:
                 ax.scatter([v.x for v in max_stress], [v.y for v
                     in max_stress], color="lime")
         omit_crank= 1
     miny, maxy = ax.get_ylim()
     minx, maxx = ax.get_xlim()

     minimun = min([miny, minx])
     maximum = max([maxy, maxx])

     ax.set_xlim([minimun, maximum])
     ax.set_ylim([minimun, maximum])

     if not axes:
         plt.show()


def plot_rotation_mech(mechanism:gm.Mechanism, frames:int,
    inversion:int=0, colors:List[str]=["red", "purple", "green", "
    orange", "blue", "black", "yellow"], axes:Axes=None, fig:Figure
    =None):
     assert (axes == None and fig == None) or (axes != None and fig
```

```
        != None), "Assign both axes and figure or none"
    if axes:
        ax = axes
    else:
        fig = plt.figure()
        ax = fig.add_subplot(111)
    s1 = mechanism.solution(0)[inversion]

    ax.set_xlim([-mechanism.size+mechanism.location().x, mechanism
        .size+mechanism.location().x])
    ax.set_ylim([-mechanism.size+mechanism.location().y, mechanism
        .size+mechanism.location().y])

    lines = []
    color = 0
    for link in s1:
        lines.append([])
        for curve in link.curves:
            lines[-1].append(*ax.plot([vector.x for vector in
                curve.vectors], [vector.y for vector in curve.
                vectors], color=colors[color%len(colors)]))
        color+=1

    def animate(i):
        radian = 2*gm.pi*i/frames
        solution = mechanism.solution(radian)[inversion]
        for link in range(4):
            counter = 0
            for curve in solution[link].curves:
                lines[link][counter].set_data([v.x for v in curve.
                    vectors], [v.y for v in curve.vectors])
                counter+=1
        ret = []
        for ll in lines:
            for cc in ll:
                ret.append(cc)
        return ret

    anim = FuncAnimation(fig, animate, frames=frames, interval=20,
        blit=True)

    if not axes:
        plt.show()
    else:
        return anim


def plot_rotation_mach(machine:gm.Machine, frames:int, inversion:
```

```
int=0, lims=[[-1.5, 7], [-2.5, 4]], colors:List[str]=["red", "
purple", "green", "orange", "blue", "black", "yellow"],\
                    axes:Axes=None, fig:Figure=None,
                        animation_limits=(0, gm.pi*2), invert:
                        bool=False, mass_center:bool=False, save
                        =""):
    """
    Inversion can be either 0 for 0s list a 1 for 1s list or a
        list indicating the inversion for each mechanism
    """
    assert (axes == None and fig == None) or (axes != None and fig
        != None), "Assign both axes and figure or none"
    if axes:
        ax = axes
    else:
        fig = plt.figure()
        ax = fig.add_subplot(111)

    starter_sol = machine.solution(0, inversion)

    ax.set_xlim(lims[0])
    ax.set_ylim(lims[1])


    #Important to skip crank for all mechanisms except input one
    lines = []
    current_mech = 0
    current_link = 0
    color = 0
    centroids = []
    for mechanism_ in starter_sol:
        lines.append([])
        current_link = 0
        if starter_sol[current_mech][0].centroid_vector and
            mass_center:
            if current_mech == 0:
                inputs_ = 0
            else:
                inputs_ = 1
            centroids.append(ax.scatter([link.centroid_vector.x
                for link in starter_sol[current_mech][inputs_:]], [
                link.centroid_vector.y for link in starter_sol[
                current_mech][inputs_:]], color="red"))
        else:
            centroids.append(None)
        for link_ in mechanism_:
            lines[-1].append([])
            if not(current_mech != 0 and current_link == 0):
```

```
            for curve_ in link_.curves:
                lines[-1][-1].append(*ax.plot([vector.x for
                    vector in curve_.vectors], [vector.y for
                    vector in curve_.vectors], color=colors[
                    color%len(colors)]))
        current_link+=1
        color+=1
    current_mech+=1


def animate(i):
    if invert:
        if i >= frames/2:
            radian = animation_limits[1]-(animation_limits[1]-
                animation_limits[0])*(i/frames-0.5)*2
        else:
            radian = (animation_limits[1]-animation_limits[0])
                *2*i/frames+animation_limits[0]
    else:
        radian = (animation_limits[1]-animation_limits[0])*i/
            frames+animation_limits[0]
    solution = machine.solution(radian, inversion)
    for mechanism in range(len(lines)):
        for link in range(len(lines[mechanism])):
            counter = 0
            if mechanism != 0 and link == 0:
                continue
            for curve in solution[mechanism][link].curves:
                lines[mechanism][link][counter].set_data([v.x
                    for v in curve.vectors], [v.y for v in
                    curve.vectors])
                counter+=1

    ret = []
    for mech in lines:
        for link in mech:
            for curve in link:
                ret.append(curve)

    for mechanism__ in range(len(solution)):
        if centroids[mechanism__]:
            if mechanism__ == 0:
                inputs_ = 0
            else:
                inputs_ = 1
            centroids[mechanism__].set_offsets([(link.
                centroid_vector.x, link.centroid_vector.y) for
                link in solution[mechanism__][inputs_:]])
```

```
                ret.append(centroids[mechanism__])
        return ret

    anim = FuncAnimation(fig, animate, frames=frames, interval=20,
        blit=True)

    if save:
        anim.save(save)

    if not axes:
        plt.show()
    else:
        return anim



if __name__ == "__main__":

    if '1' in sys.argv:
        machine = examples.build_machine()
        plot_rotation_mach(machine, frames=100, inversion=1,
            mass_center=True)

    if '2' in sys.argv:
        compresor = examples.build_compresor(5)
        plot_rotation_mach(compresor, frames=100, inversion=1,
            lims=[[-17, 17], [-17, 17]], save="")

    if '3' in sys.argv:
        powered = examples.build_double_crank(5)
        plot_rotation_mach(powered, frames=200, inversion=[0, 1,
            1, 1, 1, 1], lims=[[-12, 24], [-17, 17]], save="")


    if '4' in sys.argv:
        machine = examples.build_machine()
        plot_machine(machine.solution(1.8*gm.pi, pattern=1),
            mass_center=True)
```

12.4.2. Interacción con el usuario

```
import tkinter as tk
from typing import Tuple, List
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
from matplotlib.figure import Figure
```

```
class Graphics():

    def __init__(self, parent, size:Tuple[int, int], row, column,
       columnspan, rowspan, dpi:int, title="", remove_ticks=True):
        self.fig = Figure(figsize=size, dpi=dpi)
        self.canvas = FigureCanvasTkAgg(self.fig, master=parent)
        self.canvas.draw()
        self.canvas.get_tk_widget().grid(column=column, row=row,
           rowspan=rowspan, columnspan=columnspan, sticky=tk.SE+tk
           .NW)
        self.axis = self.fig.add_subplot(111)
        self.axis.set_title(title)
        self.fig.tight_layout()
        if remove_ticks:
            self.axis.set_xticklabels([])
            self.axis.set_xticks([])
            self.axis.set_yticks([])
            self.axis.set_yticklabels([])


    def render(self):
        self.canvas.draw()
        self.canvas.flush_events()


    def clear(self):
        self.axis.cla()
        self.render()


    def set_lims(self, xlims:Tuple[float, float], ylims:Tuple[
       float, float]):
        self.axis.axis(xmin=xlims[0], xmax=xlims[1], ymin=ylims
           [0], ymax=ylims[1])
        self.render()


    def make_animation(self, sequence:Tuple[List[List[float]],
       List[List[float]]], *, time_out:float=0):
        line = self.axis.plot(sequence[0][0], sequence[1][0])[0]
        if len(str(self.axis.get_yticks()[0]))>4:
            self.axis.set_yticklabels(self.axis.get_yticks(),
               rotation=70)
        self.render()
        for frame in range(len(sequence[0])):
            line.set_data(sequence[0][frame], sequence[1][frame])
            time.sleep(time_out)
```

```
            self.render()


    def static_drawing(self, sequence:Tuple[List[float], List[
        float]], grid=False):
        self.axis.plot(sequence[0], sequence[1])
        #if len(str(self.axis.get_yticks()[0]))>4:
        #    self.axis.set_yticklabels(self.axis.get_yticks(),
            rotation=70)
        if not grid:
            self.axis.set_xticklabels([])
            self.axis.set_xticks([])
            self.axis.set_yticks([])
            self.axis.set_yticklabels([])
        else:
            self.axis.grid(visible=grid)
        self.render()


    def moving(self):
        self.set_lims([-1.2*4/3, 1.2*4/3], [-1.2, 1.2])
        xs = [[0, math.cos(i/1_000*2*math.pi)] for i in range(1
            _000)]
        ys = [[0, math.sin(i/1_000*2*math.pi)] for i in range(1
            _000)]
        self.make_animation([xs, ys], time_out=0)

#Fernando Jose Lavarreda Urizar
#Program to analyze Mechanisms, Graduation Project UVG


import re
import csv
import tkinter as tk
import tkinter.ttk as ttk
from graphics import graphics
from .display import Graphics
from typing import Tuple, List
import tkinter.filedialog as fd
import tkinter.messagebox as msg
from math import sin, cos, tan, pi
from tkinter.simpledialog import askfloat, askstring
from mechanisms.geometry import Function, Curve, Vector, Link,
    Mechanism, SliderCrank, Machine


class UICurve(ttk.Frame):
    def __init__(self, master, curves:List[Curve]):
```

```
super().__init__(master)
self.curves = curves
self.graphics = Graphics(self, (9.6, 6), row=0, column=0,
    columnspan=1, rowspan=10, dpi=100, title="")
self.cursor = -1
self.temp = None
#----------Control of Curve Properties----------
#Variables
self.smovex = tk.StringVar(self)
self.smovey = tk.StringVar(self)
self.srotate = tk.StringVar(self)
self.sname = tk.StringVar(self)
self.sfunctions = tk.StringVar(self)
#-----
self.controls = tk.LabelFrame(self, text="Controls")
self.controls.grid(row=10, column=0, rowspan=1, sticky=tk.
    SE+tk.NW, pady=(0, 2), padx=(2, 2))
self.name = ttk.Entry(self.controls, textvariable=self.
    sname)
self.functions = ttk.Entry(self.controls, textvariable=
    self.sfunctions)
self.movex = ttk.Entry(self.controls, width=10,
    textvariable=self.smovex)
self.movey = ttk.Entry(self.controls, width=10,
    textvariable=self.smovey)
self.rotate_angle = ttk.Entry(self.controls, width=5,
    textvariable=self.srotate)
ttk.Label(self.controls, text="Name:").grid(row=0, column
    =0, sticky=tk.SE+tk.NW)
self.name.grid(row=1, column=0, sticky=tk.SE+tk.NW,
    columnspan=2)
ttk.Label(self.controls, text="Functions:").grid(row=2,
    column=0, sticky=tk.SE+tk.NW)
self.functions.grid(row=3, column=0, sticky=tk.SE+tk.NW,
    columnspan=2)
ttk.Button(self.controls, text="Upload file", command=self
    .upload).grid(row=1, column=9, sticky=tk.SE+tk.N,
    columnspan=1, padx=(0,0))
ttk.Button(self.controls, text="clear", command=self.clear
    ).grid(row=3, column=2, sticky=tk.SE+tk.N, columnspan
    =1)
ttk.Label(self.controls, text="Translate:").grid(row=0,
    column=4, sticky=tk.SE+tk.NW, columnspan=2, padx=(15,0)
    )
ttk.Label(self.controls, text="x:").grid(row=1, column=4,
    sticky=tk.SE+tk.NW, columnspan=1, padx=(15, 0))
ttk.Label(self.controls, text="y:").grid(row=2, column=4,
    sticky=tk.SE+tk.NW, columnspan=1, padx=(15, 0))
```

```
        self.movex.grid(row=1, column=5, sticky=tk.SE+tk.NW,
            columnspan=1)
        self.movey.grid(row=2, column=5, sticky=tk.SE+tk.NW,
            columnspan=1)
        ttk.Button(self.controls, text="move", command=self.move).
            grid(row=3, column=4, sticky=tk.SE+tk.NW, columnspan=2,
             padx=(15, 0))
        ttk.Label(self.controls, text="Rotate:").grid(row=0,
            column=6, sticky=tk.SE+tk.NW, columnspan=2, padx=(15,0)
            )
        ttk.Label(self.controls, text="Angle(   )").grid(row=1,
            column=6, sticky=tk.SE+tk.NW, columnspan=1, padx=(15,0)
            )
        self.rotate_angle.grid(row=1, column=7, sticky=tk.SE+tk.NW
            , columnspan=1)
        ttk.Button(self.controls, text="rotate", command=self.
            rotate).grid(row=3, column=6, sticky=tk.SE+tk.NW,
            columnspan=2, padx=(15, 0))
        ttk.Button(self.controls, text="save", command=self.save).
            grid(row=2, column=8, sticky=tk.SE+tk.NW, columnspan=2,
             padx=(25, 0))
        ttk.Button(self.controls, text="new", command=self.new).
            grid(row=1, column=8, sticky=tk.SE+tk.NW, columnspan=1,
             padx=(25, 0))
        ttk.Button(self.controls, text="delete", command=self.
            delete).grid(row=3, column=8, sticky=tk.SE+tk.NW,
            columnspan=2, padx=(25, 0))
        self.select = ttk.Combobox(self.controls, state="readonly"
            )
        self.select.bind("<<ComboboxSelected>>", self.selection)
        self.select.grid(row=0, column=8, sticky=tk.SE+tk.NW,
            columnspan=2, padx=(25, 0))
        self.functions.bind("<Return>", self.func)
        if self.curves:
            self.select["values"] = [c.name for c in curves]
        #-----------------------------


def func(self, *args):
    m = re.match("[\d|\.|(sin)|(cos)|(tan)|x|\*|\+|\-|/|\(|\)
        ]*\{(-?\d+\.?\d*,\s*){2}(\d+\s*){1}\}", self.sfunctions
        .get())
    vectors = []
    if m != None:
        evaluate = m.group()
        limits = re.findall("-?\d+\.?\d*", evaluate)[-3:]
        evaluate = evaluate[:evaluate.find('{')]
        start = float(limits[0])
```

```
            end = float(limits[1])
            samples = int(limits[2])
            save_start = start
            if start>=end:
                msg.showerror(parent=self, title="Error", message=
                    "End can't be smaller than beginning \n(limits
                    of the function)")
                return
            try:
                dx = (end-start)/samples
                for i in range(samples):
                    result = eval(evaluate.replace("x", "("+str(
                        start)+")"))
                    vectors.append(Vector(start, result))
                    start+=dx
                vectors.append(Vector(end, eval(evaluate.replace("
                    x", "("+str(end)+")"))))
            except Exception as e:
                msg.showerror(parent=self, title="Error", message=
                    "Couldn't evaluate expression")
            else:
                self.temp.vectors = list(self.temp.vectors)+
                    vectors
                self.temp.functions = self.sfunctions.get()
                self.temp.name = self.sname.get()
                self.temp.function = Function(start=save_start,
                    end=end, string_function=evaluate)
                self.load_curve(self.temp)
        else:
            msg.showerror(parent=self, title="Error", message="
                Couldn't interpret expression")


    def upload(self, *args):
        rd = fd.askopenfilename(parent=self, title="Load Data",
            initialdir="C:\\Documents", filetypes=(("CSV", "*.csv")
            ,("CSV", "*.txt")))
        if rd:
            response = askstring(title="Load", prompt="Load as one
                curve? (y/n)")
            multiple = True
            if response and response.lower() == "y":
                multiple = False
            file = open(rd)
            try:
                cc = Curve.build_from_io(file, multiple=multiple)
            except Exception:
                msg.showerror(parent=self, title="Error", message=
```

```
                "Could not process selected file")
        else:
            name = rd[-rd[::-1].find("/"):-4]
            if type(cc) == list:
                counter = 1
                for c in cc:
                    c.name = name + f"{counter}"
                    counter+=1
                    self.curves.append(c)
            else:
                cc.name = name
                self.curves.append(cc)
            self.select["values"] = [c.name for c in self.
                curves]
        file.close()


def clear(self):
    if self.temp != None:
        self.temp.vectors = []
        self.load_curve(self.temp)


def move(self):
    try:
        vx = float(self.smovex.get())
        vy = float(self.smovey.get())
    except Exception:
        msg.showerror(parent=self, title="Error", message="
            Translations must have real values")
    else:
        self.temp.translate(vx, vy)
        self.load_curve(self.temp)


def rotate(self):
    try:
        v = float(self.srotate.get())
    except Exception:
        msg.showerror(parent=self, title="Error", message="
            Rotation must have real values")
    else:
        self.temp.rotate_angle(v)
        self.load_curve(self.temp)


def delete(self):
    if self.cursor != -1:
```

```
            self.curves.pop(self.cursor)
            counter = 0
            options = []
            for v in self.select["values"]:
                if counter != self.cursor:
                    options.append(v)
                counter+=1
            self.select["values"] = options
            self.select.set('')
            self.load_curve(Curve(Vector(0, 0), ()))
            self.cursor = -1


    def save(self, *args):
        if self.cursor != -1:
            self.temp.name = self.sname.get()
            self.temp.functions = self.sfunctions.get()
            list_options = list(self.select["values"])
            list_options[self.cursor] = self.temp.name
            self.select["values"] = tuple(list_options)
            self.curves[self.cursor] = self.temp.copy()
            self.select.set(self.select["values"][self.cursor])


    def new(self, *args):
        biggest = 1
        for v in self.select["values"]:
            try:
                ff = int(re.search(r"\d+", re.search(r"new\s\d+",
                   v).group()).group())
                if ff>=biggest:
                    biggest = ff+1
            except AttributeError:
                pass
        name = "new "+str(biggest)

        ncurve = Curve(Vector(0, 0), (), name)
        self.temp = ncurve
        self.load_curve(self.temp)
        self.curves.append(ncurve)
        self.cursor = len(self.curves)-1
        self.select["values"] = list((*self.select["values"], name
           ))
        self.select.set(name)


    def selection(self, *args):
        self.cursor = self.select.current()
```

```
        self.temp = self.curves[self.cursor].copy()
        self.load_curve(self.temp)


    def load_curve(self, curve:Curve):
        self.graphics.clear()
        self.sname.set(curve.name)
        self.sfunctions.set(curve.functions)
        self.smovex.set("0")
        self.smovey.set("0")
        self.srotate.set("0")
        if len(curve.vectors):
            self.graphics.static_drawing([[v.x for v in curve.
                vectors], [v.y for v in curve.vectors]])



class UILink(ttk.Frame):
    def __init__(self, master, curves:List[Curve], links:List[Link
        ]):
        super().__init__(master)
        self.curves = curves
        self.links = links
        self.graphics = Graphics(self, (9.6, 6), row=0, column=0,
            columnspan=1, rowspan=10, dpi=100, title="")
        self.cursor = -1
        self.temp = None
        #----------Controls----------
        #Variables
        self.sname = tk.StringVar(self)
        self.sthickness = tk.StringVar(self)
        self.sadd_curve = tk.StringVar(self)
        self.sremove_curve = tk.StringVar(self)
        self.sconnectionx = tk.StringVar(self)
        self.sconnectiony = tk.StringVar(self)
        self.supper = tk.StringVar(self)
        self.slower = tk.StringVar(self)
        self.bgrid = tk.BooleanVar(self)
        self.bconnections = tk.BooleanVar(self)
        self.bresize = tk.BooleanVar(self)
        #---------
        self.controls = tk.LabelFrame(self, text="Controls")
        self.controls.grid(row=10, column=0, rowspan=1, sticky=tk.
            SE+tk.NW, pady=(0, 2), padx=(2, 2))
        ttk.Label(self.controls, text="Name:").grid(row=0, column
            =0, sticky=tk.SE+tk.NW)
        ttk.Entry(self.controls, textvariable=self.sname).grid(row
            =1, column=0, sticky=tk.SE+tk.NW, columnspan=2)
```

```
ttk.Label(self.controls, text="Thickness:").grid(row=2,
    column=0, sticky=tk.SE+tk.NW)
ttk.Entry(self.controls, textvariable=self.sthickness).
    grid(row=3, column=0, sticky=tk.SE+tk.NW, columnspan=2)
ttk.Label(self.controls, text="Curves:").grid(row=0,
    column=2, sticky=tk.SE+tk.NW, padx=(25, 0))
ttk.Button(self.controls, text="add", command=self.
    add_curve).grid(row=1, column=2, sticky=tk.SE+tk.NW,
    columnspan=1, padx=(25, 0))
self.select_add_curve = ttk.Combobox(self.controls, state=
    "readonly")
ttk.Button(self.controls, text="remove", command=self.
    remove_curve).grid(row=2, column=2, sticky=tk.SE+tk.NW,
     columnspan=1, padx=(25, 0))
self.select_remove_curve = ttk.Combobox(self.controls,
    state="readonly")
self.select_add_curve.grid(row=1, column=3, sticky=tk.SE+
    tk.NW, columnspan=2)
self.select_remove_curve.grid(row=2, column=3, sticky=tk.
    SE+tk.NW, columnspan=2)
ttk.Label(self.controls, text="Connections:").grid(row=0,
    column=5, sticky=tk.SE+tk.NW, padx=(25, 0))
ttk.Label(self.controls, text="x:").grid(row=1, column=5,
    sticky=tk.SE+tk.NW, padx=(25, 0))
ttk.Entry(self.controls, width=10, textvariable=self.
    sconnectionx).grid(row=1, column=6, sticky=tk.SE+tk.NW)
ttk.Label(self.controls, text="y:").grid(row=2, column=5,
    sticky=tk.SE+tk.NW, padx=(25, 0))
ttk.Entry(self.controls, width=10, textvariable=self.
    sconnectiony).grid(row=2, column=6, sticky=tk.SE+tk.NW)
ttk.Button(self.controls, text="add", command=self.
    add_connection).grid(row=3, column=5, sticky=tk.SE+tk.
    NW, columnspan=2, padx=(25, 0))
ttk.Button(self.controls, text="remove", command=self.
    remove_connection).grid(row=4, column=5, sticky=tk.SE+
    tk.NW, columnspan=1, padx=(25, 0))
self.select_connection = ttk.Combobox(self.controls, state
    ="readonly", width=5)
self.select_connection.grid(row=4, column=6, sticky=tk.SE+
    tk.NW, columnspan=1)
ttk.Button(self.controls, text="save", command=self.save).
    grid(row=2, column=9, sticky=tk.SE+tk.NW, columnspan=2,
     padx=(25, 0))
ttk.Button(self.controls, text="new", command=self.new).
    grid(row=1, column=9, sticky=tk.SE+tk.NW, columnspan=1,
     padx=(25, 0))
ttk.Button(self.controls, text="Upload␣file", command=self
    .upload).grid(row=1, column=10, sticky=tk.SE+tk.N,
```

```
            columnspan=1, padx=(0,0))
        ttk.Button(self.controls, text="delete", command=self.
            delete).grid(row=3, column=9, sticky=tk.SE+tk.NW,
            columnspan=2, padx=(25, 0))
        self.select = ttk.Combobox(self.controls, state="readonly"
            , values=[i.name for i in links])
        self.select.bind("<<ComboboxSelected>>", self.selection)
        self.select_connection.bind("<<ComboboxSelected>>", self.
            select_conn)
        self.select.grid(row=0, column=9, sticky=tk.SE+tk.NW,
            columnspan=2, padx=(25, 0))
        ttk.Label(self.controls, text="Upper Limits:").grid(row=4,
             column=0, sticky=tk.SE+tk.NW, padx=(0, 0), pady=(5, 0)
            )
        ttk.Label(self.controls, text="Lower Limits:").grid(row=5,
             column=0, sticky=tk.SE+tk.NW, padx=(0, 0))
        ttk.Entry(self.controls, textvariable=self.supper).grid(
            row=4, column=1, sticky=tk.SE+tk.NW, columnspan=2, pady
            =(5, 0))
        ttk.Entry(self.controls, textvariable=self.slower).grid(
            row=5, column=1, sticky=tk.SE+tk.NW, columnspan=2)
        ttk.Button(self.controls, text="set", command=self.
            set_lims).grid(row=4, column=3, sticky=tk.SE+tk.NW,
            rowspan=2, pady=(5, 0))
        ttk.Checkbutton(self.controls, text="show grid", variable=
            self.bgrid).grid(row=1, column=8, sticky=tk.W, padx=(5,
             0))
        ttk.Checkbutton(self.controls, text="show connections",
            variable=self.bconnections).grid(row=2, column=8,
            sticky=tk.E,padx=(5, 0))
        ttk.Checkbutton(self.controls, text="resize", variable=
            self.bresize).grid(row=0, column=8, sticky=tk.W, padx
            =(5, 0))
        self.curves_available()
        #------------------------------


def set_lims(self, *args):
    if self.select.current() != -1:
        try:
            ups = [int(i) for i in self.supper.get().split(','
                )]
            lower = [int(i) for i in self.slower.get().split('
                ,')]
            if min(ups)<0 or max(ups)>len(self.
                select_remove_curve['values']) or min(lower)<0
                or max(lower)>len(self.select_remove_curve['
                values']):
```

```
                raise IndexError
            except ValueError:
                msg.showerror(parent=self, title="Error", message=
                    "Not a number in the limits of the link")
            except IndexError:
                msg.showerror(parent=self, title="Error", message=
                    f"Limits must be between 1 and the curves
                    available: {len(self.select_remove_curve['
                    values'])}")
        else:
            self.temp.set_lims([i-1 for i in ups], 1)
            self.temp.set_lims([i-1 for i in ups], 0)


    def select_conn(self, *args):
        if self.cursor != -1:
            if self.select_connection.current() != -1:
                vec = self.temp.connections[self.select_connection
                    .current()]
                self.sconnectionx.set(str(vec.x))
                self.sconnectiony.set(str(vec.y))


    def curves_available(self):
        if self.curves:
            self.select_add_curve["values"] = [c.name for c in
                self.curves]


    def add_connection(self):
        if self.cursor != -1:
            try:
                x_val = float(self.sconnectionx.get())
                y_val = float(self.sconnectiony.get())
                self.temp.connections.append(Vector(x_val, y_val))
                self.select_connection["values"] = list(self.
                    select_connection["values"])+ [str(len(self.
                    temp.connections))]
            except Exception as e:
                msg.showerror(parent=self, title="Error", message=
                    "x,y must be real numbers")
            else:
                self.sconnectionx.set('')
                self.sconnectiony.set('')


    def remove_connection(self):
        if self.cursor != -1:
```

```
        if self.select_connection.current() != -1:
            self.temp.connections.pop(self.select_connection.
                current())
            self.select_connection["values"] = [str(i+1) for i
                in range(len(self.temp.connections))]
            self.select_connection.set('')
            self.sconnectionx.set('')
            self.sconnectiony.set('')


def add_curve(self):
    if self.select_add_curve.current() != -1:
        if self.temp:
            self.temp.curves.append(self.curves[self.
                select_add_curve.current()].copy())
            self.select_remove_curve["values"] = (*self.
                select_remove_curve["values"], self.curves[self.
                select_add_curve.current()].name)
            self.select_add_curve.set('')
            name = self.sname.get()
            self.load_link(self.temp)
            self.sname.set(name)
        else:
            msg.showerror(parent=self, title="Error", message=
                "No link in workspace, create a new one or load
                an existing link")


def remove_curve(self):
    if self.select_remove_curve.current()!=-1:
        if self.temp:
            self.temp.curves.pop(self.select_remove_curve.
                current())
            new_list = list(self.select_remove_curve["values"
                ])
            new_list.pop(self.select_remove_curve.current())
            self.select_remove_curve["values"] = tuple(
                new_list)
            self.select_remove_curve.set('')
            name = self.sname.get()
            self.load_link(self.temp)
            self.sname.set(name)
        else:
            msg.showerror(parent=self, title="Error", message=
                "No link in workspace, create a new one or load
                an existing link")
```

```
def upload(self, *args):
    rd = fd.askopenfilename(parent=self, title="Load Data",
        initialdir="C:\\Documents", filetypes=(("CSV", "*.csv")
        ,("CSV", "*.txt")))
    if rd:
        file = open(rd)
        try:
            link = Link.build_from_io(file)
        except Exception:
            msg.showerror(parent=self, title="Error", message=
                "Could not process selected file")
        else:
            name = rd[-rd[::-1].find("/"):-4]
            link.name = name
            self.links.append(link)
            self.select["values"] = [l.name for l in self.
                links]
        file.close()


def save(self):
    if self.cursor != -1:
        try:
            n = float(self.sthickness.get())
            if n<=0:
                raise ValueError("")
        except Exception:
            msg.showerror(parent=self, title="Error", message=
                "Thickness must be set to a number>0")
        else:
            self.temp.name = self.sname.get()
            self.temp.thickness = float(self.sthickness.get())
            list_options = list(self.select["values"])
            list_options[self.cursor] = self.temp.name
            self.select["values"] = tuple(list_options)
            self.links[self.cursor] = self.temp.copy()
            self.select.set(self.select["values"][self.cursor
                ])


def new(self):
    biggest = 1
    for v in self.select["values"]:
        try:
            ff = int(re.search(r"\d+", re.search(r"new\s\d+",
                v).group()).group())
            if ff>=biggest:
                biggest = ff+1
```

```
            except AttributeError:
                pass
        name = "new "+str(biggest)

        nlink = Link(Vector(0, 0), [], [], 0.1, name)
        self.temp = nlink
        self.load_link(self.temp)
        self.links.append(nlink)
        self.cursor = len(self.links)-1
        self.select["values"] = (*self.select["values"], name)
        self.select.set(name)


    def delete(self):
        if self.cursor != -1:
            self.links.pop(self.cursor)
            counter = 0
            options = []
            for v in self.select["values"]:
                if counter != self.cursor:
                    options.append(v)
                counter+=1
            self.select["values"] = options
            self.select.set('')
            self.load_link(Link(Vector(0, 0), (), (), 0))
            self.cursor = -1


    def selection(self, *args):
        self.cursor = self.select.current()
        self.temp = self.links[self.cursor].copy()
        self.load_link(self.temp)


    def load_link(self, link:Link):
        self.graphics.clear()
        self.sname.set(link.name)
        self.sthickness.set(str(round(link.thickness, 6)))
        self.select_remove_curve["values"] = [c.name for c in link
           .curves]
        self.select_remove_curve.set('')
        self.select_connection["values"] = [str(i+1) for i in
           range(len(link.connections))]
        self.select_connection.set('')
        self.sconnectionx.set('')
        self.sconnectiony.set('')
        self.slower.set('')
        self.supper.set('')
```

```
            if link.upper:
                v = ''
                for func in link.upper:
                    for i in range(len(link.curves)):
                        if link.curves[i].function == func:
                            v+=str(i+1)+","
                            break

                if v:
                    self.supper.set(v[:-1])

            if link.lower:
                v = ''
                for func in link.lower:
                    for i in range(len(link.curves)):
                        if link.curves[i].function == func:
                            v+=str(i+1)+","
                            break
                if v:
                    self.slower.set(v[:-1])

            graphics.plot_link(link, axes=self.graphics.axis,
               show_connections=self.bconnections.get(), grid=self.
               bgrid.get(), resize=self.bresize.get())
            self.graphics.render()



class UIMechanism(ttk.Frame):
    def __init__(self, master, links:List[Link], mechanisms:List[
       Mechanism]):
        super().__init__(master)
        self.links = links
        self.mechanisms = mechanisms
        self.graphics = Graphics(self, (9.6, 6), row=0, column=0,
           columnspan=1, rowspan=10, dpi=100, title="")
        self.cursor = -1
        self.temp = None
        self.moved = Vector(0, 0)
        self.rotated = 0.0
        #Variables
        self.sname = tk.StringVar(self)
        self.slider = tk.StringVar(self)
        self.soffset = tk.StringVar(self)
        self.srotation = tk.StringVar(self)
        self.angle_rotate = tk.DoubleVar(self)
        self.inversion = tk.IntVar(self)
```

```
self.stress_analysis = tk.IntVar(self)
self.crank = None
self.coupler = None
self.output = None
self.ground = None
#----------

self.controls = tk.LabelFrame(self, text="Controls")
self.controls.grid(row=10, column=0, rowspan=1, sticky=tk.
    SE+tk.NW, pady=(0, 2), padx=(2, 2))
ttk.Label(self.controls, text="Name:").grid(row=1, column
    =0, sticky=tk.SE+tk.NW)
ttk.Label(self.controls, text="Connections:").grid(row=0,
    column=2, sticky=tk.SE+tk.NW, padx=(20, 0))
ttk.Label(self.controls, text="Mech. Rotation (   ):").grid
    (row=0, column=4, sticky=tk.SE+tk.N, padx=(10, 0))
ttk.Entry(self.controls, textvariable=self.sname).grid(row
    =1, column=1, sticky=tk.SE+tk.NW)
ttk.Entry(self.controls, textvariable=self.srotation).grid
    (row=0, column=5, sticky=tk.SE+tk.NW, padx=(0, 15))
ttk.Checkbutton(self.controls, text="Slider-Crank",
    variable=self.slider, onvalue="Offset:", offvalue="
    Ground:", command=self.change_type).grid(row=0, column
    =0, sticky=tk.SE+tk.NW)


ttk.Label(self.controls, text="Crank:").grid(row=2, column
    =0, sticky=tk.SE+tk.NW)
ttk.Label(self.controls, text="Coupler:").grid(row=3,
    column=0, sticky=tk.SE+tk.NW)
ttk.Label(self.controls, text="Output:").grid(row=4,
    column=0, sticky=tk.SE+tk.NW)
ttk.Label(self.controls, textvariable=self.slider).grid(
    row=5, column=0, sticky=tk.SE+tk.NW)
self.crank = ttk.Combobox(self.controls, state="readonly")
self.coupler = ttk.Combobox(self.controls, state="readonly
    ")
self.output = ttk.Combobox(self.controls, state="readonly"
    )
self.ground = ttk.Combobox(self.controls, state="readonly"
    )
self.offset_entry = ttk.Entry(self.controls, textvariable=
    self.soffset)

self.connections = [ttk.Combobox(self.controls, state="
    readonly") for i in range(8)]
for i, con in enumerate(self.connections):
    con.grid(row=i//2+2, column=i%2+2, sticky=tk.SE+tk.NW,
```

```
            padx=((i+1)%2*20, 0))


ttk.Label(self.controls, text="Input angle (  )").grid(row
    =2, column=4, columnspan=1, sticky=tk.SE+tk.NW, padx
    =(10, 0))
ttk.Label(self.controls, textvariable=self.angle_rotate).
    grid(row=2, column=5, columnspan=1, sticky=tk.SE+tk.NW,
     padx=(0, 10))
ttk.Scale(self.controls, variable=self.angle_rotate, from_
    =0, to=360).grid(row=3, column=4, columnspan=2, sticky=
    tk.SE+tk.NW, padx=(10, 10))

ttk.Button(self.controls, text="Redraw", command=self.
    redraw).grid(row=4, column=4, columnspan=2, sticky=tk.
    SE+tk.NW, padx=(10, 10))
ttk.Button(self.controls, text="Animate", command=self.
    animate).grid(row=5, column=4, columnspan=2, sticky=tk.
    SE+tk.NW, padx=(10, 10))

ttk.Checkbutton(self.controls, text="Inversion", variable=
    self.inversion, onvalue=1, offvalue=0).grid(row=0,
    column=6, columnspan=2, sticky=tk.SE+tk.NW, padx=(15,
    0))
ttk.Checkbutton(self.controls, text="Stress Analysis",
    variable=self.stress_analysis, onvalue=1, offvalue=0).
    grid(row=1, column=6, columnspan=2, sticky=tk.SE+tk.NW,
     padx=(15, 0))
ttk.Button(self.controls, text="new", command=self.new).
    grid(row=3, column=6, sticky=tk.SE+tk.NW, columnspan=2,
     padx=(15, 0))
ttk.Button(self.controls, text="save", command=self.save).
    grid(row=4, column=6, sticky=tk.SE+tk.NW, columnspan=2,
     padx=(15, 0))
ttk.Button(self.controls, text="delete", command=self.
    delete).grid(row=5, column=6, sticky=tk.SE+tk.NW,
    columnspan=2, padx=(15, 0))
self.select = ttk.Combobox(self.controls, state="readonly"
    , values=[i.name for i in mechanisms])
self.select.bind("<<ComboboxSelected>>", self.selection)
self.crank.bind("<<ComboboxSelected>>", self.selection)
self.coupler.bind("<<ComboboxSelected>>", self.selection)
self.output.bind("<<ComboboxSelected>>", self.selection)
self.ground.bind("<<ComboboxSelected>>", self.selection)
self.select.grid(row=2, column=6, sticky=tk.SE+tk.NW,
    columnspan=2, padx=(15, 0))
self.crank.grid(row=2, column=1, sticky=tk.SE+tk.NW)
self.coupler.grid(row=3, column=1, sticky=tk.SE+tk.NW)
```

```
        self.output.grid(row=4, column=1, sticky=tk.SE+tk.NW)
        self.ground.grid(row=5, column=1, sticky=tk.SE+tk.NW)
        self.slider.set("Offset:")
        self.slider.set("Ground:")
        self.links_available()
        #-----------------------------


    def links_available(self):
        if self.links:
            link_names = [l.name for l in self.links]
            self.crank["values"] = link_names
            self.coupler["values"] = link_names
            self.output["values"] = link_names
            self.ground["values"] = link_names


    def save(self):
        if self.select.current() != -1:
            try:
                name = self.sname.get()
                specific_msg = "Check rotation"
                rotation = float(self.srotation.get())*pi/180
                crank = self.crank.current()
                coupler = self.coupler.current()
                output = self.output.current()
                crank_connections = [self.connections[0].current()
                    , self.connections[1].current()]
                coupler_connections = [self.connections[2].current
                    (), self.connections[3].current()]
                output_connections = [self.connections[4].current
                    () ,]
                if coupler == -1 or crank == -1 or output == -1:
                    specific_msg="Crank, Coupler or Output not
                        selected"
                    raise ValueError("")

                if -1 in crank_connections:
                    specific_msg="Missing one or more connections
                        for crank"
                    raise ValueError("")
                if crank_connections[0] == crank_connections[1]:
                    specific_msg="Cannot repeat crank connection"
                    raise ValueError("")

                if -1 in coupler_connections:
                    specific_msg="Missing one or more connections
                        for coupler"
```

```
                raise ValueError("")
            if coupler_connections[0] == coupler_connections
               [1]:
                specific_msg="Cannot repeat coupler connection
                   "
                raise ValueError("")

            if -1 in output_connections:
                    specific_msg="Missing one or more
                       connections for output"
                    raise ValueError("")



            if self.slider.get() == "Ground:":
                ground = self.ground.current()
                ground_connections = [self.connections[6].
                   current(), self.connections[7].current()]
                output_connections.append(self.connections[5].
                   current())


                if -1 in output_connections:
                    specific_msg="Missing one or more
                       connections for output"
                    raise ValueError("")

                if output_connections[0] == output_connections
                   [1]:
                    specific_msg="Missing one or more
                       connections for crank"
                    raise ValueError("")

                if ground == -1:
                    specific_msg="Ground not selected"
                    raise ValueError("")
                if -1 in ground_connections:
                    specific_msg="Missing one or more
                       connections for ground"
                    raise ValueError("")

            else:
                specific_msg="Offset not a number"
                offset = float(self.soffset.get())

        except Exception:
            msg.showerror(parent=self, title="Error", message=
               "Invalid parameters\n"+specific_msg)
```

```
else:
    if self.slider.get() == "Ground:":
        stress_analysis=False
        differential_ = 0
        density_ = 0
        if self.stress_analysis.get():
            dx = askfloat("Stress Analysis", "
                differentials for:", parent=self)
            density = askfloat("Stress Analysis", "
                density of the material:", parent=self)
            if type(dx) == float and type(density) ==
                float:
                if dx < 0 or density < 0:
                    msg.showerror(parent=self, title="
                        Error", message="Cannot work
                        with negative density nor
                        differentials")
                else:
                    differential_ = dx
                    density_ = density
                    stress_analysis = True
        self.temp = Mechanism(origin=Vector(0, 0),
            rotation=rotation, links=[self.links[crank
            ].copy(), self.links[coupler].copy(), self.
            links[output].copy(), self.links[ground].
            copy()],\
                connections=[crank_connections,
                    coupler_connections,
                    output_connections,
                    ground_connections], name=name,
                    stress_analysis=stress_analysis,
                    dx=differential_, density=
                    density_, init=True)
        self.temp.moved = Vector(0, 0)
        self.temp.rotation = rotation
    else:
        self.temp = SliderCrank(origin=Vector(0, 0),
            rotation=rotation, links=[self.links[crank
            ].copy(), self.links[coupler].copy(), self.
            links[output].copy()],\
                    connections=[crank_connections,
                        coupler_connections,
                        output_connections], offset=
                        offset, name=name, init=True)
        self.temp.moved = Vector(0, 0)
        self.temp.rotation = rotation

    list_options = list(self.select["values"])
```

```
            list_options[self.select.current()] = self.temp.
                name
            self.select["values"] = tuple(list_options)
            self.mechanisms[self.select.current()] = self.temp
                .copy()
            self.select.set(self.select["values"][self.select.
                current()])
            self.load_mechanism(self.temp)



def new(self):
    biggest = 1
    for v in self.select["values"]:
        try:
            ff = int(re.search(r"\d+", re.search(r"new\s\d+",
                v).group()).group())
            if ff>=biggest:
                biggest = ff+1
        except AttributeError:
            pass
    name = "new "+str(biggest)
    available = [self.crank, self.coupler, self.output, self.
        ground]

    for option in available:
        option.set('')

    for con in self.connections:
        con.set('')

    self.temp = None
    self.sname.set(name)
    self.load_mechanism(self.temp)
    self.mechanisms.append(self.temp)
    self.cursor = len(self.links)-1
    self.select["values"] = (*self.select["values"], name)
    self.select.set(name)


def delete(self):
    if self.select.current() != -1:
        self.mechanisms.pop(self.select.current())
        counter = 0
        options = list(self.select["values"])
        options.pop(self.select.current())
        self.select["values"] = options
        self.select.set('')
```

```
            self.temp = None
            self.load_mechanism(self.temp)


    def selection(self, comp):
        if comp.widget == self.select:
            self.load_mechanism(self.mechanisms[self.select.
                current()])
            self.temp = self.mechanisms[self.select.current()].
                copy()
            return
        try:
            available = [self.crank, self.coupler, self.output,
                self.ground]
            i = available.index(comp.widget)
            self.connections[i*2]['values'] = [f'x:{v.x} y:{v.y}'
                for v in self.links[available[i].current()].
                connections]
            self.connections[i*2+1]['values'] = [f'x:{v.x} y:{v.y}
                ' for v in self.links[available[i].current()].
                connections]
            self.temp = None
        except ValueError:
            #Selection of mechanism, shouldn't happen
            pass


    def change_type(self):
        if self.slider.get()=="Offset:":
            self.ground.grid_forget()
            self.connections[-1]['state'] = 'disabled'
            self.connections[-2]['state'] = 'disabled'
            self.connections[5]['state'] = 'disabled'
            self.offset_entry.grid(row=5, column=1, sticky=tk.SE+
                tk.NW)
        else:
            self.offset_entry.grid_forget()
            self.connections[-1]['state'] = 'readonly'
            self.connections[-2]['state'] = 'readonly'
            self.connections[5]['state'] = 'readonly'
            self.ground.grid(row=5, column=1, sticky=tk.SE+tk.NW)


    def load_mechanism(self, mechanism:Mechanism|SliderCrank,
        animate_:bool=False):
        self.graphics.clear()
        if mechanism == None:
            return
```

```
self.sname.set(mechanism.name)
self.srotation.set(str(round(mechanism.rotation*180/pi, 2)
    ))
self.crank.set(mechanism.links[0].name)
self.coupler.set(mechanism.links[1].name)
self.output.set(mechanism.links[2].name)
self.stress_analysis.set(int(mechanism.stress_analysis))
self.moved = mechanism.moved.copy()
self.rotated = mechanism.rotation
self.connections[0]['values'] = [f'x:{v.x} y:{v.y}' for v
    in mechanism.links[0].connections]
self.connections[1]['values'] = [f'x:{v.x} y:{v.y}' for v
    in mechanism.links[0].connections]
self.connections[2]['values'] = [f'x:{v.x} y:{v.y}' for v
    in mechanism.links[1].connections]
self.connections[3]['values'] = [f'x:{v.x} y:{v.y}' for v
    in mechanism.links[1].connections]
self.connections[4]['values'] = [f'x:{v.x} y:{v.y}' for v
    in mechanism.links[2].connections]
self.connections[5]['values'] = [f'x:{v.x} y:{v.y}' for v
    in mechanism.links[2].connections]


self.connections[0].current(mechanism.connections[0][0])
self.connections[1].current(mechanism.connections[0][1])

self.connections[2].current(mechanism.connections[1][0])
self.connections[3].current(mechanism.connections[1][1])

self.connections[4].current(mechanism.connections[2][0])


if self.crank.current() == -1:
    msg.showwarning("Warning", message="Crank link has
        been deleted or its name has changed if new changes
        are going to be saved to the mechanism check for
        the correct link")
if self.coupler.current() == -1:
    msg.showwarning("Warning", message="Coupler link has
        been deleted or its name has changed if new changes
        are going to be saved to the mechanism check for
        the correct link")
if self.output.current() == -1:
    msg.showwarning("Warning", message="Output link has
        been deleted or its name has changed if new changes
        are going to be saved to the mechanism check for
        the correct link")
```

```
if type(mechanism) == Mechanism:
    self.slider.set("Ground:")
    self.ground.set(mechanism.links[3].name)

    self.connections[6]['values'] = [f'x:{v.x} y:{v.y}'
       for v in mechanism.links[3].connections]
    self.connections[7]['values'] = [f'x:{v.x} y:{v.y}'
       for v in mechanism.links[3].connections]
    self.connections[5].current(mechanism.connections
       [2][1])
    self.connections[6].current(mechanism.connections
       [3][0])
    self.connections[7].current(mechanism.connections
       [3][1])

    if self.ground.current() == -1:
        msg.showwarning("Warning", message="Ground link
            has been deleted or its name has changed if new
            changes are going to be saved to the mechanism
            check for the correct link")


elif type(mechanism) == SliderCrank:
    self.sname.set(mechanism.name)
    self.slider.set("Offset:")
    self.soffset.set(mechanism.c)

    if self.crank.current() == -1:
        msg.showwarning("Warning", message="Crank link has
            been deleted or its name has changed if new
            changes are going to be saved to the mechanism
            check for the correct link")
    if self.coupler.current() == -1:
        msg.showwarning("Warning", message="Coupler link
            has been deleted or its name has changed if new
            changes are going to be saved to the mechanism
            check for the correct link")
    if self.output.current() == -1:
        msg.showwarning("Warning", message="Output link
            has been deleted or its name has changed if new
            changes are going to be saved to the mechanism
            check for the correct link")

self.change_type()

if animate_:
```

```
            animation = graphics.plot_rotation_mech(mechanism,
                frames=100, inversion=self.inversion.get(), axes=
                self.graphics.axis, fig=self.graphics.fig)
        else:
            solution = mechanism.solution(self.angle_rotate.get()*
                pi/180)[self.inversion.get()]
            graphics.plot_mechanism(*solution, self.graphics.axis)

        self.graphics.render()


    def redraw(self, *args):
        if self.temp:
            self.load_mechanism(self.temp)
            return
        msg.showerror("Error", message="Mechanism must be saved
            before")


    def animate(self, *arga):
        if self.temp:
            self.load_mechanism(self.temp, animate_=True)
            return
        msg.showerror("Error", message="Mechanism must be saved
            before")



class UIMachine(ttk.Frame):
    def __init__(self, master, links:List[Link], mechanisms:List[
        Mechanism], machines:List[Machine]):
        super().__init__(master)
        self.links = links
        self.mechanisms = mechanisms
        self.machines = machines
        self.graphics = Graphics(self, (9.6, 6), row=0, column=0,
            columnspan=1, rowspan=10, dpi=100, title="")
        self.temp = None
        self.temp_mechanisms = []
        self.background = []
        #Variables
        self.sname = tk.StringVar(self)
        self.angle_rotate = tk.DoubleVar(self)
        self.bsave_image = tk.BooleanVar(self)
        self.spower_graph = tk.StringVar(self)
        self.sinversion_array = tk.StringVar(self)
        #---------
        self.controls = tk.LabelFrame(self, text="Controls")
```

```
self.controls.grid(row=10, column=0, rowspan=1, sticky=tk.
    SE+tk.NW, pady=(0, 2), padx=(2, 2))

ttk.Label(self.controls, text="Name:").grid(row=0, column
    =0, columnspan=1, sticky=tk.SE+tk.NW, pady=(10, 0))
ttk.Entry(self.controls, textvariable=self.sname).grid(row
    =0, column=1, columnspan=1, sticky=tk.SE+tk.NW, pady
    =(10, 0))

ttk.Label(self.controls, text="Mechanisms:").grid(row=1,
    column=0, columnspan=1, sticky=tk.SE+tk.NW)
self.integrating_mech = ttk.Combobox(self.controls, state=
    "readonly")
self.integrating_mech.grid(row=1, column=1, columnspan=1,
    sticky=tk.SE+tk.NW)
ttk.Button(self.controls, text="remove", command=self.
    remove_mech).grid(row=2, column=0, columnspan=2, sticky
    =tk.SE+tk.NW)

ttk.Label(self.controls, text="Mechanism␣Library:").grid(
    row=3, column=0, columnspan=1, sticky=tk.SE+tk.NW)
ttk.Label(self.controls, text="Available:").grid(row=4,
    column=0, columnspan=1, sticky=tk.SE+tk.NW)
self.available_mechanisms = ttk.Combobox(self.controls,
    state="readonly")
self.available_mechanisms.grid(row=4, column=1, columnspan
    =1, sticky=tk.SE+tk.NW)
ttk.Button(self.controls, text="add", command=self.
    add_mech).grid(row=5, column=0, columnspan=2, sticky=tk
    .SE+tk.NW)

ttk.Label(self.controls, text="Power␣Graph:").grid(row=0,
    column=2, columnspan=1, sticky=tk.SE+tk.NW, padx=(3, 0)
    )
ttk.Entry(self.controls, textvariable=self.spower_graph).
    grid(row=1, column=2, columnspan=2, sticky=tk.SE+tk.NW,
     padx=(10, 0))
ttk.Label(self.controls, text="Inversion␣Array:").grid(row
    =2, column=2, columnspan=1, sticky=tk.SE+tk.NW, padx
    =(3, 0))
ttk.Entry(self.controls, textvariable=self.
    sinversion_array).grid(row=3, column=2, columnspan=2,
    sticky=tk.SE+tk.NW, padx=(10, 0))

ttk.Checkbutton(self.controls, text="Save␣Image/Video",
    variable=self.bsave_image, onvalue=True, offvalue=False
    ).grid(row=4, column=5, sticky=tk.SE+tk.NW, padx=(0,
    10))
```

```
ttk.Label(self.controls, text="Input angle (  )").grid(row
    =0, column=4, columnspan=1, sticky=tk.SE+tk.NW, padx
    =(10, 0))
ttk.Label(self.controls, textvariable=self.angle_rotate).
    grid(row=0, column=5, columnspan=1, sticky=tk.SE+tk.NW,
     padx=(0, 10))
ttk.Scale(self.controls, variable=self.angle_rotate, from_
    =0, to=360).grid(row=1, column=4, columnspan=2, sticky=
    tk.SE+tk.NW, padx=(10, 10))

ttk.Button(self.controls, text="Redraw", command=self.
    redraw).grid(row=2, column=4, columnspan=2, sticky=tk.
    SE+tk.NW, padx=(10, 10))
ttk.Button(self.controls, text="Animate", command=self.
    animate).grid(row=3, column=4, columnspan=2, sticky=tk.
    SE+tk.NW, padx=(10, 10))

ttk.Label(self.controls, text="Background:").grid(row=0,
    column=6, columnspan=2, sticky=tk.SE+tk.NW, padx=(10,
    10))
self.list_background = ttk.Combobox(self.controls, state="
    readonly")
self.list_background.grid(row=1, column=6, columnspan=2,
    sticky=tk.SE+tk.NW, padx=(10, 10))
self.list_links = ttk.Combobox(self.controls, state="
    readonly")
self.list_links.grid(row=2, column=7, columnspan=1, sticky
    =tk.SE+tk.NW, padx=(0, 10))

ttk.Button(self.controls, text="add", command=self.
    add_background).grid(row=2, column=6, columnspan=1,
    sticky=tk.SE+tk.NW, padx=(10, 0))
ttk.Button(self.controls, text="remove", command=self.
    remove_background).grid(row=3, column=6, columnspan=2,
    sticky=tk.SE+tk.NW, padx=(10, 10))

ttk.Button(self.controls, text="new", command=self.new).
    grid(row=2, column=8, sticky=tk.SE+tk.NW, columnspan=2,
     padx=(15, 0))
ttk.Button(self.controls, text="save", command=self.save).
    grid(row=3, column=8, sticky=tk.SE+tk.NW, columnspan=2,
     padx=(15, 0))
ttk.Button(self.controls, text="delete", command=self.
    delete).grid(row=4, column=8, sticky=tk.SE+tk.NW,
    columnspan=2, padx=(15, 0))
self.select = ttk.Combobox(self.controls, state="readonly"
    , values=[i.name for i in machines])
self.select.bind("<<ComboboxSelected>>", self.selection)
```

```
        self.select.grid(row=1, column=8, sticky=tk.SE+tk.NW,
            columnspan=2, padx=(15, 0))
        self.mechanisms_available()


    def links_available(self):
        if self.links:
            self.list_links["values"] = [v.name for v in self.
                links]


    def mechanisms_available(self):
        if self.mechanisms:
            self.available_mechanisms["values"] = [v.name for v in
                self.mechanisms]



    def check_power_graph(self):
        inp = self.spower_graph.get().strip()
        graph = []
        stack = []
        current = ""
        incomplet_graph = {}
        max_ = None
        numbers = [f"{i}" for i in range(10)]
        symbols_ = ""
        if len(inp) == 0:
            raise ValueError(f"Invalid power graph")

        for i, m in enumerate(inp):
            if m in numbers:
                current+=m
                symbols_+=m
            elif m == "{" or m == "}":
                if current == "":
                    if m == "}" and symbols_[-1] == "{":
                        stack = []
                        symbols_+=m
                        continue
                    else:
                        raise ValueError(f"Invalid input at
                            position {i+1}")
                stack.append(int(current))
                if max_ == None or int(current)>max_:
                    max_ = int(current)
                current = ""
                symbols_+=m
```

```
                if m == "}":
                    incomplet_graph[stack[0]] = stack[1:]
                    stack = []
            elif m == "␣":
                continue
            elif m == ",":
                if current == "" or len(stack) == 0:
                    raise ValueError(f"Invalid␣input␣at␣position␣{
                        i+1}")
                stack.append(int(current))
                if int(current)>max_:
                    max_ = int(current)
                current = ""
            else:
                raise ValueError("Invalid␣input␣"+ m + f"␣at␣
                    position␣{i+1}")

        graph = [[] for i in range(max_+1)]
        for i in range(max_+1):
            if i in incomplet_graph:
                graph[i] = incomplet_graph[i]
            else:
                graph[i] = []
        return graph


    def check_inversion_array(self):
        inp = self.sinversion_array.get()
        if inp == "0" or inp == "1":
            return int(inp)
        m = re.fullmatch("[01]+", inp)
        if m:
            return [int(i) for i in m.group()]
        raise ValueError("Inversion␣array␣not␣valid␣must␣be␣a␣
            series␣of␣0s␣and␣1s␣or␣a␣single␣0␣or␣1")


    def save(self):
        if self.select.current() != -1:
            try:
                pw_graph = self.check_power_graph()
                name = self.sname.get()
                self.temp = Machine(self.temp_mechanisms, pw_graph
                    , auto_adjust=True, name=name)
            except ValueError as e:
                msg.showerror(parent=self, title="Error", message=
                    str(e))
            except AssertionError as e:
```

```
            msg.showerror(parent=self, title="Error", message=
                str(e))
        else:

            list_options = list(self.select["values"])
            list_options[self.select.current()] = self.temp.
                name
            self.select["values"] = list_options
            self.machines[self.select.current()] = self.temp.
                copy()
            self.select.set(self.select["values"][self.select.
                current()])
            self.load_machine(self.temp)


def new(self):
    biggest = 1
    for v in self.select["values"]:
        try:
            ff = int(re.search(r"\d+", re.search(r"new\s\d+",
                v).group()).group())
            if ff>=biggest:
                biggest = ff+1
        except AttributeError:
            pass
    name = "new␣"+str(biggest)

    self.temp = None
    self.machines.append(self.temp)
    self.select["values"] = list(self.select["values"]) + [
        name,]
    self.select.set(name)
    self.load_machine(self.temp)
    self.temp_mechanisms = []
    self.sname.set(name)
    self.integrating_mech["values"] = []
    self.integrating_mech.set('')
    self.spower_graph.set('')
    self.sinversion_array.set('')


def delete(self):
    if self.select.current() != -1:
        self.machines.pop(self.select.current())
        counter = 0
        options = list(self.select["values"])
        options.pop(self.select.current())
        self.select["values"] = options
```

```
            self.select.set('')
            self.temp = None
            self.load_machine(self.temp)
            self.temp_mechanisms = []


    def remove_mech(self):
        if self.integrating_mech.current() != -1:
            if self.temp != None:
                self.temp.mechanisms.pop(self.integrating_mech.
                    current())
            v = list(self.integrating_mech['values'])
            v.pop(self.integrating_mech.current())
            self.integrating_mech["values"] = v
            self.temp_mechanisms.pop(self.integrating_mech.current
                ())
            self.integrating_mech.set('')


    def add_mech(self):
        if self.available_mechanisms.current() != -1:
            if self.mechanisms[self.available_mechanisms.current()
                ] == None:
                msg.showerror(parent=self, title="Error", message=
                    "Cannot add a mechanism that has not been saved
                    ")
                return
            self.temp_mechanisms.append(self.mechanisms[self.
                available_mechanisms.current()].copy())
            self.integrating_mech["values"] = [*self.
                integrating_mech["values"], self.mechanisms[self.
                available_mechanisms.current()].name]


    def selection(self, *args):
        self.sinversion_array.set('1')
        self.load_machine(self.machines[self.select.current()])
        if self.machines[self.select.current()]:
            self.temp = self.machines[self.select.current()].copy
                ()
        else:
            self.temp = None


    def load_machine(self, machine:Machine, animate_:bool=False):
        self.graphics.clear()
        if machine != None:
            self.sname.set(machine.name)
```

```
if self.sinversion_array.get().strip() == "":
    self.sinversion_array.set(1)
self.temp_mechanisms = [mech.copy() for mech in
   machine.mechanisms]
self.integrating_mech["values"] = [mech.name for mech
   in machine.mechanisms]
self.spower_graph.set(''.join([str(i)+"{"+','.join([
   str(o) for o in machine.power_graph[i]])+"}" for i
   in range(len(machine.power_graph))]))
try:
    inversions = self.check_inversion_array()
except ValueError as e:
    msg.showerror(parent=self, title="Error", message=
       str(e))
else:
    if animate_:
        remember = self.bsave_image.get()
        self.bsave_image.set(False)
        self.load_machine(machine)
        xlims = [abs(x) for x in list(self.graphics.
           axis.get_xlim())]
        ylims = [abs(y) for y in list(self.graphics.
           axis.get_ylim())]
        biggest = max(ylims+xlims)
        xlims[0] = biggest*-1.15
        xlims[1] = biggest*1.15
        ylims[0] = biggest*-1.15
        ylims[1] = biggest*1.15
        self.graphics.clear()
        self.graphics.render()
        self.bsave_image.set(remember)
        save_name = ""
        if self.bsave_image.get():
            save_name = fd.asksaveasfilename(parent=
               self, title="Save screen", filetypes=((
               "GIF", "gif"),), initialdir="C:",
               defaultextension="gif")

        for link in self.background:
            graphics.plot_link(link, self.graphics.
               axis, color="black")

        animation = graphics.plot_rotation_mach(
           machine, frames=100, lims=[xlims, ylims],
           inversion=inversions, axes=self.graphics.
           axis, fig=self.graphics.fig, save=save_name
           )
```

```
                else:
                    save_name = ""
                    if self.bsave_image.get():
                        save_name = fd.asksaveasfilename(parent=
                            self, title="Save screen", filetypes=((
                            "PNG", "png"),), initialdir="C:",
                            defaultextension="png")
                    try:
                        solution = machine.solution(self.
                            angle_rotate.get()*pi/180, inversions)
                    except Exception as e:
                        msg.showerror(parent=self, title="Error",
                            message=str(e))

                    for link in self.background:
                        graphics.plot_link(link, self.graphics.
                            axis, color="black")

                    graphics.plot_machine(solution, self.graphics.
                        axis)
                    if save_name:
                        self.graphics.fig.savefig(save_name)

                self.graphics.render()
            else:
                self.sinversion_array.set('')
                self.spower_graph.set('')


        def add_background(self, *args):
            if self.list_links.current() != -1:
                self.background.append(self.links[self.list_links.
                    current()])
                self.list_background["values"] = [link.name for link
                    in self.background]


        def remove_background(self, *args):
            if self.list_background.current() != -1:
                new_values = list(self.list_background["values"])
                new_values.pop(self.list_background.current())
                self.list_background["values"] = new_values
                self.background.pop(self.list_background.current())
                self.list_background.set('')


        def redraw(self, *args):
            if self.temp:
```

```
            self.load_machine(self.temp)
            return
        msg.showerror("Error", message="Machine must be saved
            before")


    def animate(self, *args):
        if self.temp:
            self.load_machine(self.temp, animate_=True)
            return
        msg.showerror("Error", message="Machine must be saved
            before")



class UIStress(ttk.Frame):
    def __init__(self, master, machines:List[Machine]):
        super().__init__(master)
        self.machines = machines
        self.graphics = Graphics(self, (9.6, 6), row=0, column=0,
            columnspan=1, rowspan=10, dpi=100, title="")
        self.angle_rotate = tk.DoubleVar(self)
        self.sangular_speed = tk.StringVar(self)
        self.sangular_acceleration = tk.StringVar(self)
        self.sinversion_array = tk.StringVar(self)
        self.smoments_array = tk.StringVar(self)
        self.smoments_array_coupler = tk.StringVar(self)
        self.bmass_center = tk.BooleanVar(self)
        self.breport = tk.BooleanVar(self)
        #-----
        self.controls = tk.LabelFrame(self, text="Controls")
        self.controls.grid(row=10, column=0, rowspan=1, sticky=tk.
            SE+tk.NW, pady=(0, 2), padx=(2, 2))
        ttk.Button(self.controls, text="Solve", command=self.solve
            ).grid(row=2, column=6, columnspan=2, sticky=tk.SE+tk.
            NW, padx=(20, 0))
        self.select = ttk.Combobox(self.controls, state="readonly"
            , values=[i.name for i in machines])
        ttk.Label(self.controls, text="Machine:").grid(row=0,
            column=0, columnspan=1, sticky=tk.SE+tk.NW, padx=(10,
            0))
        self.select.grid(row=0, column=1, sticky=tk.SE+tk.NW,
            columnspan=1, padx=(15, 0))

        ttk.Label(self.controls, text="Input angle (  )").grid(row
            =1, column=0, columnspan=1, sticky=tk.SE+tk.NW, padx
            =(10, 0))
        ttk.Label(self.controls, textvariable=self.angle_rotate).
```

```
            grid(row=1, column=1, columnspan=1, sticky=tk.SE+tk.NW,
             padx=(0, 10))
        ttk.Scale(self.controls, variable=self.angle_rotate, from_
            =0, to=360).grid(row=2, column=0, columnspan=2, sticky=
            tk.SE+tk.NW, padx=(10, 10))
        ttk.Label(self.controls, text="Angular␣speed").grid(row=3,
             column=0, columnspan=1, sticky=tk.SE+tk.NW, padx=(10,
            0))
        ttk.Label(self.controls, text="Angular␣acceleration").grid
            (row=4, column=0, columnspan=1, sticky=tk.SE+tk.NW,
            padx=(10, 0))
        ttk.Entry(self.controls, textvariable=self.sangular_speed)
            .grid(row=3, column=1, columnspan=1, sticky=tk.SE+tk.NW
            )
        ttk.Entry(self.controls, textvariable=self.
            sangular_acceleration).grid(row=4, column=1, columnspan
            =1, sticky=tk.SE+tk.NW)
        ttk.Label(self.controls, text="Inversion␣Array:").grid(row
            =0, column=2, columnspan=1, sticky=tk.SE+tk.NW, padx
            =(10, 0))
        ttk.Entry(self.controls, textvariable=self.
            sinversion_array).grid(row=1, column=2, columnspan=2,
            sticky=tk.SE+tk.NW, padx=(10, 0))
        ttk.Label(self.controls, text="External␣Moments␣Cranks:").
            grid(row=2, column=2, columnspan=1, sticky=tk.SE+tk.NW,
             padx=(10, 0))
        ttk.Entry(self.controls, textvariable=self.smoments_array)
            .grid(row=3, column=2, columnspan=2, sticky=tk.SE+tk.NW
            , padx=(10, 0))
        ttk.Label(self.controls, text="External␣Moments␣Couplers:"
            ).grid(row=2, column=4, columnspan=1, sticky=tk.SE+tk.
            NW, padx=(10, 0))
        ttk.Entry(self.controls, textvariable=self.
            smoments_array_coupler).grid(row=3, column=4,
            columnspan=2, sticky=tk.SE+tk.NW, padx=(10, 0))
        ttk.Checkbutton(self.controls, text="Mass␣Center",
            variable=self.bmass_center, onvalue=True, offvalue=
            False).grid(row=0, column=6, sticky=tk.SE+tk.NW, padx
            =(20, 0))
        ttk.Checkbutton(self.controls, text="Report", variable=
            self.breport, onvalue=True, offvalue=False).grid(row=1,
             column=6, sticky=tk.SE+tk.NW, padx=(20, 0))


def solve(self, *args):
    if self.select.current() == -1:
        return
    try:
```

```
machine = self.machines[self.select.current()]
self.graphics.clear()
input_rad = self.angle_rotate.get()*pi/180
speed = float(self.sangular_speed.get())
acceleration = float(self.sangular_acceleration.get())
inversions = self.check_inversion_array()

moments_ = self.smoments_array.get()
if moments_.strip() == '':
    moments = []
else:
    moments = [float(o) for o in moments_.split(',')]


moments_coupler = self.smoments_array_coupler.get()
if moments_coupler.strip() == '':
    moments__couplers = []
else:
    moments__couplers = [float(o) for o in
       moments_coupler.split(',')]

accelerations, forces, stresses, vonMises, locations,
   snapshot, order = machine.solution_kinetics(
   input_rad, speed, acceleration,
   external_moments_cranks=moments,
   external_moments_couplers=moments__couplers,
   pattern=inversions)
graphics.plot_machine(snapshot, mass_center=self.
   bmass_center.get(), max_stress=locations, axes=self
   .graphics.axis)
if self.breport.get():
    save_name = fd.asksaveasfilename(parent=self,
       title="Save screen", filetypes=(("html", "html"
       ),), initialdir="C:", defaultextension="html")
    if save_name:
        info = "<html><h1>Report</h1><hr>"
        include_crank = 0
        names = ["crank", "coupler", "output", "ground
           "]
        counter = 0
        for i in order[1:]:
            info+=f"<br><h3>Mechanism: {machine.
               mechanisms[i-1].name}<br><h3>
               Accelerations:</h3><br>"
            mech_index = counter*4

            for accel in accelerations[counter][
               include_crank:-1]:
```

```
                        info+=f"x: {accel[0]} <br> y: {accel
                            [1]} <br> angular: {accel[2]} <br>"

                    info+=f"<h3>Forces:</h3><br>"

                    for f in range(include_crank*2, 4):
                        info+=f"Connection {f+1}: \nx: {forces
                            [counter][f*2][0]} \ny: {forces[
                            counter][f*2+1][0]} \n"


                    info+=f"<h3>Stresses:</h3><br>"
                    for f in range(include_crank, 3):
                        info+=f"<br><br>Link {names[f]}: <br>
                            shear: {stresses[counter*2+f][0]} <
                            br>normal: {stresses[counter*2+f
                            ][1]} <br>moment stres: {stresses[
                            counter*2+f][2]}"
                        info+=f"<br>von Mises: {vonMises[
                            counter*2+f]} <br>"

                    include_crank = 1
                    counter+=1

                with open(save_name, "w") as ff:
                    ff.write(info)



            self.graphics.render()
        except ValueError:
            msg.showerror(parent=self, title="Error", message="
                Possible errors: acceleration or speed not a number
                or inversion array not correct")
        except Exception as e:
            msg.showerror(parent=self, title="Error", message=str(
                e))


    def check_inversion_array(self):
        inp = self.sinversion_array.get()
        if inp == "0" or inp == "1":
            return int(inp)
        m = re.fullmatch("[01]+", inp)
        if m:
            return [int(i) for i in m.group()]
        raise ValueError("Inversion array not valid must be a
            series of 0s and 1s or a single 0 or 1")
```

```
    def machines_available(self):
        if self.machines:
            self.select["values"] = [i.name for i in self.machines
                if i]



class GUI(tk.Tk):
    def __init__(self, curves:List[Curve]=[], links:List[Link]=[],
        mechanisms:List[Mechanism|SliderCrank]=[], machines:List[
       Machine]=[], icon=""):
        super().__init__()
        if icon:
            self.iconbitmap(icon)
        self.notebook = ttk.Notebook(self)
        self.curves = curves
        self.links = links
        self.mechanisms = mechanisms
        self.machines = machines
        self.geometry(f"+{self.winfo_screenwidth()//6}+0")
        self.resizable(False, False)
        self.title("KASM")

        self.pages = {
            "Curves": UICurve(self.notebook, self.curves),
            "Links":  UILink(self.notebook, self.curves, self.
               links),
            "Mechanisms": UIMechanism(self.notebook, self.links,
               self.mechanisms),
            "Machines": UIMachine(self.notebook, self.links, self.
               mechanisms, self.machines),
            "Stresses":UIStress(self.notebook, self.machines)
        }

        for key, value in self.pages.items():
            self.notebook.add(value, text=key, underline=0, sticky
               =tk.NE + tk.SW)

        self.notebook.bind("<<NotebookTabChanged>>", self.
           change_page)
        self.notebook.enable_traversal()
        self.notebook.pack()


    def change_page(self, *args):
        self.pages["Links"].curves_available()
```

```
        self.pages["Mechanisms"].links_available()
        self.pages["Machines"].mechanisms_available()
        self.pages["Machines"].links_available()
        self.pages["Stresses"].machines_available()



if __name__ == "__main__":
    import examples
    compresor = examples.build_compresor(3)
    machine = examples.build_machine()
    vline = examples.build_vline()
    power_comp = examples.build_double_crank(5)
    gui = GUI(links=compresor.mechanisms[0].links[:]+machine.
        mechanisms[0].links[:], mechanisms=compresor.mechanisms[:]+
        machine.mechanisms[:], machines=[compresor, machine, vline,
         power_comp], icon="Icon/pistons.ico")
    gui.mainloop()
```

12.5. Utilidades

```
'Fernando Lavarreda
'Export Inventor sketch to csv, by diffault pick the first sketch
Sub Main()
    Dim partDoc As PartDocument
    partDoc = ThisApplication.ActiveDocument
        Dim oFileDlg As FileDialog
    Call ThisApplication.CreateFileDialog(oFileDlg)
    oFileDlg.Filter = "CSV (*.csv;*.txt)|*.csv;*.txt|All Files
        (*.*)|*.*"
        oFileDlg.DialogTitle = "Save Info"
    oFileDlg.InitialDirectory = "C:\Documents"
        oFileDlg.CancelError = True
        On Error Resume Next
        oFileDlg.ShowSave
        If Err.Number
                Exit Sub
        End If
        filePath = oFileDlg.FileName
        Dim sketch As Sketch
    input_ = InputBox("Select a sketch to process:", "Sketch", "1
        ")
        sketch_num = 1
        If IsNumeric(input_)
                sketch_num = CInt(input_)
                If Not sketch_num <= partDoc.ComponentDefinition.
                    Sketches.Count
```

```
                sketch_num = 1
            End If
            If sketch_num <= 0
                sketch_num = partDoc.ComponentDefinition.
                    Sketches.Count
            End If
    End If
    save_ = Applitcation
sketch = partDoc.ComponentDefinition.Sketches.Item(sketch_num)

Dim coordinateData As String
coordinateData = "X, Y" & vbCrLf

Dim entity As SketchEntity
For Each entity In sketch.SketchEntities
    If TypeOf entity Is SketchLine Then
                coordinateData = coordinateData & "New
                    Line" & vbCrLf
        Dim line As SketchLine
        line = entity
        coordinateData = coordinateData & line.
            StartSketchPoint.Geometry.X & ", " & line.
            StartSketchPoint.Geometry.Y & vbCrLf
        coordinateData = coordinateData & line.EndSketchPoint.
            Geometry.X & ", " & line.EndSketchPoint.Geometry.Y
            & vbCrLf
    ElseIf TypeOf entity Is SketchArc Then
                coordinateData = coordinateData & "New Arc
                    " & vbCrLf
                Dim arc As SketchArc
        arc = entity
                startPoint = arc.StartSketchPoint.Geometry
            endPoint = arc.EndSketchPoint.Geometry
                centerPoint = arc.CenterSketchPoint.
                    Geometry
            radius = arc.Radius

                totalAngle = arc.SweepAngle
                differential = totalAngle / 1000
                For angle = arc.StartAngle + differential
                    To arc.StartAngle + totalAngle Step
                    differential
                        x = centerPoint.X + radius * Math.
                            Cos(angle)
                y = centerPoint.Y + radius * Math.Sin(
                    angle)
                        coordinateData = coordinateData &
                            x & ", " & y & vbCrLf
```

```
                Next
            coordinateData = coordinateData & endPoint.X & ",␣
                " & endPoint.Y & vbCrLf
            ElseIf TypeOf entity Is SketchCircle Then
                coordinateData = coordinateData & "New␣
                    Circle" & vbCrLf
                Dim circle As SketchCircle
        circle = entity
                radius = circle.Radius
                center = circle.CenterSketchPoint.Geometry
                stepSize = 0.01
                For angle = 0 To 2*PI Step stepSize
                        x = center.X + radius * Math.Cos(
                            angle)
                y = center.Y + radius * Math.Sin(angle)
                        coordinateData = coordinateData &
                            x & ",␣" & y & vbCrLf
                Next
            ElseIf TypeOf entity Is SketchSpline
                coordinateData = coordinateData & "New␣
                    Spline" & vbCrLf
                Dim spline As SketchSpline
                spline = entity
                stepSize = 0.01

                Dim evaluator As Curve2dEvaluator
            evaluator = spline.Geometry.Evaluator

                Dim endparam As Integer
                endparam = 1

                points = Int(endparam/stepSize)
                Dim arr((points+1)*2), tparam(points+1) As
                    Double

                counter = 0
                For start = 0 To points Step 1
                        tparam(counter) = start/points
                        counter = counter + 1
                Next

            evaluator.GetPointAtParam(tparam, arr)
            For count = 0 To points Step 1
                        coordinateData = coordinateData &
                            arr(count*2) & ",␣" & arr(count
                            *2+1) & vbCrLf
                Next
    ElseIf TypeOf entity Is SketchPoint
```

```
                        Dim point As SketchPoint
                        point = entity
                        If point.AttachedEntities.Count
                                Continue For
                        End If
                        coordinateData = coordinateData & "Point"
                           & vbCrLf
                        coords = point.Geometry
                        coordinateData = coordinateData & coords.X
                            & ", " & coords.Y & vbCrLf
                End If
    Next

    output = System.IO.File.CreateText(filePath)
    output.Write(coordinateData)
    output.Close()
End Sub
```

12.6. Ejemplos

```
#Fernando Lavarreda Examples

if __name__ == "__main__":
    import os
    import sys
    parent = os.path.abspath(os.path.join(os.path.dirname(__file__
        ), ".."))
    sys.path.append(parent)

from mechanisms import geometry as gm


def build_machine():
    c1 = gm.Curve(gm.Vector(0, 0), [gm.Vector(x/20, (x/20)**2) for
         x in range(11)], function=gm.Function(start=0, end=0.5,
        process=lambda x: x**2))
    c2 = gm.Curve(gm.Vector(0, 0), [gm.Vector(x/20, (x/20-1)**2)
        for x in range(10, 21)], function=gm.Function(start=0.5,
        end=1, process=lambda x: (x-1)**2))
    c3 = gm.Curve(gm.Vector(0, 0), [gm.Vector(0, 0), gm.Vector(1,
        0)], function=gm.Function(start=0, end=1, process=lambda x:
         0))

    hc1 = gm.Curve(gm.Vector(0, 0), [gm.Vector(x/40, 1+(0.25-(x
        /40-0.5)**2)**0.5) for x in range(41)], function=gm.
        Function(start=0, end=1, process=lambda x: 1+((0.25-(x-0.5)
        **2))**0.5))
```

```
hc2 = gm.Curve(gm.Vector(0, 0), [gm.Vector(x/40, -1-(0.25-(x
    /40-0.5)**2)**0.5) for x in range(41)], function=gm.
    Function(start=0, end=1, process=lambda x: -1-((0.25-(x
    -0.5)**2))**0.5))
jcurve = gm.Curve(gm.Vector(0, 0), [gm.Vector(0, 1), gm.Vector
    (0, -1)])
jcurve2 = gm.Curve(gm.Vector(0, 0), [gm.Vector(1, 1), gm.
    Vector(1, -1)])


bc1 = gm.Curve(gm.Vector(0, 0), [gm.Vector(x/20, -(x/20)**2)
    for x in range(21)], function=gm.Function(start=0, end=1,
    process=lambda x: -x**2))
bc2 = gm.Curve(gm.Vector(0, 0), [gm.Vector(x/20, -(x/20-2)**2)
     for x in range(20, 41)], function=gm.Function(start=1, end
    =2, process=lambda x: -(x-2)**2))
bc3 = gm.Curve(gm.Vector(0, 0), [gm.Vector(0, 0), gm.Vector(2,
     0)], function=gm.Function(start=0, end=2, process=lambda x
    : 0))


link = gm.Link(gm.Vector(0, 0), [gm.Vector(0, 0), gm.Vector(1,
     0)], [c1, c2, c3], thickness=0.1, name="crank machine")
link2 = gm.Link(gm.Vector(0, 0), [gm.Vector(0.5, 1.25), gm.
    Vector(0.5, -1.25)], [hc1.copy(), hc2.copy(), jcurve.copy()
    , jcurve2.copy()], thickness=0.1, name="coupler machine")
link3 = gm.Link(gm.Vector(0, 0), [gm.Vector(0.5, 1.25), gm.
    Vector(0.5, -1.25)], [hc1, hc2, jcurve, jcurve2], thickness
    =0.1, name="output machine")
link4 = gm.Link(gm.Vector(0, 0), [gm.Vector(0, 0), gm.Vector
    (2, 0)], [bc1, bc2, bc3], thickness=0.1, name="ground
    machine")

#Centroid calculations
link.set_lims([0, 1], 1)
link.set_lims([2,], 0)

link2.set_lims([0,], 1)
link2.set_lims([1,], 0)

link3.set_lims([0,], 1)
link3.set_lims([1,], 0)

link4.set_lims([2,], 1)
link4.set_lims([0,1], 0)

mech = gm.Mechanism(gm.Vector(0, 0), -0.2*gm.pi, [link, link2,
     link3, link4], ((0, 1), (0, 1), (0, 1), (0, 1)),
```

```
        stress_analysis=True, dx=1e-3, density=1)

    mech2 = mech.copy()
    mech2.rotate(0.2*gm.pi)
    mech3 = mech.copy()
    mech3.rotate(0.2*gm.pi)
    mech.name = "machine 1"
    mech2.name = "machine 2"
    mech3.name = "machine 3"
    machine = gm.Machine([mech, mech2, mech3], power_graph=[[1],
        [2], [3], []], name="Machine", auto_adjust=True)
    return machine


def build_compresor(pistons:int):
    gg = gm.Curve(gm.Vector(0, 0), [gm.Vector(0,0,),])
    ground = gm.Link(gm.Vector(0, 0), [gm.Vector(0, 0)], [gg,],
        0.0, name="ground compressor")
    half_circle = gm.Curve(gm.Vector(0, 0), [gm.Vector(x/20, (4-(x
        /20)**2)**0.5) for x in range(-40, 41)])
    half_circle.rotate(gm.pi/2)
    half_circle2 = gm.Curve(gm.Vector(0, 0), [gm.Vector(x/80,
        (0.5-(x/80)**2)**0.5) for x in range(-36, 37)])
    half_circle2.rotate(-gm.pi/2)
    half_circle2.translate(4, 0)
    line_down = gm.Curve(gm.Vector(0, 0), [gm.Vector(0, 2), gm.
        Vector(4.5, 0.45)])
    line_up = gm.Curve(gm.Vector(0, 0), [gm.Vector(0, -2), gm.
        Vector(4.5, -0.45)])
    crank = gm.Link(gm.Vector(0, 0), [gm.Vector(0, 0), gm.Vector
        (4, 0)], [half_circle, line_down, line_up, half_circle2],
        0.0, name="crank compressor")

    half_circle3 = gm.Curve(gm.Vector(0, 0), [gm.Vector(x/80,
        (0.25-(x/80)**2)**0.5) for x in range(-40, 41)])
    half_circle3.rotate(gm.pi/2)
    line_down2 = gm.Curve(gm.Vector(0, 0), [gm.Vector(0, 0.5), gm.
        Vector(12, 0)])
    line_up2 = gm.Curve(gm.Vector(0, 0), [gm.Vector(0, -0.5), gm.
        Vector(12, 0)])
    coupler = gm.Link(gm.Vector(0, 0), [gm.Vector(0, 0), gm.Vector
        (12, 0)], [half_circle3, line_down2, line_up2], 0.0, name="
        coupler compressor")

    side1 = gm.Curve(gm.Vector(0, 0), [gm.Vector(0, 0), gm.Vector
        (1, 0)])
    side2 = gm.Curve(gm.Vector(0, 0), [gm.Vector(0, 0), gm.Vector
        (0, 2)])
```

```
    side3 = gm.Curve(gm.Vector(0, 0), [gm.Vector(0, 2), gm.Vector
        (1, 2)])
    side4 = gm.Curve(gm.Vector(0, 0), [gm.Vector(1, 0), gm.Vector
        (1, 2)])
    slider = gm.Link(gm.Vector(0, 0), [gm.Vector(0.5, 1)], [side1,
         side2, side3, side4], 0.0, name="compressor slider")
    piston_1 = gm.SliderCrank(gm.Vector(0, 0), 0, [crank, coupler,
         slider, ground], ((0, 1), (0, 1), (0,), (0,)))

    if pistons%2:
        above_x_axis = (pistons+1)//2
        phase_dif = (gm.pi-(above_x_axis-1)*2*gm.pi/pistons)/2
    else:
        phase_dif = 0

    piston_1.rotate(phase_dif)
    distance = 2*gm.pi/pistons
    npistons = []
    for i in range(pistons):
        p = piston_1.copy()
        p.name = "piston "+str(i+1)
        p.rotate(distance*i)
        npistons.append(p)
    empty = [[] for i in range(pistons)]
    compresor = gm.Machine(npistons, power_graph=[[i+1 for i in
        range(pistons)], *empty], name=f"Compresor {pistons}
        pistons")
    return compresor


def build_vline():
    gg = gm.Curve(gm.Vector(0, 0), [gm.Vector(0,0,),],)
    ground = gm.Link(gm.Vector(0, 0), [gm.Vector(0, 0)], [gg,],
        0.0)
    half_circle = gm.Curve(gm.Vector(0, 0), [gm.Vector(x/20, (4-(x
        /20)**2)**0.5) for x in range(-40, 41)])
    half_circle.rotate(gm.pi/2)
    half_circle2 = gm.Curve(gm.Vector(0, 0), [gm.Vector(x/80,
        (0.5-(x/80)**2)**0.5) for x in range(-36, 37)])
    half_circle2.rotate(-gm.pi/2)
    half_circle2.translate(4, 0)
    line_down = gm.Curve(gm.Vector(0, 0), [gm.Vector(0, 2), gm.
        Vector(4.5, 0.45)])
    line_up = gm.Curve(gm.Vector(0, 0), [gm.Vector(0, -2), gm.
        Vector(4.5, -0.45)])
    crank = gm.Link(gm.Vector(0, 0), [gm.Vector(0, 0), gm.Vector
        (4, 0)], [half_circle, line_down, line_up, half_circle2],
        0.0)
```

```
    half_circle3 = gm.Curve(gm.Vector(0, 0), [gm.Vector(x/80,
        (0.25-(x/80)**2)**0.5) for x in range(-40, 41)])
    half_circle3.rotate(gm.pi/2)
    line_down2 = gm.Curve(gm.Vector(0, 0), [gm.Vector(0, 0.5), gm.
        Vector(12, 0)])
    line_up2 = gm.Curve(gm.Vector(0, 0), [gm.Vector(0, -0.5), gm.
        Vector(12, 0)])
    coupler = gm.Link(gm.Vector(0, 0), [gm.Vector(0, 0), gm.Vector
        (12, 0)], [half_circle3, line_down2, line_up2], 0.0)

    side1 = gm.Curve(gm.Vector(0, 0), [gm.Vector(0, 0), gm.Vector
        (1, 0)])
    side2 = gm.Curve(gm.Vector(0, 0), [gm.Vector(0, 0), gm.Vector
        (0, 2)])
    side3 = gm.Curve(gm.Vector(0, 0), [gm.Vector(0, 2), gm.Vector
        (1, 2)])
    side4 = gm.Curve(gm.Vector(0, 0), [gm.Vector(1, 0), gm.Vector
        (1, 2)])
    slider = gm.Link(gm.Vector(0, 0), [gm.Vector(0.5, 1)], [side1,
         side2, side3, side4], 0.0)
    piston_1 = gm.SliderCrank(gm.Vector(0, 0), 0, [crank, coupler,
         slider, ground], ((0, 1), (0, 1), (0,), (0,)), name="v-
        piston-1")
    piston_1.rotate(gm.pi/4)
    piston_2 = piston_1.copy()
    piston_2.rotate(gm.pi/2)
    piston_2.name = "v-piston-2"
    vline = gm.Machine([piston_1, piston_2], power_graph=[[1, 2],
        [], []], name=f"V-engine")
    return vline


def build_double_crank(pistons:int):
    compresor = build_compresor(pistons)
    half_circle = gm.Curve(gm.Vector(0, 0), [gm.Vector(x/20, (4-(x
        /20)**2)**0.5) for x in range(-40, 41)])
    half_circle.rotate(gm.pi/2)
    half_circle2 = gm.Curve(gm.Vector(0, 0), [gm.Vector(x/80,
        (0.5-(x/80)**2)**0.5) for x in range(-36, 37)])
    half_circle2.rotate(-gm.pi/2)
    half_circle2.translate(4, 0)
    line_down = gm.Curve(gm.Vector(0, 0), [gm.Vector(0, 2), gm.
        Vector(4.5, 0.45)])
    line_up = gm.Curve(gm.Vector(0, 0), [gm.Vector(0, -2), gm.
        Vector(4.5, -0.45)])
    crank = gm.Link(gm.Vector(0, 0), [gm.Vector(0, 0), gm.Vector
        (4, 0)], [half_circle, line_down, line_up, half_circle2],
```

```
        0.0)
    output = crank.copy()

    output.connections = output.connections[::-1]
    radii_1 = gm.Curve(gm.Vector(0, 0), [gm.Vector(x/10, -(6.25-(x
        /10)**2)**0.5) for x in range(-25, 26)])
    radii_2 = gm.Curve(gm.Vector(0, 0), [gm.Vector(x/10, -(12.25-(
        x/10)**2)**0.5) for x in range(-35, 36)])

    r_1 = gm.Curve(gm.Vector(0, 0), [gm.Vector(x/10, (0.25-(x/10)
        **2)**0.5) for x in range(-5, 6)])
    r_2 = r_1.copy()
    r_1.translate(-3, 0)
    r_2.translate(3, 0)
    coupler = gm.Link(gm.Vector(0, 0), [gm.Vector(-3, 0), gm.
        Vector(3, 0)], [radii_1, radii_2, r_1, r_2], 0.0)

    horizontal_1 = gm.Curve(gm.Vector(0, 0), [gm.Vector(0.5, 1),
        gm.Vector(1.5, 1)])
    horizontal_2 = gm.Curve(gm.Vector(0, 0), [gm.Vector(0, 0), gm.
        Vector(2, 0)])
    diagonal_1 = gm.Curve(gm.Vector(0, 0), [gm.Vector(0, 0), gm.
        Vector(0.5, 1)])
    diagonal_2 = gm.Curve(gm.Vector(0, 0), [gm.Vector(1.5, 1), gm.
        Vector(2, 0)])
    temp_link = gm.Link(gm.Vector(0, 0), [gm.Vector(1, 0.5), ], [
        diagonal_1, horizontal_1, diagonal_2, horizontal_2], 0.0)
    temp_link_2 = temp_link.copy()
    temp_link_2.translate(6, 0)
    ground = gm.Link(gm.Vector(0, 0), [temp_link.connections[0],
        temp_link_2.connections[0]], temp_link.curves[:]+
        temp_link_2.curves[:], 0.0)


    power_mech = gm.Mechanism(gm.Vector(0, 0), 0, [crank, coupler,
         output, ground], ((0, 1), (0, 1), (0, 1), (0, 1)), name="
        Externally powered compresor")
    empty = [[] for i in range(pistons)]
    powered = [i+1 for i in range(1, pistons+1)]
    powered_compresor = gm.Machine([power_mech, *compresor.
        mechanisms[:]], power_graph=[[1,], powered, *empty], name=f
        "Powered compresor", auto_adjust=True)
    return powered_compresor


if __name__ == "__main__":
    #Test forces, speeds
```

```
if '1' in sys.argv:
    print("Calculting centroid, mass and moment of inertia
        from 'machine' crank")
    c1 = gm.Curve(gm.Vector(0, 0), [gm.Vector(x/20, (x/20)**2)
         for x in range(11)], function=gm.Function(start=0, end
        =0.5, process=lambda x: x**2))
    c2 = gm.Curve(gm.Vector(0, 0), [gm.Vector(x/20, (x/20-1)
        **2) for x in range(10, 21)], function=gm.Function(
        start=0.5, end=1, process=lambda x: (x-1)**2))
    c3 = gm.Curve(gm.Vector(0, 0), [gm.Vector(0, 0), gm.Vector
        (1, 0)], function=gm.Function(start=0, end=1, process=
        lambda x: 0))
    link = gm.Link(gm.Vector(0, 0), [gm.Vector(0, 0), gm.
        Vector(1, 0)], [c1, c2, c3], 1)
    link.set_lims([0, 1], 1)
    link.set_lims([2,], 0)
    print(link.centroid(dx=1e-4, density=1e3))
    print("Mass:", link.mass)
    print("Moment inertia:", link.moment_inertia_centroid)

if '2' in sys.argv:
    print("Calculting centroid, mass and moment of inertia
        from disc")
    half_circle = gm.Curve(gm.Vector(0, 0), [gm.Vector(x/20, (
        x/20)**2) for x in range(40)], function=gm.Function(
        start=0, end=2, process=lambda x: (1-(x-1)**2)**0.5))
    half_circle2 = gm.Curve(gm.Vector(0, 0), [gm.Vector(x/20,
        (x/20-1)**2) for x in range(10, 21)], function=gm.
        Function(start=0, end=2, process=lambda x: -(1-(x-1)
        **2)**0.5))
    link = gm.Link(gm.Vector(0, 0), [gm.Vector(0, 0), gm.
        Vector(1, 0)], [half_circle, half_circle2], 1)
    link.set_lims([0,], 1)
    link.set_lims([1,], 0)
    print(link.centroid(dx=1e-4, density=1e3))
    print("Mass:", link.mass)
    print("Moment inertia:", link.moment_inertia_centroid)

if '3' in sys.argv:
    print("Calculating accelerations for 'machine'")
    machine = build_machine()
    accelerations, forces, stresses, vonMises, locations, _, _
         = machine.solution_kinetics(1.8*gm.pi, 0.2, 1, [1,],
        pattern=1)
    print(f"Accelerations: {accelerations}\nForces: {forces}\
        nStresses: {stresses}\nvon Mises: {vonMises}")
    print("Locations:")
    for i in locations:
```

```
print(i)
```