eman ta zabal zazu

Universidad del País Vasco | Euskal Herriko Unibertsitatea

BILBOKO INGENIARITZA ESKOLA
ESCUELA DE INGENIERÍA DE BILBAO

GRADO EN INGENIERÍA EN TECNOLOGÍA INDUSTRIAL

TRABAJO FIN DE GRADO

RESOLUCIÓN DEL PROBLEMA DE POSICIÓN DE MECANISMOS ULTRAFLEXIBLES PLANOS DE CADENA CERRADA

Alumno/Alumna: Pérez Gámiz, Javier

Director/Directora (1): Altuzarra Maestre, Óscar

Curso: 2017-2018

Fecha: Bilbao, 28-06-2018

ABSTRACT

This document describes a research project based on planar parallel continuum mechanisms analysis.

The objective is to deepen the knowledge of this type of mechanisms that have great advantage over conventional rigid-rods mechanisms, and therefore, they are being increasingly developed and have great potential for industrial applications. The purpose is to study the theorical behavior of this type of mechanisms and to solve both the direct and inverse kinematic problems for 2 and 3 degrees of freedom mechanisms through the use of Matlab software, which allows great flexibility when considering different working situations. Obtained results will be analysed in detail so that conclusions will be taken.

RESUMEN

Este documento recoge un trabajo de investigación basado en el análisis de mecanismos ultraflexibles planos de cadena cerrada.

El objetivo es profundizar en el conocimiento de este tipo de mecanismos, que presentan grandes ventajas sobre los mecanismos más convencionales de barra rígidas, y por lo tanto, están siendo objeto de un desarrollo creciente por su gran potencial en aplicaciones industriales. Se pretende estudiar el comportamiento teórico de este tipo de mecanismos y resolver tanto los problemas cinemáticos directos como inversos de mecanismos de 2 y 3 grados de libertad, mediante el uso del software Matlab, que permite una gran versatilidad a la hora de estudiar diferentes situaciones de trabajo. De esta manera, se puede llevar a cabo un análisis exhaustivo que permita obtener conclusiones.

LABURPENA

Dokumentu honek ultramalguak, lauak eta kate itxikoak diren mekanismoen analisia den ikerketa-lana jasotzen du.

Hain zuzen, mekanismo mota horien ezagueran sakontzea du helburu. Mekanismo horiek, barrako mekanismo konbentzionalekin alderatuz, abantaila handiak dituzte. Beraz, mekanismo horiek garapen handia izan dute, industrian aplikatzearren. Lanak mekanismo horien jarduera teorikoa aztertzea du helburu, baita arazo zinematiko zuzena eta alderantzizkoa ebaztea ere, bi eta hiru askatasun graduko mekanismoetan. Horretarako, Matlab software-a erabiliko da hainbat egoera aztertzeko aproposa da eta. Horrenbestez, analisi sakonaburutuko da, hainbat ondorio lortzearren.

PALABRAS CLAVE*: mecanismos planos ultraflexibles de cadena cerrada, manipuladores planos, problemas cinemáticos, análisis cinemático, Matlab.*

ÍNDICE

1. INTRODUCCIÓN ... 7
2. CONTEXTO ... 8
3. OBJETIVOS Y ALCANCE ... 9
4. BENEFICIOS ... 12
5. ESTADO DEL ARTE ... 13
5.1. DEFORMACIÓN DE UNA BARRA EN EL ESPACIO. MODELO DE COSSERAT ... 13
5.2. PARTICULARIZACIÓN PARA EL ANÁLISIS DE MANIPULADORES PARALELOS PLANOS ... 20
6. ANÁLISIS DE ALTERNATIVAS ... 25
7. ANÁLISIS DE RIESGOS ... 26
8. DESCRIPCIÓN DE LA PROPUESTA ... 27
9. DESCRIPCIÓN DE TAREAS, PROCEDIMIENTOS Y PLANIFICACIÓN ... 28
10. DIAGRAMA DE GANTT ... 30
11. CÁLCULOS Y ALGORITMOS ... 31
11.1. 3PRR ... 31
11.1.1. PROBLEMA INVERSO ... 32
11.1.2. PROBLEMA DIRECTO ... 40
11.1.3. VALIDACIÓN DE LOS CÁLCULOS ... 44
11.2. GENERALIDADES DEL CÁLCULO DE BARRAS FLEXIBLES ... 46
11.3. 2RFR ... 49
11.3.1. INTEGRACIÓN NUMÉRICA ... 49
11.3.2. INTEGRALES ELÍPTICAS. PROBLEMA DIRECTO ... 56
11.3.3. VALIDACIÓN DE LOS CÁLCULOS ... 58
11.4. 3PFR ... 60
11.4.1. INTEGRACIÓN NUMÉRICA ... 60
11.4.2. INTEGRALES ELÍPTICAS. PROBLEMA DIRECTO ... 64
11.4.3. VALIDACIÓN DE LOS RESULTADOS ... 66
12. DESCRIPCIÓN DE LOS RESULTADOS ... 69
13. EJEMPLOS DE APLICACIÓN ... 71
13.1. 3PRR ... 71
13.1.1. PROBLEMA INVERSO ... 71
13.1.2. PROBLEMA DIRECTO ... 71
13.2. 2RFR ... 72
13.2.1. INTEGRALES ELÍPTICAS. PROBLEMA DIRECTO ... 72
13.3. 3PFR ... 73

13.3.1. INTEGRALES ELÍPTICAS. PROBLEMA DIRECTO 73
14. ASPECTOS ECONÓMICOS. PRESUPUESTO 74
15. CONCLUSIONES 75
16. BIBLIOGRAFÍA 76
ANEXO I 77
1. DESARROLLO MATEMÁTICO DEL PROBLEMA DIRECTO DEL MECANISMO 3PRR 77
ANEXO II 84
1. CÓDIGO DEL PROGRAMA DE BARRA SIMPLEMENTE EMPOTRADA 84
1.1. INTEGRACIÓN NUMÉRICA. PROBLEMA INVERSO 84
1.2. INTEGRACIÓN NUMÉRICA. FUNCIONES ASOCIADAS 86
2. CÓDIGO DE LOS PROGRAMAS DEL MECANISMO 3PRR 90
2.1. PROBLEMA INVERSO 3PRR 90
2.2. PROBLEMA DIRECTO 3PRR 94
2.3. FUNCIONES ASOCIADAS 103
3. CÓDIGO DE LOS PROGRAMAS DEL MECANISMO 2RFR 107
3.1. INTEGRACIÓN NUMÉRICA. PROBLEMA INVERSO 107
3.2. INTEGRACIÓN NUMÉRICA. PROBLEMA DIRECTO 117
3.3. INTEGRACIÓN NUMÉRICA. FUNCIONES ASOCIADAS 126
3.4. INTEGRALES ELÍPTICAS. PROBLEMA DIRECTO 153
3.5. INTEGRALES ELÍPTICAS. FUNCIONES ASOCIADAS 161
4. CÓDIGO DE LOS PROGRAMAS DEL MECANISMO 3PFR 177
4.1. INTEGRACIÓN NUMÉRICA. PROBLEMA INVERSO 177
4.2. INTEGRACIÓN NUMÉRICA. PROBLEMA DIRECTO 186
4.3. INTEGRACIÓN NUMÉRICA. FUNCIONES ASOCIADAS 190
4.4. INTEGRALES ELÍPTICAS. PROBLEMA DIRECTO 206
4.5. INTEGRALES ELÍPTICAS. FUNCIONES ASOCIADAS 213

LISTA DE ILUSTRACIONES

Ilustración 1. Mecanismo 3PRR 9
Ilustración 2. Mecanismo 2RFR 10
Ilustración 3. Mecanismo 3PFR 10
Ilustración 4. Vector posición de un punto cualquiera de la barra 13
Ilustración 5. Porción diferencial de barra 15
Ilustración 6. Sistemas de referencia en una barra plana 18
Ilustración 7. Barra plana deformada por una carga en su extremo 20
Ilustración 8. Rango de valores existentes de k y k_{rel} para cada ψ 23
Ilustración 9. Superficie de relación entre k_{rel}, ψ y R 24
Ilustración 10. Relación entre k_{rel} y R para una ψ concreta 24
Ilustración 11. Mecanismo 3PRR 31
Ilustración 12. Cálculo gráfico de la barra 1 33
Ilustración 13. Solución 1 de la barra 1 34
Ilustración 14. Solución 2 de la barra 2 34
Ilustración 15. Cálculo gráfico de la barra 2 35
Ilustración 16. Solución 1 de la barra 2 36
Ilustración 17. Solución 2 de la barra 2 36
Ilustración 18. Cálculo gráfico de la barra 3 37
Ilustración 19. Solución 1 de la barra 3 38
Ilustración 20. Solución 2 de la barra 3 38
Ilustración 21. Ejemplo de representación gráfica del espacio de trabajo 40
Ilustración 22. Lazo 1 41
Ilustración 23. Lazo 2 42
Ilustración 24. Lazo 3 43
Ilustración 25. Ejemplo de posición fácilmente resoluble 45
Ilustración 26. Mecanismo 2RFR 49
Ilustración 27. Representación de la posición angular de los motores en función de (x,y) 53
Ilustración 28. Determinante del jacobiano del problema inverso en función de (x,y) 53
Ilustración 29. Representación de las coordenadas de P en función de (θ_1,θ_2) 55
Ilustración 30. Determinante del jacobiano del problema directo en función de (θ_1,θ_2) 55
Ilustración 31. Determinante del jacobiano del problema directo en función de (x,y) 56
Ilustración 32. Sistemas de referencia locales en el mecanismo 2RFR 57
Ilustración 33. Mecanismo 2RFR simétrico 59
Ilustración 34. Mecanismo 3PFR 60
Ilustración 35. Representación de los valores de λ_1, λ_2 y λ_3 en función de (x,y) 63
Ilustración 36. Determinante del jacobiano del problema inverso en función de (x,y) 63
Ilustración 37. Sistemas de referencia locales en el mecanismo 3PFR 65
Ilustración 38. Mecanismo 3PFR simétrico 67
Ilustración 39. Soluciones del ejemplo del problema directo del 2RFR 72
Ilustración 40. Soluciones del ejemplo del problema directo del 3PFR 73
Ilustración 41. Representación de de la proporción de costes totales 74

LISTA DE TABLAS

Tabla 1. Valoración de riesgos 26
Tabla 2. Comparación de grados de discretización y tolerancias 68
Tabla 3. Ejemplo 1 de aplicación del problema inverso del mecanismo 3PRR 71
Tabla 4. Ejemplo 2 de aplicación del problema inverso del mecanismo 3PRR 71
Tabla 5. Ejemplo 1 de aplicación del problema directo del mecanismo 3PRR 71
Tabla 6. Ejemplo 2 de aplicación del problema directo del mecanismo 3PRR 72
Tabla 7. Soluciones del ejemplo del problema directo del 2RFR 72
Tabla 8. Soluciones del ejemplo del problema directo del 3PFR 73
Tabla 9. Desgrane de costes 74

1. INTRODUCCIÓN

Este documento recoge el planteamiento y desarrollo de un trabajo de investigación que consiste en el estudio de mecanismos ultraflexibles planos de cadena cerrada y la resolución de sus problemas de posición.

Se analiza el comportamiento teórico de estos mecanismos, de 2 y 3 grados de libertad, y se programa la resolución de sus problemas cinemáticos de posición. Para ello, se emplea la herramienta informática Matlab. Se trata de un software con una gran potencia de cálculo que permite una gran versatilidad a la hora de variar los datos de un mismo problema. De esta manera, se pueden estudiar diferentes posibilidades extraer conclusiones de los resultados obtenidos.

Se profundiza así en el conocimiento de un tipo de mecanismos que está en continuo desarrollo y al que se le prevé un gran futuro por sus ventajas con respecto a los mecanismos rígidos, más habituales.

2. CONTEXTO

Los mecanismos de cadena cinemática cerrada son aquellos donde cada elemento está conectado a dos o más elementos. En las últimas décadas y hasta la fecha, los mecanismos de cadena cerrada han tenido una gran relevancia en aplicaciones industriales y están ampliamente extendidos en tareas de manipulación o simuladores, entre otras.

Hasta ahora, la inmensa mayoría de estos manipuladores paralelos son de barras rígidas. Sin embargo, recientemente se está desarrollando la posibilidad de que las barras ya no sean rígidas, sino flexibles. En este caso, los mecanismos pasan a ser ultraflexibles y transmiten el movimiento y la fuerza a través de la deformación elástica de sus elementos.

Los mecanismos ultraflexibles presentan ventajas frente a los mecanismos de elementos rígidos: suponen un riesgo menor en el trabajo con personas, sus diseños son más sencillos y requieren menos piezas, lo que implica una menor pérdida de energía porque no dejan lugar a que haya fricción.

Debido a que la transmisión del movimiento depende de la deformación elástica de los elementos del mecanismo, toma una gran importancia la rigidez del sistema. Se trata de un parámetro que determina totalmente el comportamiento del mecanismo.

Es por todo eso que su utilización, aunque aún no está muy extendida, se está desarrollando cada vez más y supone un interesante objeto de estudio, ya que un mayor conocimiento de su comportamiento permitirá una mayor evolución y amplitud en sus aplicaciones.

3. OBJETIVOS Y ALCANCE

El proyecto de investigación que se desarrolla en este documento tiene como objetivo el estudio teórico del comportamiento de mecanismos ultraflexibles planos de cadena cerrada, a través de la resolución de sus problemas cinemáticos de posición. Se utiliza el software Matlab para programar el cálculo de estos problemas y poder analizar distintas situaciones. De esta forma, se puede llegar a realizar un estudio completo de las posiciones que forman el espacio de trabajo de los mecanismos.

El proyecto comienza con la preparación teórica, que consiste en analizar y entender los fundamentos que se van a aplicar a la resolución de los problemas de posición y que están detallados en el ESTADO DEL ARTE. Básicamente, se trata de una descripción detallada del modelo de barra de Cosserat.

En primer lugar, antes de entrar a analizar cualquier mecanismo flexible, y como primera toma de contacto para conocer el modo de proceder en este tipo de análisis, se lleva a cabo la resolución de los problemas de posición de un mecanismo de barras rígidas de 3 grados de libertad. Se trata del mecanismo 3PRR, que se puede ver en la Ilustración 1.

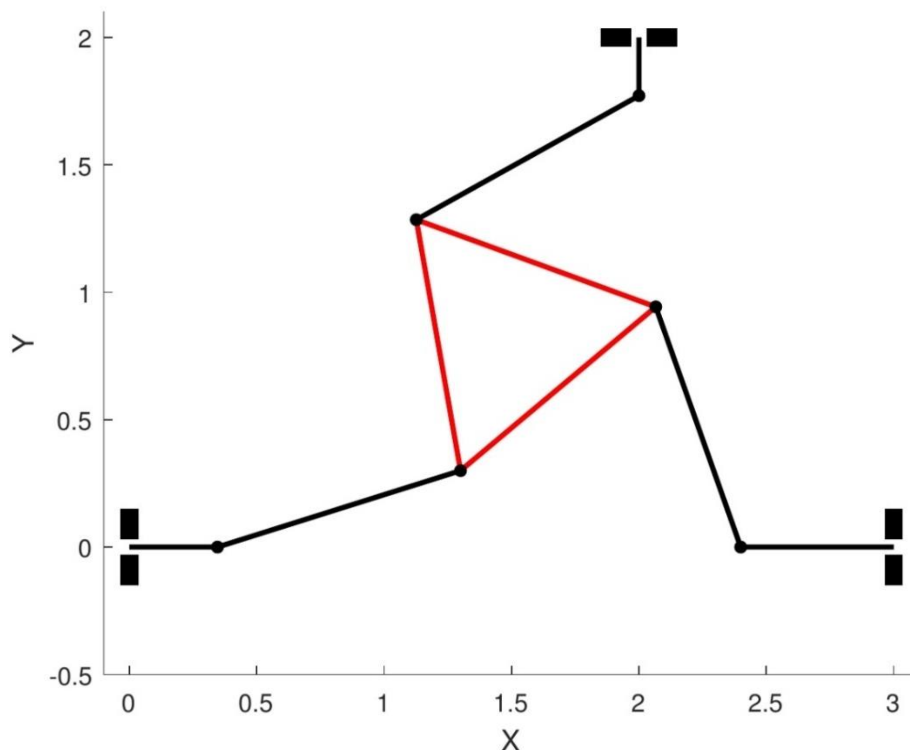


Ilustración 1. Mecanismo 3PRR

A partir de este análisis previo y teniendo en cuenta los pasos seguidos, se estudia el comportamiento de dos mecanismos ultraflexibles planos de cadena cerrada sometidos a cargas externas, resolviendo sus problemas directo e inverso. Los mecanismos en cuestión serán los siguientes:

- Mecanismo de 2 GdL (Grados de Libertad) formado por dos barras empotradas-articuladas. (Ilustración 2)
- Mecanismo de 3 GdL formado por tres barras empotradas-articuladas y un actuador triangular. (Ilustración 3)

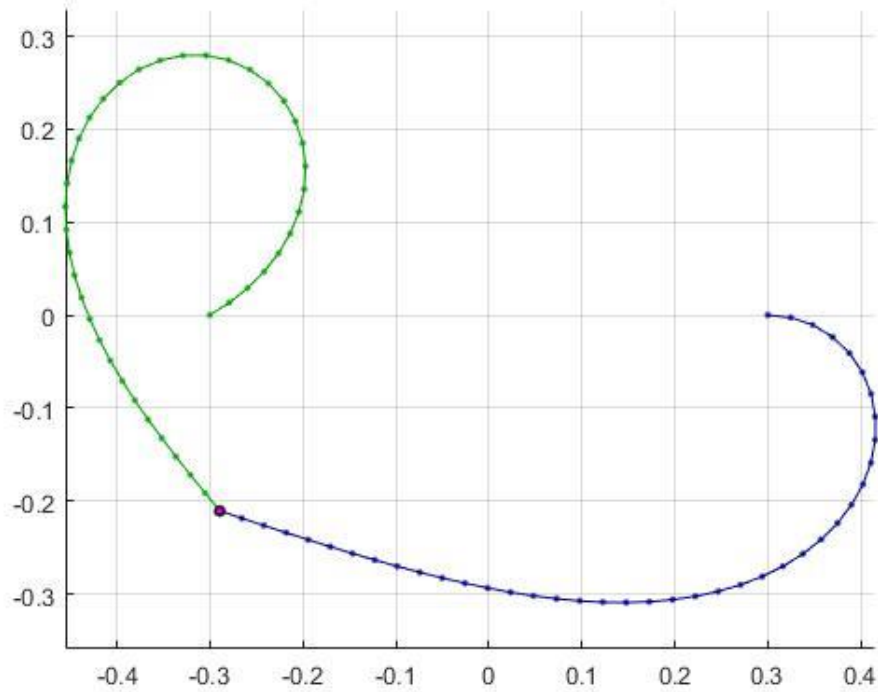


Ilustración 2. Mecanismo 2RFR

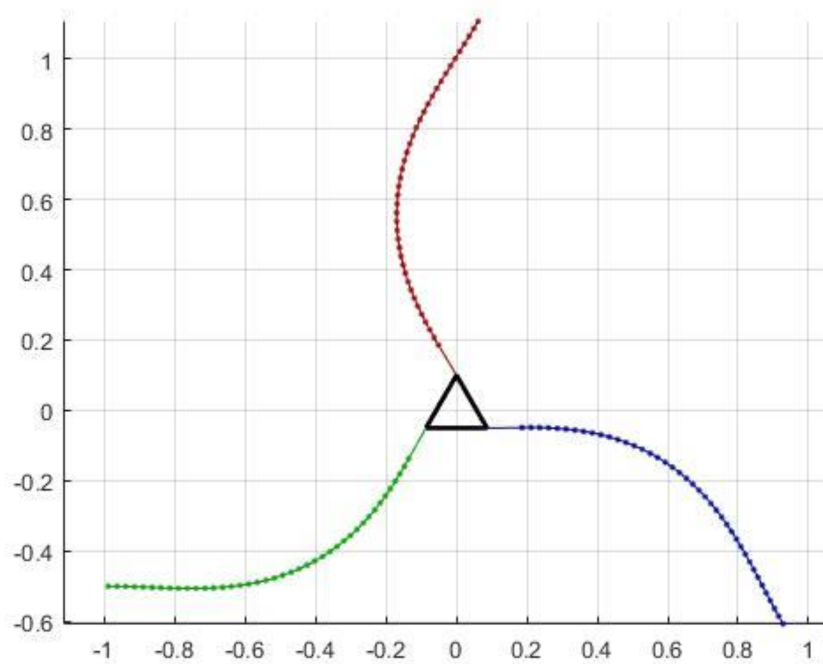


Ilustración 3. Mecanismo 3PFR

Es importante señalar que, en el caso de los dos mecanismos flexibles, la resolución se lleva a cabo de dos formas distintas, partiendo de los dos planteamientos teóricos que se describen en el apartado de ESTADO DEL ARTE.

4. BENEFICIOS

El desarrollo de este proyecto supone una profundización en el estudio y el conocimiento del comportamiento de mecanismos ultraflexibles planos de cadena cerrada. Hasta la fecha, la utilización de mecanismos de cadena cerrada en aplicaciones industriales está ampliamente extendida. Sin embargo, la mayoría de estos son de barras rígidas. En los últimos tiempos, el empleo de mecanismos ultraflexibles está en auge y en pleno desarrollo por las ventajas que presentan respecto a las barras rígidas.

Por un lado, los elementos flexibles permiten reducir el número de piezas en un mecanismo y, por lo tanto, el número de uniones entre ellas. Esto evita situaciones de deslizamiento y fricción entre superficies y permite prescindir del uso de lubricantes.

Por otro lado, las uniones rígidas conllevan necesariamente un juego que resta precisión a la respuesta del sistema. Las uniones flexibles eliminan este juego al estar previamente cargadas, y por lo tanto la respuesta del sistema es más rápida.

El diseño de los mecanismos ultraflexibles es más sencillo; son más ligeros, más baratos de fabricar y permiten construcciones mucho más pequeñas que los mecanismos de barras rígidas. En el caso de sistemas de una sola pieza, la dilatación térmica no supone ningún problema, puesto que el coeficiente de expansión es constante.

Además, el empleo de elementos flexibles tiene grandes ventajas en cuanto a seguridad en la interacción con personas. Es evidente que, en caso de accidente, resultan mucho menos agresivos que los elementos rígidos.

5. ESTADO DEL ARTE

5.1. DEFORMACIÓN DE UNA BARRA EN EL ESPACIO. MODELO DE COSSERAT

Para poder llevar a cabo el análisis de cualquier mecanismo de barras flexibles, es necesario conocer con detalle su comportamiento y las ecuaciones matemáticas que lo rigen. En este apartado, se describe el modelo de barra de Cosserat y una simplificación de la misma, el modelo de Kirchhoff, que se aplicará más adelante al cálculo de los problemas de posición de los mecanismos que se estudian en este proyecto.

Se define una barra en el espacio cuya forma está definida por una curva paramétrica $\boldsymbol{p}(s)$ que determina la posición del centroide de cada sección transversal de la barra, y una matriz ortonormal de rotación $\boldsymbol{R}(s)$ que determina su orientación. Las secciones se consideran rígidas, por lo que quedan de esta manera perfectamente definidas. Tanto $\boldsymbol{p}$ como $\boldsymbol{R}$ son funciones de s, que es la longitud de arco a lo largo de la barra. La orientación de cada sección es la misma que la del sistema de referencia (x, y, z) acoplado a ella, como se puede ver en la Ilustración 4.

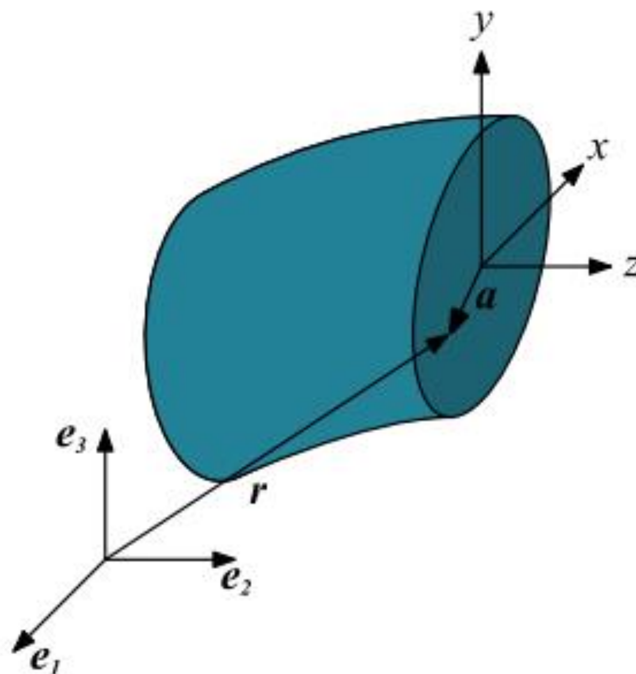


Ilustración 4. Vector posición de un punto cualquiera de la barra

La posición de cualquier punto de la barra, definida por el vector $\boldsymbol{r}$, se puede calcular a partir de $\boldsymbol{p}(s)$, $\boldsymbol{R}(s)$ y el vector $\boldsymbol{a}$ que define la posición del punto en el sistema de referencia local de la sección.

$$\boldsymbol{r}(s, \boldsymbol{a}) = \boldsymbol{R}(s)\boldsymbol{a} + \boldsymbol{p}(s) \tag{1}$$

La forma inicial de la barra, cuando no está cargada, está definida por $\boldsymbol{p}_o(s)$ y $\boldsymbol{R}_o(s)$. En principio, se podría colocar $\boldsymbol{R}_o(s)$ arbitrariamente, pero por convenio se decide que el eje z sea perpendicular a la sección y los ejes x, y coincidan con sus ejes principales de inercia.

Se considera las derivadas de $\boldsymbol{p}(s)$ y $\boldsymbol{R}(s)$ respecto del parámetro s. Teniendo en cuenta que $\boldsymbol{R}(s)$ es una matriz ortonormal y por tanto cumple que $\boldsymbol{R}(s)\boldsymbol{R}(s)^T = \boldsymbol{I}$, estas derivadas expresadas en el sistema de referencia local (x, y, z) son:

$$\boldsymbol{v}(s) = \boldsymbol{R}(s)^T \, \boldsymbol{p}'(s) \tag{2}$$

$$\boldsymbol{U}(s) = \boldsymbol{R}(s)^T \, \boldsymbol{R}'(s) \tag{3}$$

La matriz $\boldsymbol{U}(s)$ resultante es del tipo

$$\boldsymbol{U}(s) = \begin{bmatrix} 0 & -u_z(s) & u_y(s) \\ u_z(s) & 0 & -u_x(s) \\ -u_y(s) & u_x(s) & 0 \end{bmatrix} \tag{4}$$

Para facilitar los cálculos, a partir de ahora se utilizará un vector $\boldsymbol{u}(s)$ como expresión simplificada de $\boldsymbol{U}(s)$.

$$\boldsymbol{u}(s) = \begin{bmatrix} u_x(s) \\ u_y(s) \\ u_z(s) \end{bmatrix} \tag{5}$$

Tanto $\boldsymbol{u}(s)$ como $\boldsymbol{v}(s)$ tienen un significado físico. Llamando $\boldsymbol{u}_o(s)$ y $\boldsymbol{v}_o(s)$ a las magnitudes en una situación inicial, con la barra sin deformar, $\Delta\boldsymbol{v}(s) = \boldsymbol{v}(s) - \boldsymbol{v}_o(s)$ y $\Delta\boldsymbol{u}(s) = \boldsymbol{u}(s) - \boldsymbol{u}_o(s)$ describen la deformación de la barra respecto de su forma inicial. Δv_z representa la deformación longitudinal de la barra: será mayor que 1 cuando se alarga, menor que 1 cuando se comprime e igual a 1 si mantiene la misma longitud. Δv_x y Δv_y definen la deformación de la sección transversal. El vector $\boldsymbol{u}(s)$ describe la curvatura de la barra. Por tanto, $\Delta u_x(s)$ y $\Delta u_y(s)$ representan la flexión en los dos ejes principales de la sección; y $\Delta u_z(s)$, la torsión. Hay que tener en cuenta que el modelo de Cosserat que se está desarrollando es válido siempre que la sección transversal se mantenga constante.

El campo de desplazamientos que se ha definido mediante $\boldsymbol{u}(s)$ y $\boldsymbol{v}(s)$ genera un campo de tensiones. Además, la fuerza interna $\boldsymbol{n}(s)$ y el momento $\boldsymbol{m}(s)$ son consecuencia de ese campo de tensiones, por lo que pueden expresarse en función de $\boldsymbol{u}(s)$ y $\boldsymbol{v}(s)$. Aunque en general no tiene por qué ser así, se utiliza una ley de comportamiento lineal del material de la barra, válida para deformaciones pequeñas, que permite expresar la fuerza y el momento internos en el sistema de referencia local de la siguiente manera:

$$\boldsymbol{n}_{loc}(s) = \boldsymbol{K}_{SE}(s)\Delta\boldsymbol{v}(s) \tag{6}$$

$$\boldsymbol{m}_{loc}(s) = \boldsymbol{K}_{\boldsymbol{BT}}(s)\Delta\boldsymbol{u}(s) \tag{7}$$

donde

$$\boldsymbol{K}_{SE} = \begin{bmatrix} G\,A(s) & 0 & 0 \\ 0 & G\,A(s) & 0 \\ 0 & 0 & E\,A(s) \end{bmatrix} \tag{8}$$

$$\boldsymbol{K}_{BT} = \begin{bmatrix} E\, I_{xx}(s) & 0 & 0 \\ 0 & E\, I_{yy}(s) & 0 \\ 0 & 0 & G\, I_{zz}(s) \end{bmatrix} \tag{9}$$

$A(s)$ es el área de la sección transversal; I_{xx}, I_{yy}, I_{zz} son los momentos de inercia de dicha sección respecto de los ejes x, y, z; E es el módulo de Young del material y G es el módulo de elasticidad transversal.

Se puede expresar la fuerza interna y el momento en el sistema de referencia fijo simplemente multiplicando por la matriz de rotación $\boldsymbol{R}(s)$.

$$\boldsymbol{n}(s) = \boldsymbol{R}(s)\boldsymbol{K}_{SE}(s)\Delta\boldsymbol{v}(s) \tag{10}$$

$$\boldsymbol{m}(s) = \boldsymbol{R}(s)\boldsymbol{K}_{\boldsymbol{BT}}(s)\Delta\boldsymbol{u}(s) \tag{11}$$

De esta manera, se ha conseguido determinar cuáles son la fuerza y el momento internos de la barra en cada sección transversal definida por el parámetro s.

Ahora, se considera una porción infinitesimal de barra, limitada por dos secciones que están situadas en s y $s + ds$. Sobre estas secciones actúan las fuerzas y momentos correspondientes a los fragmentos de barra que quedan a ambos lados. Además, hay una fuerza $\boldsymbol{f}(s)$ y un momento $\boldsymbol{l}(s)$ distribuidos a lo largo de la superficie de la porción seleccionada. Se puede ver la distribución de cargas en la Ilustración 5.

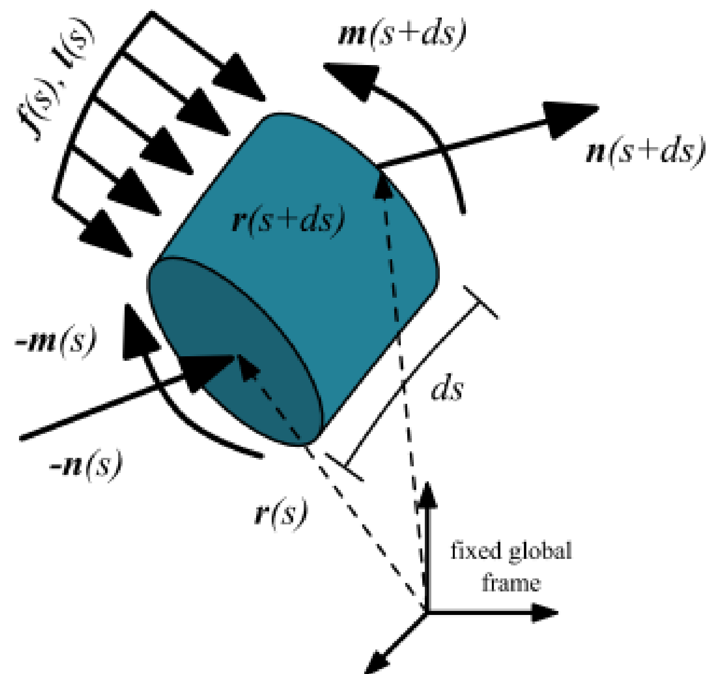


Ilustración 5. Porción diferencial de barra

El equilibrio de fuerzas en la porción de barra da como resultado la siguiente ecuación:

$$-\boldsymbol{n}(s) + \boldsymbol{n}(s + ds) + \boldsymbol{f}(s)ds = 0 \tag{12}$$

Desarrollando las funciones en $s + ds$ mediante series de Taylor hasta el elemento de primer orden y reordenando la expresión queda:

$$\frac{d\boldsymbol{n}(s)}{ds} + \boldsymbol{f}(s) = 0 \tag{13}$$

Del balance de momentos resulta la siguiente ecuación:

$$\begin{gathered}-\boldsymbol{m}(s) + \boldsymbol{m}(s+ds) + \boldsymbol{l}(s)ds + \boldsymbol{p}(s) \times \big(-\boldsymbol{n}(s)\big) + \\ +\boldsymbol{p}(s+ds) \times \boldsymbol{n}(s+ds) + \boldsymbol{p}(s+\mu ds) \times \boldsymbol{f}(s)ds = 0\end{gathered} \tag{14}$$

Donde $0 < \mu < 1$ denota la posición en la que se aplica la resultante de la fuerza $\boldsymbol{f}$.

Igual que antes, desarrollando las funciones en $s + ds$ mediante series de Taylor hasta el elemento de primer orden y reordenando, despreciando los diferenciales de segundo orden y teniendo en cuenta la igualdad obtenida en (67), resulta

$$\frac{d\boldsymbol{m}(s)}{ds} + \boldsymbol{l}(s) + \frac{d\boldsymbol{p}(s)}{ds} \times \boldsymbol{n} = 0 \tag{15}$$

Por lo tanto, el sistema de ecuaciones que describen la evolución de la fuerza interna **n**(s) y el momento **m**(s) a lo largo del modelo de barra de Cosserat queda así:

$$\begin{cases} \boldsymbol{n}'(s) + \boldsymbol{f}(s) = 0 \\ \boldsymbol{m}'(s) + \boldsymbol{p}'(s) \times \boldsymbol{n}(s) + \boldsymbol{l}(s) = 0 \end{cases} \tag{16}$$

Ahora, se trata de buscar una expresión que permita determinar la forma que toma la barra. Derivando respecto de s las ecuaciones (10) y (11), resulta:

$$\boldsymbol{n}' = \boldsymbol{R}\boldsymbol{K}_{SE}(\boldsymbol{v} - \boldsymbol{v}_o) + \boldsymbol{R}\boldsymbol{K}'_{SE}(\boldsymbol{v} - \boldsymbol{v}_o) + \boldsymbol{R}\boldsymbol{K}_{SE}\boldsymbol{v}' \tag{17}$$

$$\boldsymbol{m}' = \boldsymbol{R}\boldsymbol{K}_{BT}(\boldsymbol{u} - \boldsymbol{u}_o) + \boldsymbol{R}\boldsymbol{K}'_{BT}(\boldsymbol{u} - \boldsymbol{u}_o) + \boldsymbol{R}\boldsymbol{K}_{BT}\boldsymbol{u}' \tag{18}$$

Teniendo en cuenta las expresiones (2) y (3) e introduciendo (10) y (11) en el sistema (52), obtenemos el siguiente sistema de ecuaciones diferenciales:

$$\begin{cases} \boldsymbol{p}' = \boldsymbol{R}\boldsymbol{v} \\ \boldsymbol{R}' = \boldsymbol{R}\boldsymbol{U} \\ \boldsymbol{v}' = \boldsymbol{v}'_o - \mathbf{K}_{\mathrm{SE}}^{-1}((\boldsymbol{U}\boldsymbol{K}_{SE} + \boldsymbol{K}'_{SE})(\boldsymbol{v} - \boldsymbol{v}_o) + \boldsymbol{R}^T\boldsymbol{f}) \\ \boldsymbol{u}' = \boldsymbol{u}'_o - \boldsymbol{K}_{BT}^{-1}((\boldsymbol{U}\boldsymbol{K}_{BT} + \boldsymbol{K}'_{BT})(\boldsymbol{u} - \boldsymbol{u}_o) + \boldsymbol{V}\boldsymbol{K}_{SE}(\boldsymbol{v} - \boldsymbol{v}_o) + \boldsymbol{R}_T\boldsymbol{l}) \end{cases} \tag{19}$$

donde $\boldsymbol{V}$ es una matriz con los elementos del vector $\boldsymbol{v}$:

$$\boldsymbol{V}(s) = \begin{bmatrix} 0 & -v_z(s) & v_y(s) \\ v_z(s) & 0 & -v_x(s) \\ -v_y(s) & v_x(s) & 0 \end{bmatrix} \tag{20}$$

También se puede plantear el sistema (55) con $\boldsymbol{n}$ y $\boldsymbol{m}$ como variables a integrar:

$$\begin{cases} \boldsymbol{p}' = \boldsymbol{R}\boldsymbol{v} \\ \boldsymbol{R}' = \boldsymbol{R}\boldsymbol{U} \\ \boldsymbol{n}' = -\boldsymbol{f} \\ \boldsymbol{m}' = -\boldsymbol{p}' \times \boldsymbol{n} - \boldsymbol{l} \end{cases} \quad (21)$$

donde $\boldsymbol{v} = \boldsymbol{K}_{SE}^{-1}\boldsymbol{R}^T\boldsymbol{n} + \boldsymbol{v}_o$ y $\boldsymbol{u} = \boldsymbol{K}_{BT}^{-1}\boldsymbol{R}^T\boldsymbol{m} + \boldsymbol{u}_o$.

En el caso de la simplificación de Kirchhoff, se desprecian las dimensiones transversales de la barra y se considera que la forma que adopta la barra se debe solamente a la flexión y la torsión. Por tanto, $\boldsymbol{v} = \boldsymbol{v}_o = \boldsymbol{e}_3$ y el sistema (55) queda de la siguiente forma:

$$\begin{cases} \boldsymbol{p}' = \boldsymbol{R}\boldsymbol{e}_3 \\ \boldsymbol{R}' = \boldsymbol{R}\boldsymbol{U} \\ \boldsymbol{n}' = -\boldsymbol{f} \\ \boldsymbol{u}' = \boldsymbol{u}_o' - \boldsymbol{K}_{BT}^{-1}((\boldsymbol{U}\boldsymbol{K}_{BT} + \boldsymbol{K}_{BT}')(\boldsymbol{u} - \boldsymbol{u}_o) + \widehat{\boldsymbol{e}_3}\boldsymbol{K}_{SE}(\boldsymbol{v} - \boldsymbol{v}_o) + \boldsymbol{R}_T\boldsymbol{l}) \end{cases} \quad (22)$$

En este caso, se ha llamado $\widehat{\boldsymbol{e}_3}$ a una matriz que contiene los elementos del vector $\boldsymbol{e}_3$, de la misma forma que se relacionan las matrices $\boldsymbol{U}$ y $\boldsymbol{V}$ con los vectores $\boldsymbol{u}$ y $\boldsymbol{v}$.

Tomando $\boldsymbol{n}$ y $\boldsymbol{m}$ como variables a integrar en el modelo de Kirchhoff,

$$\begin{cases} \boldsymbol{p}' = \boldsymbol{R}\boldsymbol{e}_3 \\ \boldsymbol{R}' = \boldsymbol{R}\boldsymbol{U} \\ \boldsymbol{n}' = -\boldsymbol{f} \\ \boldsymbol{m}' = -\boldsymbol{p}' \times \boldsymbol{n} - \boldsymbol{l} \end{cases} \quad (23)$$

donde $\boldsymbol{u} = \boldsymbol{K}_{BT}^{-1}\boldsymbol{R}^T\boldsymbol{m} + \boldsymbol{u}_o$.

Hasta ahora, todas las ecuaciones que se han visto describen el comportamiento de una barra en el espacio. En el caso concreto de una barra que se deforma en un plano, se pueden introducir simplificaciones. La orientación de cada sección de la barra se puede definir ahora mediante un único ángulo $\theta(s)$, como se puede ver en la Ilustración 6.

La matriz de rotación en este caso es:

$$\boldsymbol{R}(s) = \begin{bmatrix} -\sin\theta & 0 & \cos\theta \\ \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \end{bmatrix} \quad (24)$$

Y su derivada respecto de la longitud de arco s:

$$\boldsymbol{R}'(s) = \begin{bmatrix} -\cos\theta & 0 & -\sin\theta \\ -\sin\theta & 0 & \cos\theta \\ 0 & 0 & 0 \end{bmatrix}\theta' \quad (25)$$

Según la definición de la matriz $\boldsymbol{U}$ en la ecuación (4),

$$\boldsymbol{U} = \boldsymbol{R}^T\boldsymbol{R}' = \begin{bmatrix} 0 & 0 & \theta' \\ 0 & 0 & 0 \\ -\theta' & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & u_y \\ 0 & 0 & 0 \\ -u_y & 0 & 0 \end{bmatrix} \quad (26)$$

$u_y = \theta'$ es el único componente del vector **u** no nulo, lo cual concuerda con el hecho de que no hay momento flector en el eje x ni torsión en z.

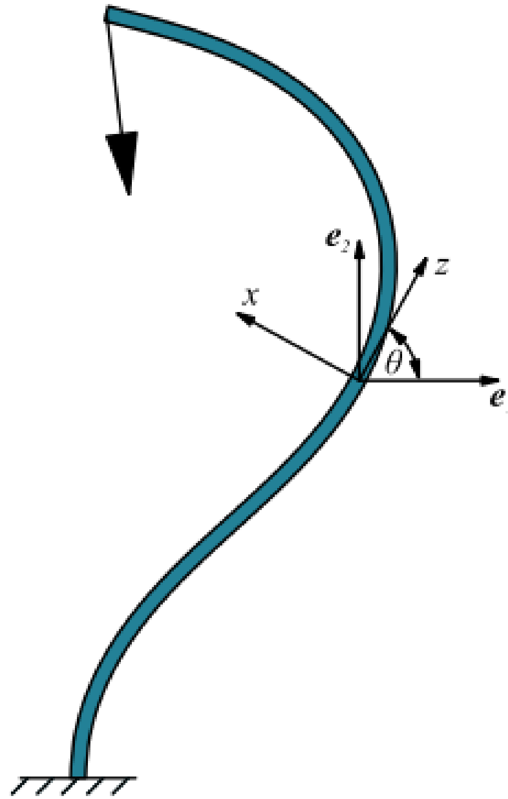


Ilustración 6. Sistemas de referencia en una barra plana

Introduciendo que $u_y = \theta'$ en el sistema (19) resulta

$$\begin{Bmatrix} p_x'(s) \\ p_y'(s) \\ \theta'(s) \\ v_x'(s) \\ v_z'(s) \\ u_y'(s) \end{Bmatrix} = \begin{Bmatrix} v_z \cos\theta - v_x \sin\theta \\ v_x \cos\theta + v_z \sin\theta \\ u_y \\ -\frac{1}{GA}[(GA)' + EAu_y(v_z - 1) + f_y \cos\theta - f_x \sin\theta] \\ -\frac{1}{EA}[(EA)'(v_z - 1) + GAu_y v_x + f_x \cos\theta + f_y \sin\theta] \\ u_{o,y}' - \frac{1}{EI}[(EI)'\left(u_y - u_{o,y}\right) - (E - G)Av_x v_z + l_z] \end{Bmatrix} \quad (27)$$

Lo mismo con el modelo de Kirchhoff:

$$\begin{Bmatrix} p_x'(s) \\ p_y'(s) \\ \theta'(s) \\ n_x'(s) \\ n_z'(s) \\ u_y'(s) \end{Bmatrix} = \begin{Bmatrix} \cos\theta \\ \sin\theta \\ u_y \\ -f_x \\ -f_y \\ u_{o,y}' - \frac{1}{EI}[(EI)'(u_y - u_{o,y}) + n_y\cos\theta - n_y\sin\theta + l_z] \end{Bmatrix} \quad (28)$$

Igual que se ha hecho antes, se pueden redefinir los sistemas (27) y (28) introduciendo $\boldsymbol{n}$ y $\boldsymbol{m}$ como variables a integrar. En el caso del modelo de Cosserat:

$$\begin{Bmatrix} p_x'(s) \\ p_y'(s) \\ \theta'(s) \\ n_x'(s) \\ n_y'(s) \\ m_z'(s) \end{Bmatrix} = \begin{Bmatrix} (\frac{C_1}{EA} + 1)\cos\theta - \frac{C_2}{GA}\sin\theta \\ (\frac{C_1}{EA} + 1)\cos\theta + \frac{C_2}{GA}\sin\theta \\ \frac{m_z}{EI} + u_{o,y} \\ -f_x \\ -f_y \\ -\left(C_2 + \left(\frac{1}{EA} - \frac{1}{GA}\right)C_1\right) - l_z \end{Bmatrix} \quad (29)$$

donde $C_1 = n_x\cos\theta + n_y\sin\theta$ y $C_2 = n_y\cos\theta - n_x\sin\theta$.

Y en el modelo de Kirchhoff:

$$\begin{Bmatrix} p_x'(s) \\ p_y'(s) \\ \theta'(s) \\ n_x'(s) \\ n_y'(s) \\ m_z'(s) \end{Bmatrix} = \begin{Bmatrix} \cos\theta \\ \sin\theta \\ \frac{m_z}{EI} + u_{o,y} \\ -f_x \\ -f_y \\ n_x\sin\theta - n_y\cos\theta - l_z \end{Bmatrix} \quad (30)$$

De esta manera, se ha obtenido un sistema de ecuaciones diferenciales que relaciona las cargas de la barra con la forma que adopta la misma.

Como ejemplo de aplicación del sistema (30), en el ANEXO II hay recogido el código de un programa que resuelve el problema de posición directo de una barra simplemente empotrada con una externa en su extremo, y que sirve de referencia y punto de partida para todos los demás programas que se describen en este proyecto. Utiliza un método de integración numérica, el Runge-Kutta de orden 4, para resolver el sistema (30) y poder obtener la forma de la barra al aplicarle una carga determinada. Pero para poder aplicar este método es necesario conocer el valor de las funciones en algún punto de la barra, por lo que se toma una aproximación inicial y se introduce todo este proceso dentro de un bucle de Newton-Raphson. La solución del problema directo dependerá de esa primera aproximación que se haya tomado. Este proceso de cálculo está descrito con más detalle en el apartado de CÁLCULOS Y ALGORITMOS, aplicado a otros casos concretos.

5.2. PARTICULARIZACIÓN PARA EL ANÁLISIS DE MANIPULADORES PARALELOS PLANOS

Ahora, se lleva a cabo el desarrollo del modelo plano de Kirchhoff, del sistema (30), para aplicarlo a la resolución de los problemas de posición de manipuladores paralelos planos. Considerando que la barra no está inicialmente deformada, $u_{o,y} = 0$. Por lo tanto, teniendo en cuenta que según la ley de Bernoulli-Euler el momento flector m_z es directamente proporcional a la curvatura κ,

$$\frac{m_z}{EI} = \frac{d\theta}{ds} = \frac{1}{\rho} = \kappa \qquad (31)$$

donde ρ es el radio de curvatura de la barra.

Por otro lado, si no hay fuerzas ni momentos aplicados a lo largo de la barra,

$$\begin{Bmatrix} \frac{dn_x}{ds} \\ \frac{dn_y}{ds} \\ \frac{dm_z}{ds} \end{Bmatrix} = \begin{Bmatrix} 0 \\ 0 \\ n_x \sin\theta - n_y \cos\theta \end{Bmatrix} \qquad (32)$$

Como vemos, la fuerza interna $\boldsymbol{n}$ es constante a lo largo de la barra. Si consideramos que sólo hay una carga aplicada en el extremo de la barra, de módulo $\boldsymbol{R}$ y orientada un ángulo ψ, tal y como aparece en la Ilustración 7, e introduciendo la ecuación (31) en la (32), queda:

$$\frac{d^2\theta}{d^2s} = \frac{1}{EI}[R\cos\psi\sin\theta - R\sin\psi\cos\theta] = \frac{R}{EI}\sin(\theta - \psi) \qquad (33)$$

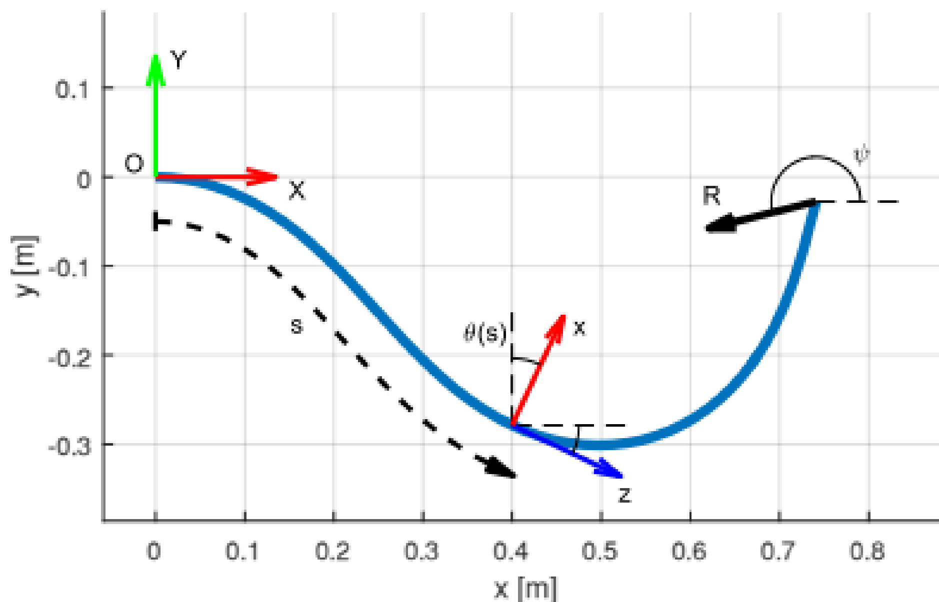


Ilustración 7. Barra plana deformada por una carga en su extremo

Por lo tanto, es sistema (30) queda simplificado de la siguiente forma:

$$\begin{Bmatrix} \frac{dx}{ds} \\ \frac{dy}{ds} \\ \frac{d^2\theta}{d^2s} \end{Bmatrix} = \begin{Bmatrix} \cos\theta \\ \sin\theta \\ \frac{R}{EI}\sin(\theta-\psi) \end{Bmatrix} \qquad (34)$$

La primera integración de la última ecuación del sistema (34) es directa y se obtiene la siguiente expresión:

$$\frac{d\theta}{ds} = \sqrt{2C - \frac{2R}{EI}\cos(\theta-\psi)} \qquad (35)$$

donde C es una constante de integración.

Integrar la ecuación (35) es más complicado y requiere un cambio en los parámetros, de θ, C a ϕ, k, de la siguiente forma:

$$C = \frac{R(2k^2-1)}{EI} \qquad (36)$$

$$\cos\left(\frac{\psi-\theta}{2}\right) = k\sin\phi \qquad (37)$$

El parámetro k se mantiene constante a lo largo de toda la barra, mientras que ϕ va variando. Al hacer la diferencial a ambos lados de la ecuación (37), obtenemos:

$$d\theta = \frac{2k\cos\phi}{\sqrt{1-k^2\sin^2\phi}}d\phi \qquad (38)$$

Al introducir las ecuaciones (36), (37) y (38) en la ecuación (35), y simplificando, resulta:

$$\int_0^L \sqrt{\frac{R}{EI}}ds = \int_{\phi_1}^{\phi_2} \frac{1}{\sqrt{1-k^2\sin^2\phi}}d\phi \qquad (39)$$

Que se puede reescribir como

$$\sqrt{R} = \frac{\sqrt{EI}}{L}[F(k,\phi_2) - F(k,\phi_1)] \qquad (40)$$

Se utiliza $F(k,\phi)$ para nombrar a la integral elíptica de primer orden. Los resultados numéricos de estas integrales están tabulados, y es lo que se empleará en la práctica para resolver el comportamiento de las barras flexibles.

Como se ha indicado antes, ϕ varía a lo largo de la barra entre ϕ_1 y ϕ_2, que dependen de las condiciones de contorno en los extremos de la barra. Por ejemplo, si consideramos una barra empotrada horizontalmente en su extremo inicial, como se ve en la Ilustración 7, sabemos que la orientación de la barra en ese punto es $\theta = 0$, y por lo tanto el valor de ϕ correspondiente es:

$$\phi_1 = \arcsin \frac{1}{k} \cos \frac{\psi}{2} \qquad (41)$$

Para el cálculo de ϕ en el otro extremo, se introducen las ecuaciones (36) y (37) en la (35):

$$\frac{d\theta}{ds} = 2k \sqrt{\frac{R}{EI}} \cos \phi \qquad (42)$$

Si el extremo es articulado, hay un punto de inflexión en la curva, lo que implica que $\frac{d\theta}{ds} = 0$, y por lo tanto en ese punto

$$\phi_2 = \frac{q\pi}{2} \qquad \text{donde } q = 1,3,5, \dots \qquad (43)$$

El parámetro q define el modo de pandeo de la barra, que se corresponde con el número de puntos de inflexión que aparecen en ella. Para el modo 1, $q = 1$, para el modo 2, $q = 3$, etc.

Utilizando la ecuación (39) poniendo $\cos \theta$ como una función de k y ψ, integramos la coordenada x del sistema (34):

$$x = -\sqrt{\frac{EI}{R}} \cos \psi \left[2 \int_{\phi_1}^{\phi_i} \sqrt{1 - k^2 \sin^2 \phi}\, d\phi - \int_{\phi_1}^{\phi_i} \frac{1}{\sqrt{1 - k^2 \sin^2 \phi}} d\phi \right] - \\ -\sqrt{\frac{EI}{R}} 2k \sin \psi \left[\cos \phi_i - \cos \phi_1 \right] \qquad (44)$$

La primera integral que aparece es la integral elíptica de segundo orden, y se denota $E(k, \phi)$. Por lo tanto, se puede obtener la coordenada x de la deformada en un punto de la barra definido por ϕ_i de la siguiente forma:

$$x = -\sqrt{\frac{EI}{R}} \cos \psi \left[2E(k, \phi_i) - 2E(k, \phi_1) - F(k, \phi_i) + F(k, \phi_1) \right] - \\ -\sqrt{\frac{EI}{R}} 2k \sin \psi \left[\cos \phi_i - \cos \phi_1 \right] \qquad (45)$$

Para obtener la coordenada x en el extremo de la barra, basta sustituir $\phi_i = \phi_2$.

De igual manera, la coordenada y en un punto cualquiera de la barra es:

$$y = -\sqrt{\frac{EI}{R}}\sin\psi\,[2E(k,\phi_i) - 2E(k,\phi_1) - F(k,\phi_i) + F(k,\phi_1)] + \\ + \sqrt{\frac{EI}{R}}\,2k\cos\psi\,[\cos\phi_i - \cos\phi_1] \qquad (46)$$

De esta forma, se ha conseguido definir la forma que adopta la barra en función de los parámetros k y ϕ. Estos dos parámetros por sí solos definen el estado de carga y deformación en el que se encuentra la barra.

No obstante, hay que tener en cuenta que existen unas limitaciones en el uso de estos parámetros. Para cada k, existe un valor mínimo k_{min} para obtener valores reales de ϕ_1. En la Ilustración 8 se puede ver cuáles son las posibles combinaciones de ψ y k que son válidas en el cálculo de ϕ_1. Las zonas entre [-1, $-k_{min}$] y [k_{min}, 1] no son válidas.

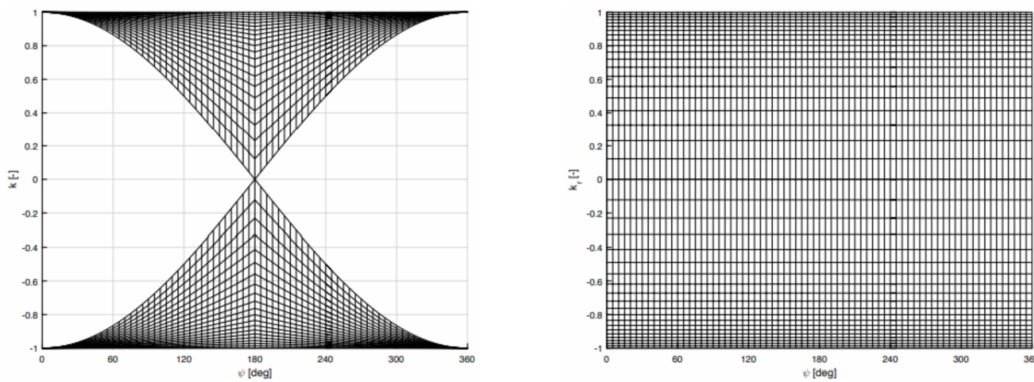


Ilustración 8. Rango de valores existentes de k y k_{rel} para cada ψ

Para simplificar este problema, se puede tomar un valor de k relativo, que transforma esa región de puntos válidos en un rectángulo, como se ve en la imagen.

$$k_{rel} = Signo\ (k)\,\frac{|k| - |k_{min}|}{1 - |k_{min}|} \qquad (47)$$

De esta manera, son válidos todos los valores de k_{rel} entre [-1, 1].

En la Ilustración 9 se puede ver gráficamente cómo se relacionan las variables k_{rel}, ψ y R.

Tomando un valor constante de ψ, cada valor de k_{rel} corresponde a un único valor de R. Sin embargo, a cada R le pueden corresponder varios valores distintos de k_{rel}. Esto se puede apreciar al hacer un corte en la gráfica a una ψ concreta, que es lo que aparece en la Ilustración 10.

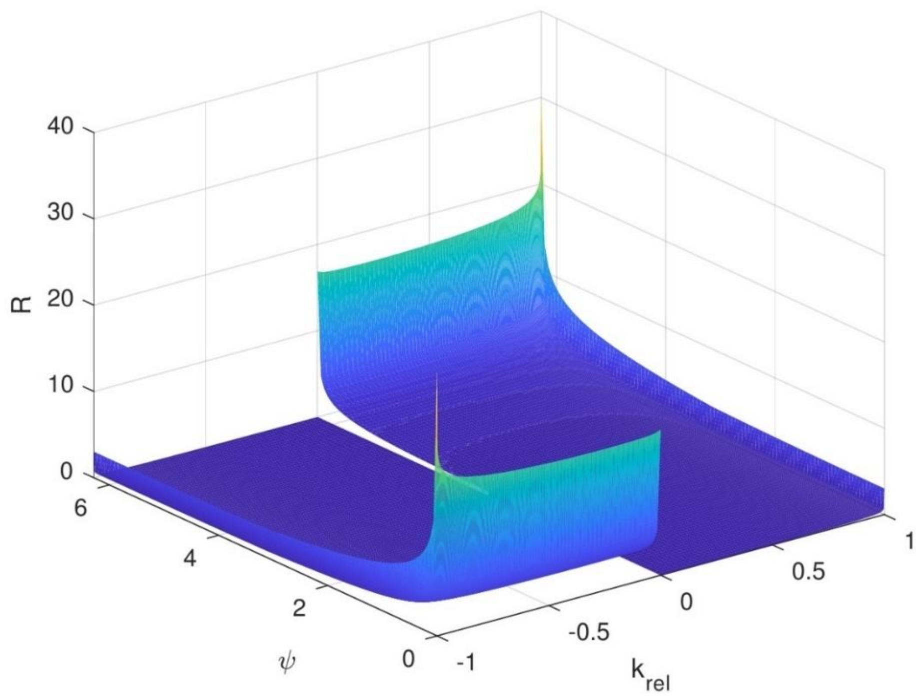


Ilustración 9. Superficie de relación entre k_{rel}, ψ y R

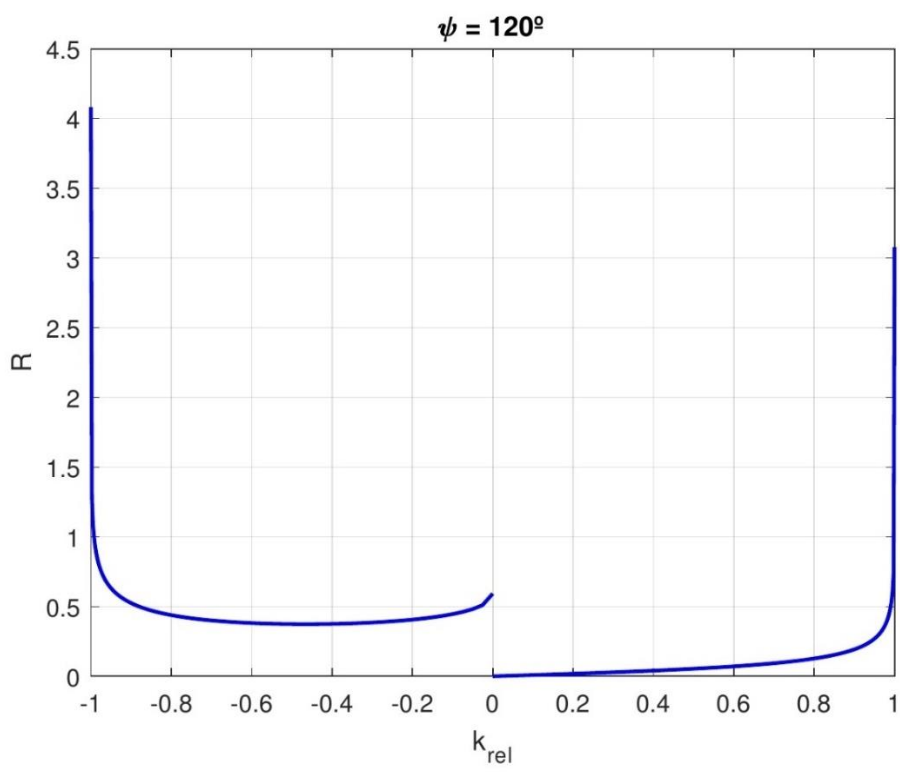


Ilustración 10. Relación entre k_{rel} y R para una ψ concreta

6. ANÁLISIS DE ALTERNATIVAS

La labor que se lleva a cabo en este proyecto es, básicamente, aplicar los principios teóricos que se han descrito en el ESTADO DEL ARTE para resolver los problemas cinemáticos de posición. Es precisamente este proceso de cálculo el que supone un mayor dilema. En varias ocasiones, es necesario el empleo de métodos iterativos para resolver sistemas de ecuaciones.

En principio, los métodos iterativos cerrados, como el de bisección, garantizan su convergencia y, por tanto, aseguran una solución para el sistema. Sin embargo, requieren conocer más datos sobre la función a tratar y, sobre todo, exigen que sea continua.

Los métodos abiertos, en cambio, aunque no siempre converjan, son mucho más eficientes y rápidos y tienen, en general, menos requerimientos para su aplicación. El método de Newton-Raphson o el de la secante son algunos ejemplos de este tipo.

Los programas que se realizan llevan a cabo cálculos repetitivos en miles de puntos. Y es posible que en todos ellos haya que aplicar uno de estos métodos de cálculo. Por lo tanto, la rapidez y la eficiencia son fundamentales. En este sentido, el método de Newton-Raphson tiene una velocidad de convergencia elevada sin demasiados requerimientos en cuanto a las ecuaciones a resolver. Es por eso que es el método elegido para aplicar en el desarrollo del proyecto.

No obstante, el método de la bisección, antes mencionado, es otra opción que perfectamente podría ser válida. Exige conocer más valores de la función y que ésta sea continua, pero es bastante rápido y tiene la ventaja de que siempre converge.

7. ANÁLISIS DE RIESGOS

El resultado del proyecto son los programas que resuelven los problemas de posición de los mecanismos que se han descrito. Por lo tanto, los riesgos que se consideran tienen que ver con la eficacia con que funcionan estos programas.

Tras un análisis de los posibles errores o problemas que puedan surgir en el cálculo, se llega a la conclusión de que hay dos riesgos fundamentales relacionados con los programas que aplican integrales elípticas y que se debe tener en cuenta. Se valoran según la siguiente matriz de probabilidad-impacto:

	IMPACTO LEVE (1)	IMPACTO MODERADO (2)	IMPACTO GRAVE (3)
IMPROBABLE (1)	1	2	3
OCASIONAL (2)	2	4	6
FRECUENTE (3)	3	6	9

Tabla 1. Valoración de riesgos

Los dos riesgos que se consideran son:

1. Que el programa no encuentre todas las soluciones posibles. No quiere decir que las que alcanza no sean válidas; simplemente podría haber más y sería imposible saberlo. La probabilidad de que esto ocurra es frecuente, aunque depende de las discretizaciones y tolerancias que se tomen. Además, se considera que tiene un impacto moderado sobre los intereses del proyecto. La valoración de este riesgo, por lo tanto, es de 6 en la Tabla 1, una valoración elevada que obliga a tener especial cuidado a la hora de escoger una discretización fina y una tolerancia suficientemente grande, con el fin de reducir este riesgo.

2. Que el programa tarde demasiado tiempo en resolver el problema. Es un riesgo especialmente evidente en el caso del mecanismo de 3 grados de libertad, que tiene un impacto leve, pero es relativamente frecuente y, por lo tanto, una valoración de 3, menor que en el riesgo anterior. En este caso, cuanto más grosera sea la discretización de las variables, menos tiempo tarda el programa, pero es evidente que esta medida sería contraproducente con el riesgo anterior. Es necesario encontrar un equilibrio, que permita reducir ambos en la medida de lo posible.

8. DESCRIPCIÓN DE LA PROPUESTA

Para llevar a cabo el análisis del comportamiento de los mecanismos ultraflexibles, es necesario conocer previamente los principios teóricos que rigen el comportamiento de una barra flexible. Se estudia el modelo barra de Cosserat y la simplificación plana de Kirchhoff, como fundamentos básicos a partir de los cuales se llevará a cabo el posterior análisis de mecanismos. El conocimiento de estos principios es el punto de partida del proyecto y se aplican en la resolución de los problemas cinemáticos directo e inverso de los distintos mecanismos.

El desarrollo teórico, desde los modelos citados hasta las ecuaciones definitivas que se aplican, está descrito en el apartado de ESTADO DEL ARTE, y se ha obtenido de los artículos *"Position Analysis in Planar Parallel Continuum Mechanisms"* y "*Nonlinear Flexure Analysis of Mechanisms"*, ambos referenciados en la BIBLIOGRAFÍA.

Se emplea la herramienta informática Matlab para la resolución de los problemas cinemáticos. A partir de los fundamentos teóricos, se realiza un programa que permite resolver los problemas directo e inverso de cada mecanismo, pudiendo simular su comportamiento.

Además de los dos mecanismos de barras flexibles que se han indicado anteriormente, y que son el objetivo principal del proyecto, el análisis previo y la programación del mecanismo de barras rígidas sirve como referencia en la forma de proceder en la resolución de los problemas cinemáticos. Se trata de un caso que matemáticamente no es tan complicado, pero que permite generar una idea clara de lo que se quiere en los siguientes programas.

En este caso, Matlab permite una gran versatilidad. En un mismo programa, se puede modificar fácilmente los datos y parámetros de entrada, de manera que es posible analizar el comportamiento del mecanismo para diferentes situaciones: variaciones de la posición de entrada, geometría del mecanismo, cargas externas aplicadas, etc. De esta forma, se puede llevar a cabo un estudio más exhaustivo, considerando un gran número de variables que harán que las conclusiones obtenidas sean más precisas.

Además, se emplea la guía *"Essential MATLAB for engineers and scientists"* para resolver las dudas que pueda plantear el uso del propio software

9. DESCRIPCIÓN DE TAREAS, PROCEDIMIENTOS Y PLANIFICACIÓN

En este apartado se describen las fases y tareas de que consta el proyecto. En todas ellas, los únicos recursos técnicos que se utilizan son una licencia del software Matlab y un ordenador.

Las tareas son las siguientes:

T.1 PREPARACIÓN

El primer paso en el proyecto es definir completamente las líneas de actuación que se van a seguir, los principios de los que se parte y los objetivos que se quieren alcanzar.
Duración: 1 semana

T.2 FUNDAMENTOS TÉORICOS

T.2.1 Modelo de Cosserat

Estudio de los fundamentos teóricos del modelo de barra de Cosserat. El objetivo es llegar a entenderlos y conocerlos a fondo para poder aplicarlos cálculos posteriores mediante integración numérica.
Esta tarea debe ser predecesora de la T.3.1 y la T.4.1.
Duración: 1 semana

T.2.2 Aplicación a mecanismos planos de cadena cerrada

Análisis del desarrollo matemático de las ecuaciones del modelo de Kirchhoff aplicadas a mecanismos planos de cadena cerrada. Se utilizan integrales elípticas que se aplicarán posteriormente en la programación.
Esta tarea debe ser predecesora de la T.3.2 y la T.4.2.
Duración: 1 semana

T.3 MECANISMO 3PRR

T.3.1 Programación

Programación en Matlab de la resolución de los problemas de posición del mecanismo 3PRR. Sirve como referencia del método a seguir en este tipo de cálculos para los programas siguientes.
Duración: 4 semanas

T.3.2 Validación (HITO)

Certificación de que los programas funcionan correctamente

T.4 MECANISMO 2RFR

T.4.1 Programación por integración numérica

Consiste en llevar a cabo los programas que resuelvan los problemas de posición directo e inverso del mecanismo 2RFR mediante integración numérica.
Duración: 5 semanas

T.4.2 Programación por integrales elípticas

Programación de la resolución del problema directo del mecanismo 2RFR utilizando integrales elípticas.
Duración: 5 semanas

T.4.3 Validación (HITO)

Certificación de que los programas funcionan correctamente

T.5 MECANISMO 3PFR

T.5.1 Programación por integración numérica

Consiste en llevar a cabo los programas que resuelvan los problemas de posición directo e inverso del mecanismo 3PFR mediante integración numérica.

Duración: 6 semanas

T.5.2 Programación por integrales elípticas

Programación de la resolución del problema directo del mecanismo 2RFR utilizando integrales elípticas.

Duración: 6 semanas

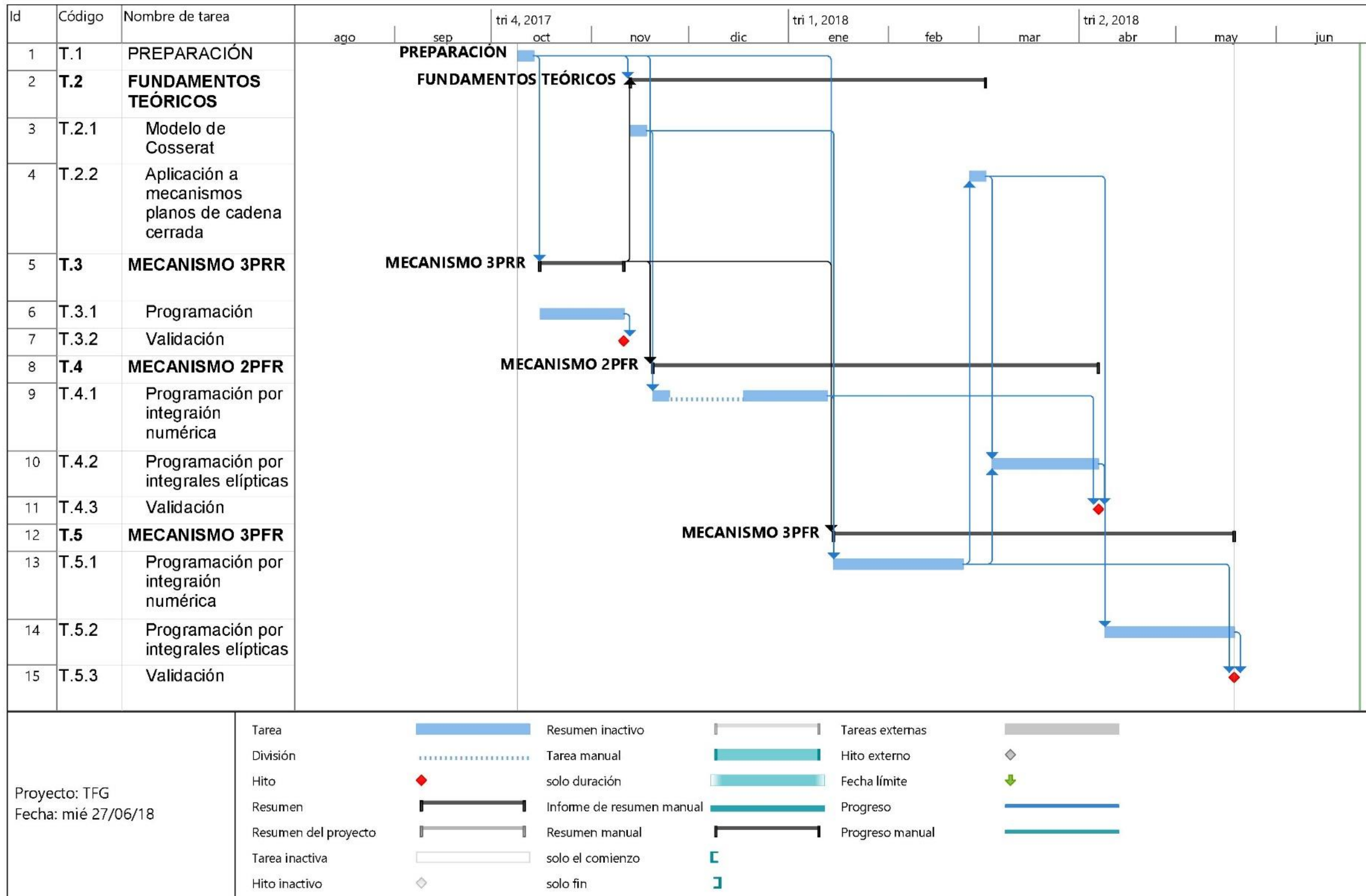
T.5.3 Validación (HITO)

Certificación de que los programas funcionan correctamente

El alumno es el responsable encargado de todas las tareas, y el tutor del proyecto es el supervisor.

El proyecto comienza el 19 de octubre de 2017 y finaliza el 18 de mayo de 2018. En este caso, todas las tareas son sucesivas, por lo que podría considerarse que todas son críticas. Un retraso en cualquiera de ellas supone un alargamiento en el tiempo total del proyecto, que conlleva un aumento en los costes del mismo. En el siguiente apartado se puede ver el diagrama de Gantt de las tareas descritas.

10. DIAGRAMA DE GANTT



11. CÁLCULOS Y ALGORITMOS

En este apartado, se describen los procedimientos de cálculo que se siguen para la resolución de los problemas de posición. La implementación de estos algoritmos en Matlab da como resultado los programas que están recogidos en el ANEXO II.

11.1. 3PRR

En esta parte del proyecto, se ha analizado el comportamiento del mecanismo 3RPP que se puede ver en la figura. Se trata de un mecanismo de 3 grados de libertad, en el que cada una de las ramas que sujetan el actuador triangular consta de dos pares de rotación y un par prismático. Los pares prismáticos, dos horizontales y uno vertical, son las entradas del mecanismo; y su posición, que viene dada por los parámetros λ_1, λ_2 y λ_3, determinará la posición de los motores que accionan el sistema.

Se coloca un sistema de referencia fijo con origen en el motor 1 y otro sistema de referencia móvil unido a la plataforma triangular, lo cual permite determinar su posición en todo momento a través de las coordenadas del punto $P\ (x_P, y_P)$ y del ángulo de orientación ψ. Se conocen todas las demás dimensiones geométricas del mecanismo reflejadas en la Ilustración 11.

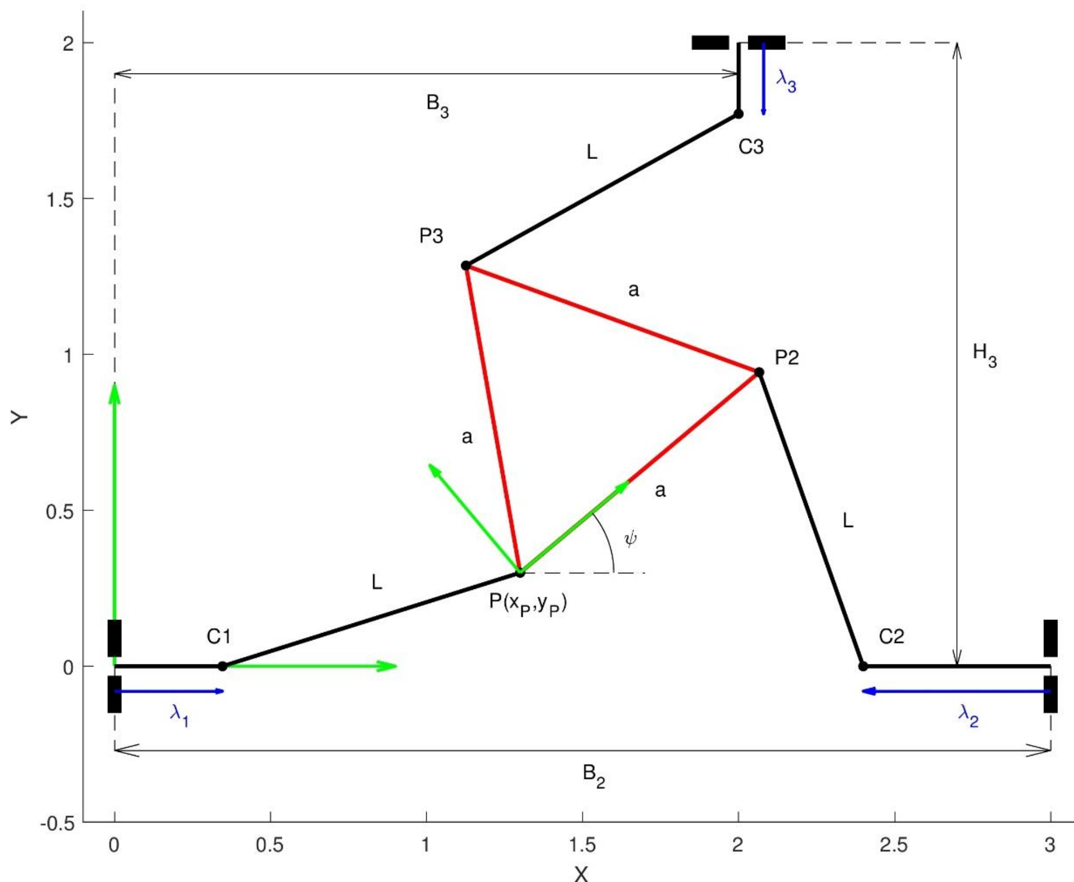


Ilustración 11. Mecanismo 3PRR

El análisis detallado del comportamiento de este mecanismo implica la resolución de los problemas cinemáticos directo e inverso.

11.1.1. PROBLEMA INVERSO

La resolución del problema cinemático inverso consiste en determinar cuáles deben ser las entradas al mecanismo dada una posición concreta del elemento terminal. En este caso, la posición de la plataforma triangular está definida por las coordenadas del punto $P\ (x_P, y_P)$ y el ángulo de orientación ψ, que serán valores conocidos; y las entradas del sistema serán las posiciones de los motores, definidas por λ_1, λ_2 y λ_3, cuyo valor es la solución que se busca del problema inverso.

Se aborda la resolución del problema geométricamente. La idea es buscar qué puntos de cada una de las guías lineales cumple los requisitos geométricos del mecanismo.

Se conoce la posición exacta y dimensiones de la plataforma triangular, por lo que, además de las coordenadas del punto P, se puede calcular las coordenadas de los otros dos vértices, $P2$ y $P3$ como:

$$\begin{cases} x_{P2} = x_P + a\cos\psi \\ y_{P2} = y_P + a\sin\psi \end{cases} \tag{48}$$

$$\begin{cases} x_{P3} = x_P + \dfrac{a}{2}\cos\psi - \dfrac{a\sqrt{3}}{2}\sin\psi \\ y_{P3} = y_P + \dfrac{a}{2}\sin\psi + \dfrac{a\sqrt{3}}{2}\cos\psi \end{cases} \tag{49}$$

Para calcular la posición de cada uno de los puntos $C1, C2, C3$ del mecanismo, hay que tener en cuenta que deben cumplir dos condiciones: por un lado, estar sobre la guía lineal del motor correspondiente; y por otro, estar separados una distancia L (longitud de la barra intermedia) del vértice correspondiente de la plataforma. Estas dos condiciones se traducen geométricamente como una recta coincidente con la guía y una circunferencia de radio L y centro en el vértice, respectivamente. La intersección entre ambas es el punto C, cuya posición determina el valor de λ que se busca.

Para el caso de la rama 1, se puede ver las condiciones geométricas mencionadas en la Ilustración 12.

Matemáticamente, las ecuaciones de la recta que define la guía y la circunferencia de puntos que distan L del vértice P, respectivamente, son:

$$y = 0 \tag{50}$$

$$(x - x_P)^2 + (x - y_P)^2 = L^2 \tag{51}$$

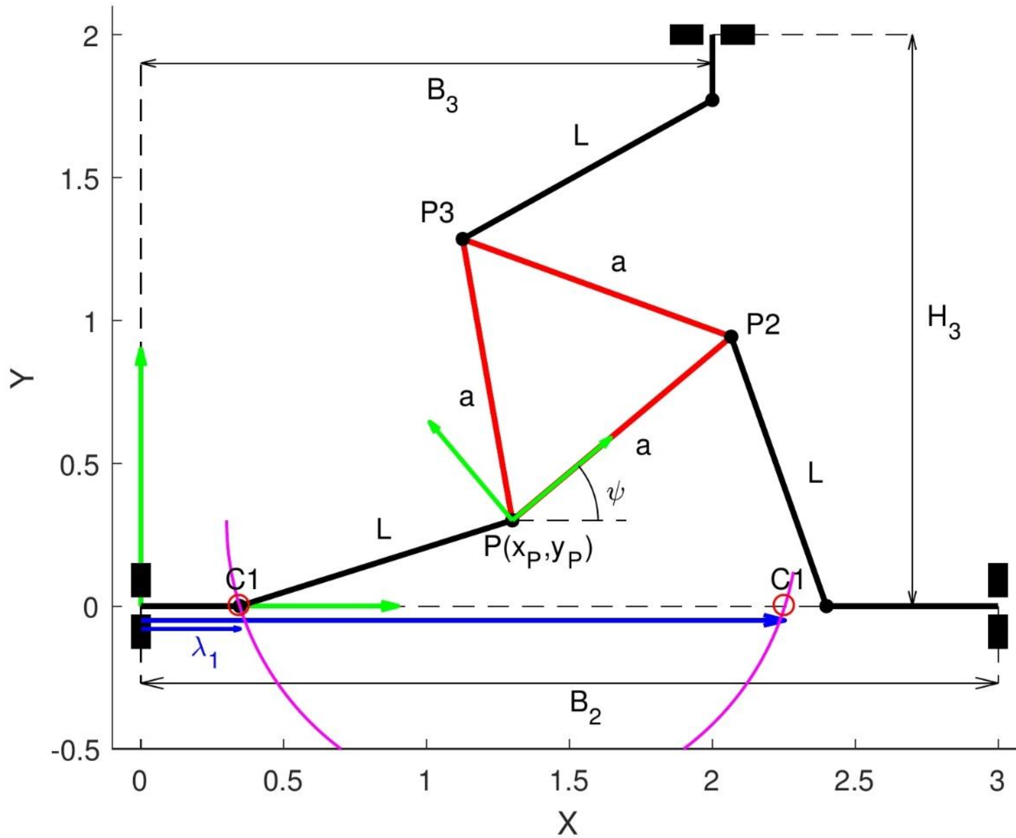


Ilustración 12. Cálculo gráfico de la barra 1

Resolviendo el sistema de ecuaciones (50) y (51), se obtienen las coordenadas de la intersección entre ambas, $C1$. Es evidente, por la ecuación (50), que la coordenada y_{C1} tiene que ser 0. Al ser (4) una ecuación de segundo orden, puede haber dos posibles valores para la coordenada x_{C1}:

$$\begin{cases} x_{C1} = x_P \pm \sqrt{L^2 - y_P^2} \\ y_{C1} = 0 \end{cases} \tag{52}$$

Es decir, puede haber dos puntos de intersección entre la recta (50) y la circunferencia (51), dos puntos que cumplan las condiciones geométricas que impone el mecanismo.

En este caso, como se puede ver en la imagen, la longitud λ_1 coincide con la coordenada x de $C1$. Por lo tanto, λ_1 tiene dos valores posibles:

$$\lambda_1 = \begin{cases} x_P + \sqrt{L^2 - y_P^2} \\ x_P - \sqrt{L^2 - y_P^2} \end{cases} \tag{53}$$

Los dos valores de λ_1 que se obtienen son dos posibles posiciones de la rama 1 del mecanismo, como se pueden ver a continuación:

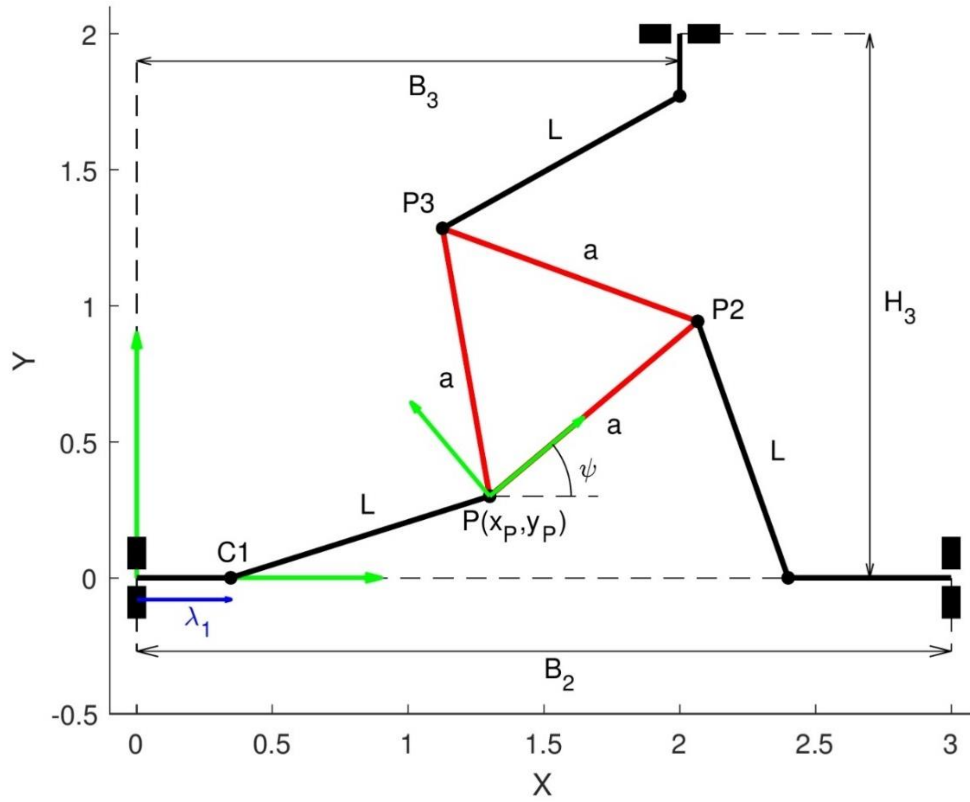


Ilustración 13. Solución 1 de la barra 1

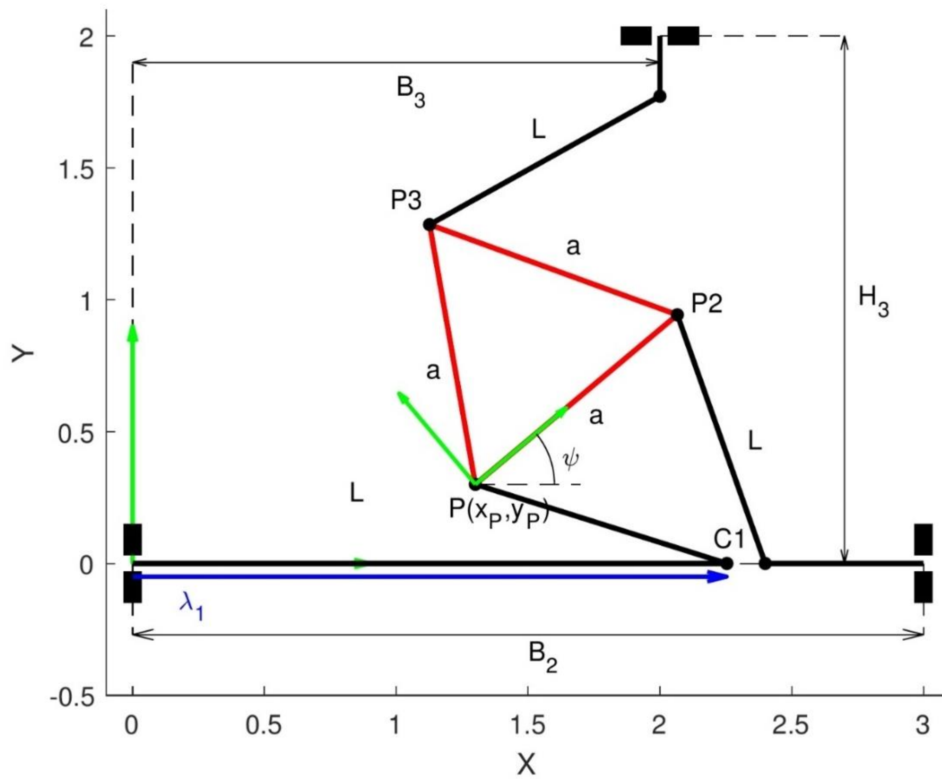


Ilustración 14. Solución 2 de la barra 2

Siguiendo el mismo razonamiento, se pueden calcular los valores de λ_2 y λ_3. Para la rama 2, las ecuaciones de la recta que define la guía y de la circunferencia de puntos situados a una distancia L del vértice $P2$ son:

$$y = 0 \qquad (54)$$

$$(x - x_{P2})^2 + (x - y_{P2})^2 = L^2 \qquad (55)$$

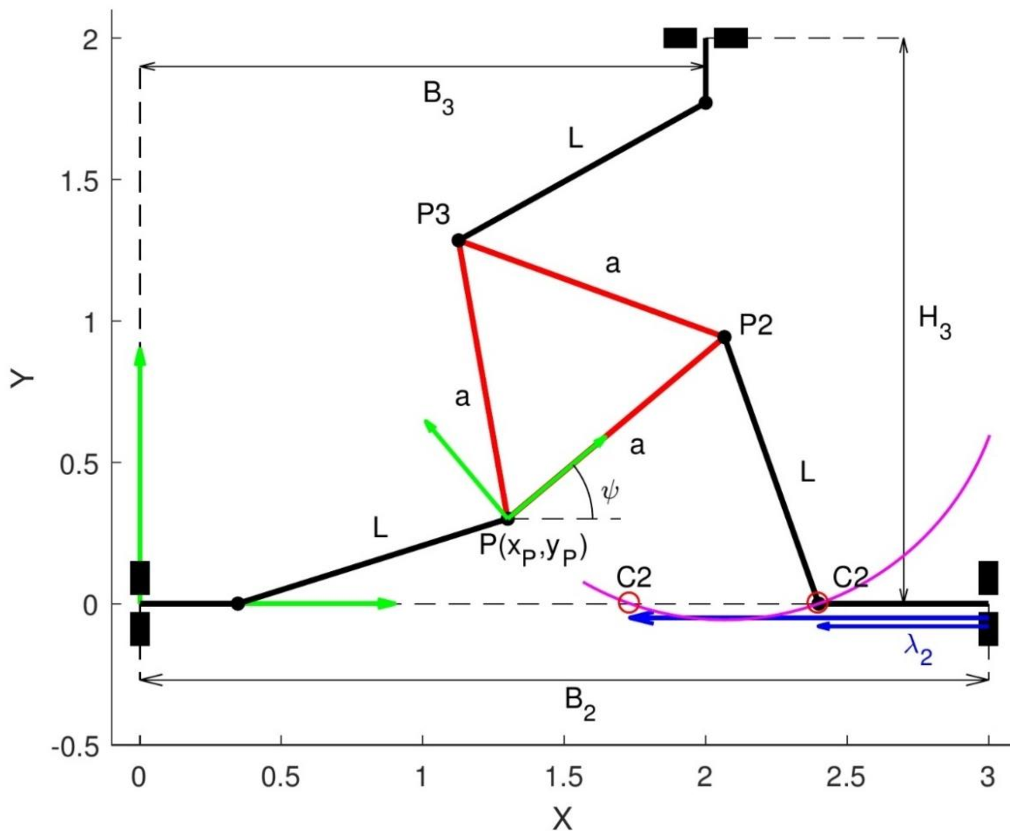


Ilustración 15. Cálculo gráfico de la barra 2

Al igual que antes, resolviendo el sistema de ecuaciones (54) y (55) se obtienen las coordenadas del punto $C2$. Hay dos soluciones posibles:

$$\begin{cases} x_{C2} = x_{P2} \pm \sqrt{L^2 - y_{P2}^2} \\ y_{C2} = 0 \end{cases} \qquad (56)$$

A partir de la coordenada x del punto $C2$ se puede calcular el valor de λ_2 como:

$$\lambda_2 = B_2 - x_{C2} = \begin{cases} B_2 - x_{P2} - \sqrt{L^2 - y_{P2}^2} \\ B_2 - x_{P2} + \sqrt{L^2 - y_{P2}^2} \end{cases} \qquad (57)$$

Los dos valores posibles de λ_2 determinan dos posibles posiciones de la rama 2:

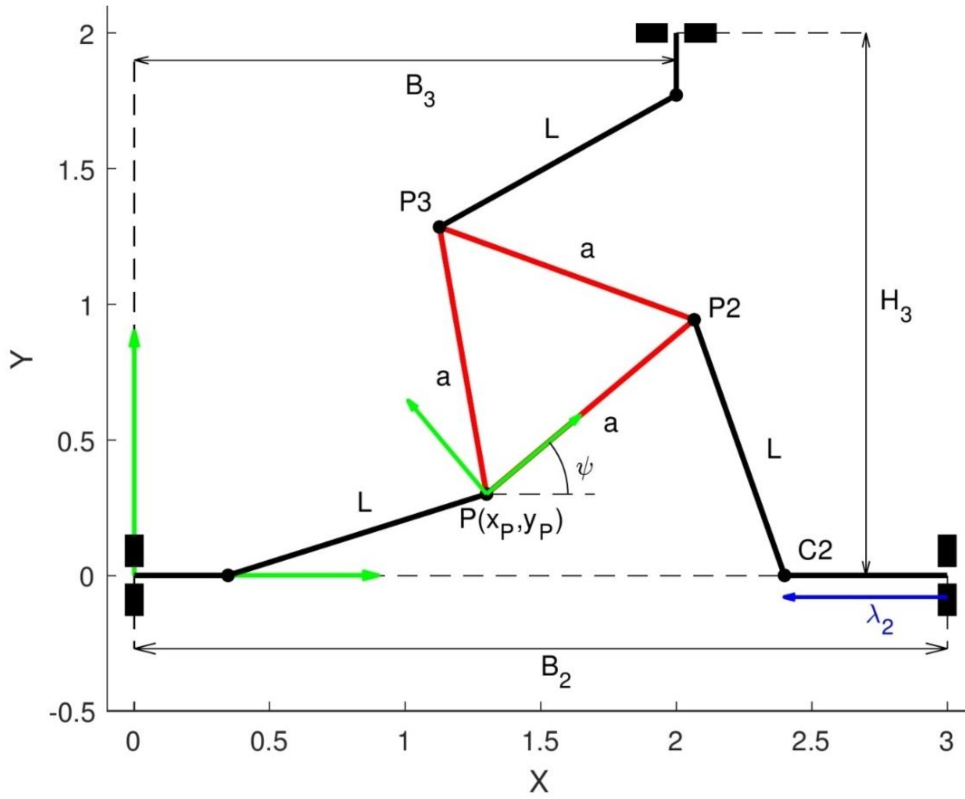


Ilustración 16. Solución 1 de la barra 2

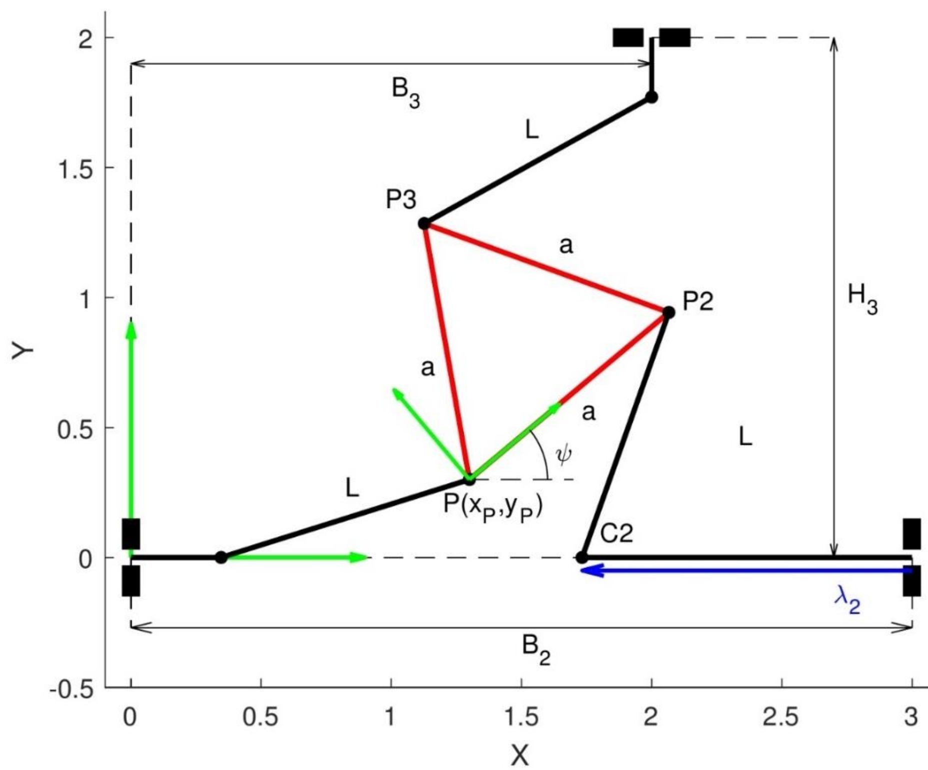


Ilustración 17. Solución 2 de la barra 2

El procedimiento para la rama 3 es el mismo, como se ve en la siguiente imagen.

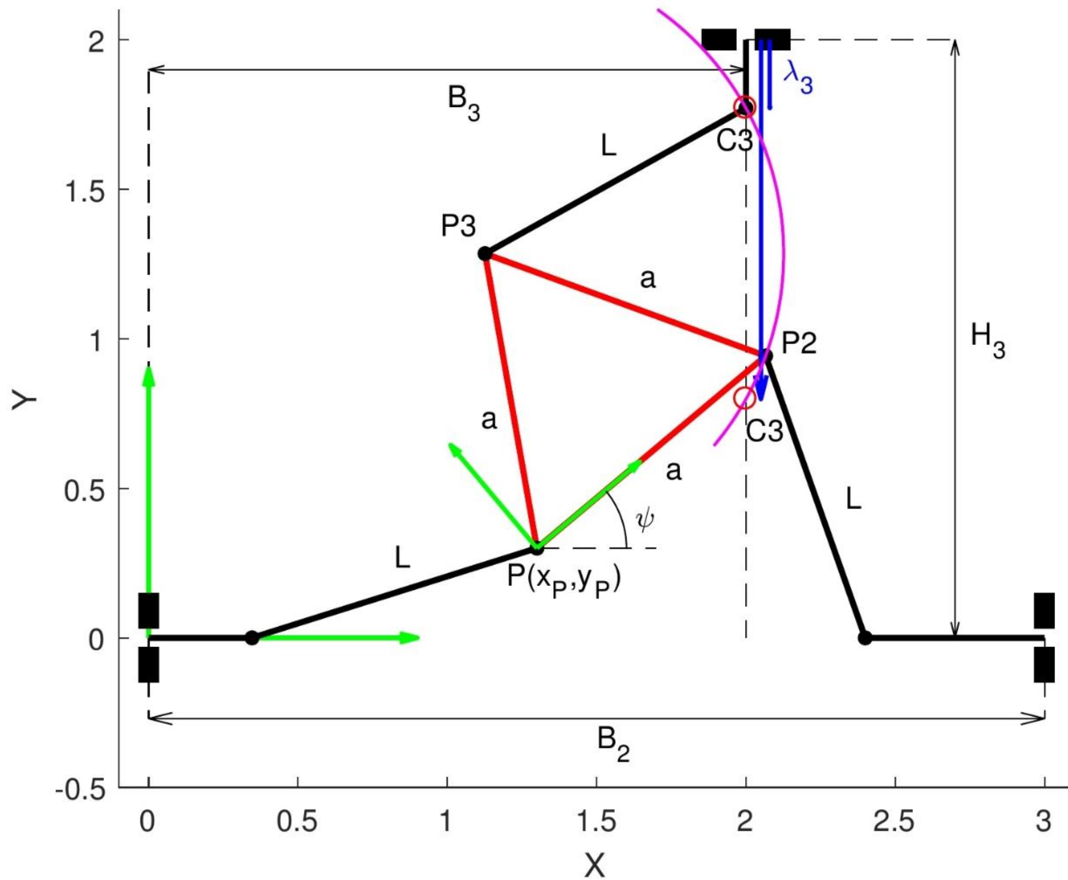


Ilustración 18. Cálculo gráfico de la barra 3

Las ecuaciones matemáticas de la recta que define la guía y la circunferencia de centro $P3$ y radio L son:

$$(x - x_{P3})^2 + (y - y_{P3})^2 = L^2 \qquad (58)$$

$$x = B_3 \qquad (59)$$

La resolución del sistema de ecuaciones (58) y (59) da las coordenadas del punto $C3$:

$$\begin{cases} x_{C3} = B_3 \\ y_{C3} = y_{P3} \pm \sqrt{L^2 - (B_3 - x_{P3})^2} \end{cases} \qquad (60)$$

Con la coordenada y de $C3$ se puede calcular λ_3 como:

$$\lambda_3 = H_3 - y_{C3} = \begin{cases} H_3 - y_{P3} - \sqrt{L^2 - (B_3 - x_{P3})^2} \\ H_3 - y_{P3} + \sqrt{L^2 - (B_3 - x_{P3})^2} \end{cases} \qquad (61)$$

De nuevo, existen dos posibles posiciones para la rama 3 del mecanismo:

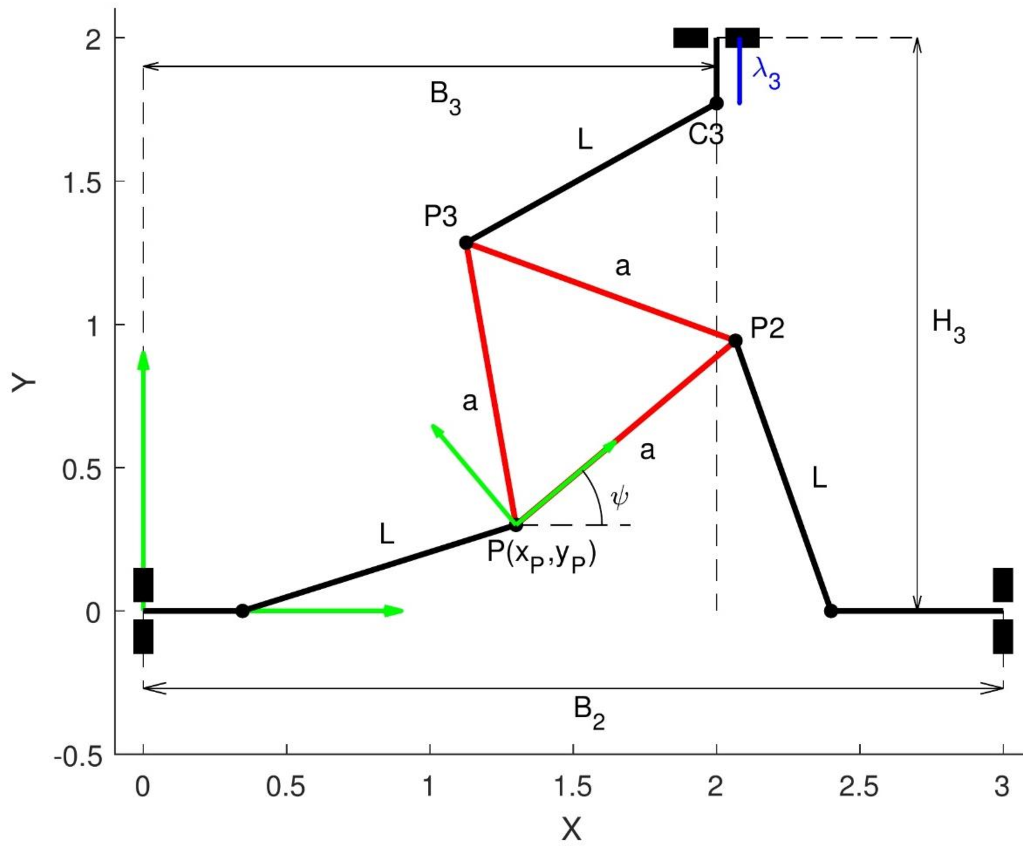


Ilustración 19. Solución 1 de la barra 3

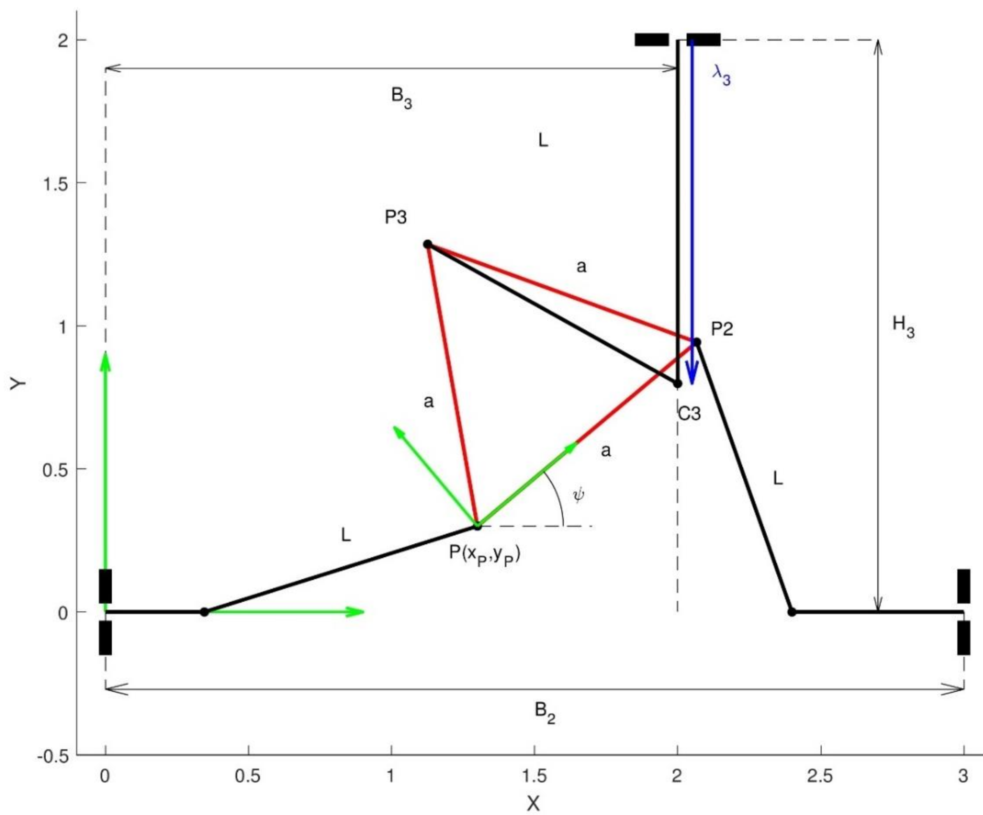


Ilustración 20. Solución 2 de la barra 3

Hasta ahora, se ha considerado que hay dos puntos de intersección para cada una de las tres ramas del mecanismo; es decir, que los sistemas de ecuaciones tienen dos soluciones reales y, por tanto, hay dos valores reales y distintos de λ. Sin embargo, esto no tiene por qué ser siempre así. Viendo las soluciones de los sistemas de ecuaciones, es evidente que para que éstas sean reales los radicandos deben ser mayores o iguales que 0. Por lo tanto, para que todos los sistemas tengan solución real, y en consecuencia el problema inverso tenga solución, tienen que cumplirse las siguientes condiciones:

$$L^2 - y_P^2 \geq 0 \tag{62}$$

$$L^2 - y_{P2}^2 \geq 0 \tag{63}$$

$$L^2 - (B_3 - x_{P3})^2 \geq 0 \tag{64}$$

Si alguna de estas condiciones no se cumple, significa que alguno de los sistemas de ecuaciones que se han planteado no tiene solución real, y no existe intersección entre la recta y la circunferencia considerados. Por lo tanto, el problema inverso no tiene solución.

También puede ser que alguno de los radicandos sea igual a 0. Esto implica la tangencia entre la recta y la circunferencia, lo que dará una solución doble en el sistema en cuestión, y por tanto un único valor de λ. El mecanismo se encuentra en una posición de singularidad del problema inverso, en la que pierde grados de libertad porque aparece una dependencia entre las variables de salida, x_P, y_P, ψ. En la práctica, es muy difícil que los radicandos mencionados tomen un valor exactamente igual a 0, por lo que se considera que esto es así cuando es menor que una tolerancia previamente establecida, en valor absoluto.

Las soluciones del problema inverso son todas las posibles combinaciones de valores λ_1, λ_2 y λ_3 obtenidos de los sistemas de ecuaciones planteados. Puede haber hasta 8 combinaciones distintas, en el caso de que haya dos valores posibles para cada uno de λ_1, λ_2 y λ_3. El número de combinaciones se reduce si alguno de los sistemas tiene solución doble (el mecanismo está en una posición singular), hasta un mínimo de 1 cuando λ_1, λ_2 y $\lambda_3$son únicos.

Además de la mera resolución del problema cinemático inverso, se lleva a cabo un análisis del espacio de trabajo del mecanismo. Se trata de, para una posición determinada de la plataforma triangular definida por (x_P, y_P) y ψ, aplicar las condiciones (62), (63) y (64) para ver si el problema inverso tiene solución. Se puede comprobar de esta manera si el elemento terminal del mecanismo puede llegar a dicha posición.

Para analizar en profundidad el espacio de trabajo, se discretizan las variables x_P, y_P y ψ en un intervalo concreto y se comprueba que el problema inverso tenga solución en cada combinación (x_P, y_P, ψ). Las ternas que sí lo cumplan se representan como puntos en un sistema de ejes coordenados, con el fin de ver los resultados de una forma más gráfica, como se muestra en la siguiente imagen:

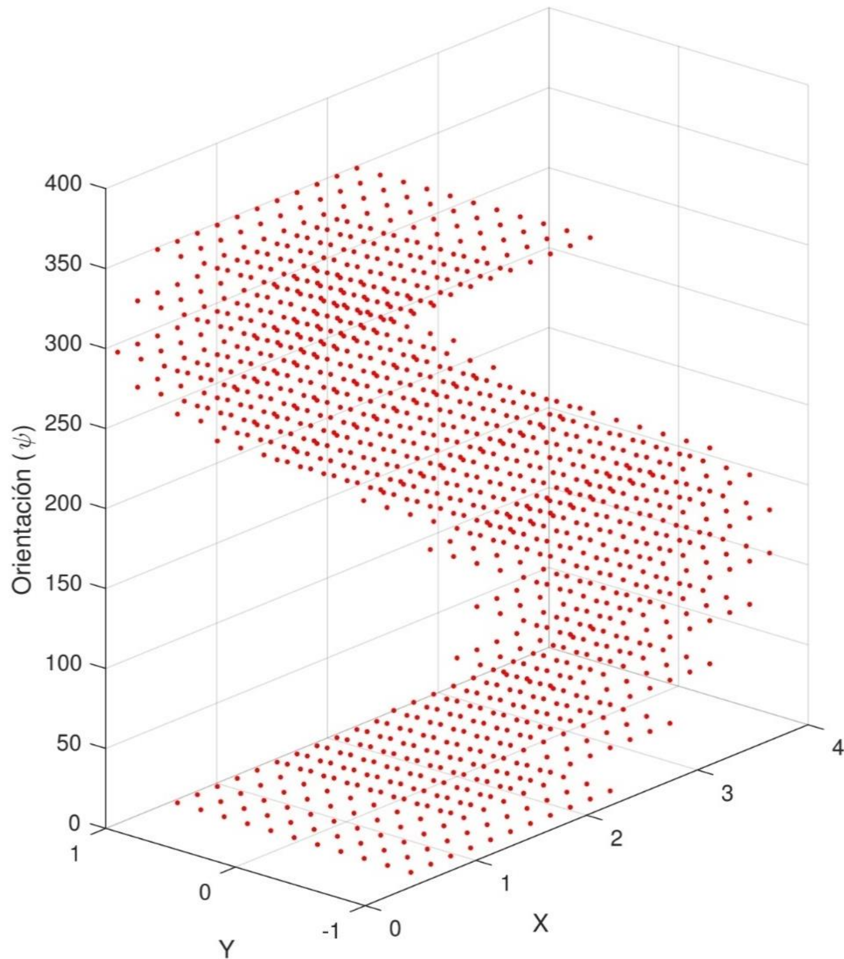


Ilustración 21. Ejemplo de representación gráfica del espacio de trabajo

Así, se aprecia con mayor facilidad cuál es la región del espacio (x_P, y_P, ψ) a la que el mecanismo es capaz de llegar. Es importante señalar que el espacio de trabajo está delimitado por las posiciones de singularidad que se ha mencionado anteriormente.

11.1.2. PROBLEMA DIRECTO

La resolución del problema cinemático directo consiste en determinar cuál es la salida del mecanismo (la posición del su elemento terminal) para una entrada dada. En este caso, la entrada del mecanismo está definida por las variables λ_1, λ_2 y λ_3, que son conocidas; y la posición de la plataforma triangular viene dada por x_P, y_P y ψ, cuyos valores son la solución del problema directo.

Es necesario obtener ecuaciones que relacionen las variables de entrada con la posición del elemento terminal. Para ello, se toman tres lazos cerrados en el mecanismo y se plantean sus ecuaciones vectoriales. Los lazos elegidos y las ecuaciones resultantes, una vez desarrolladas en sus componentes, de cada uno de ellos son los siguientes:

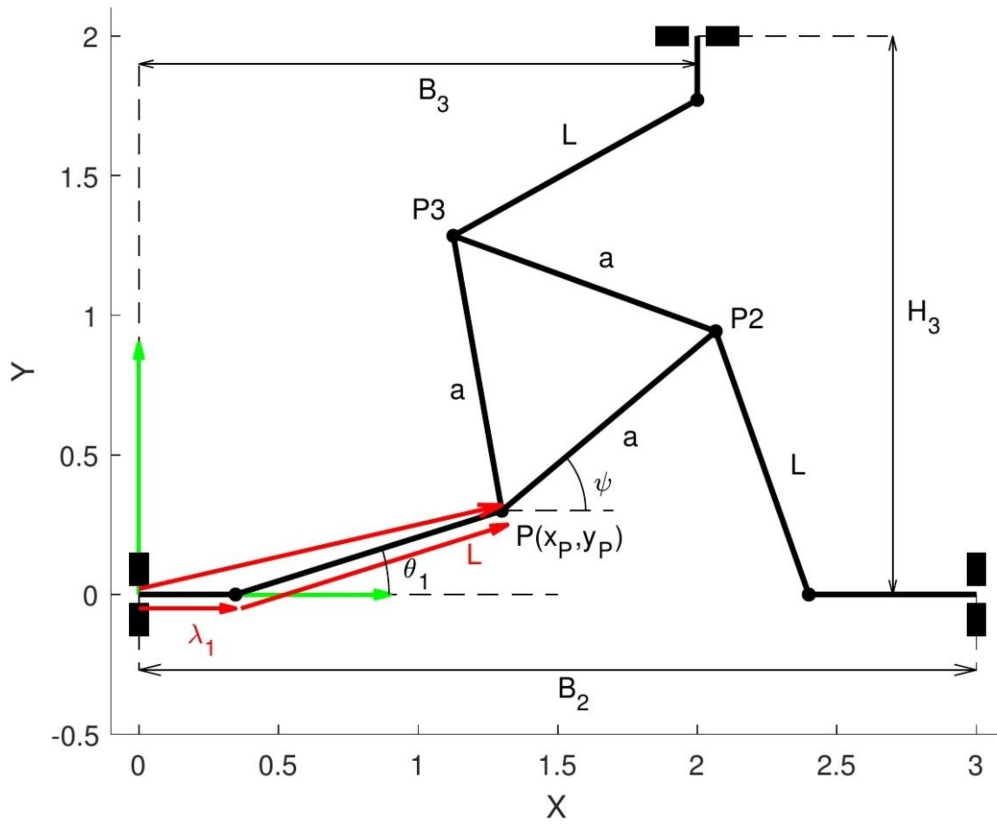


Ilustración 22. Lazo 1

$$\begin{cases} x_P = \lambda_1 + L\cos\theta_1 \\ y_P = L\sin\theta_1 \end{cases} \qquad (65)$$

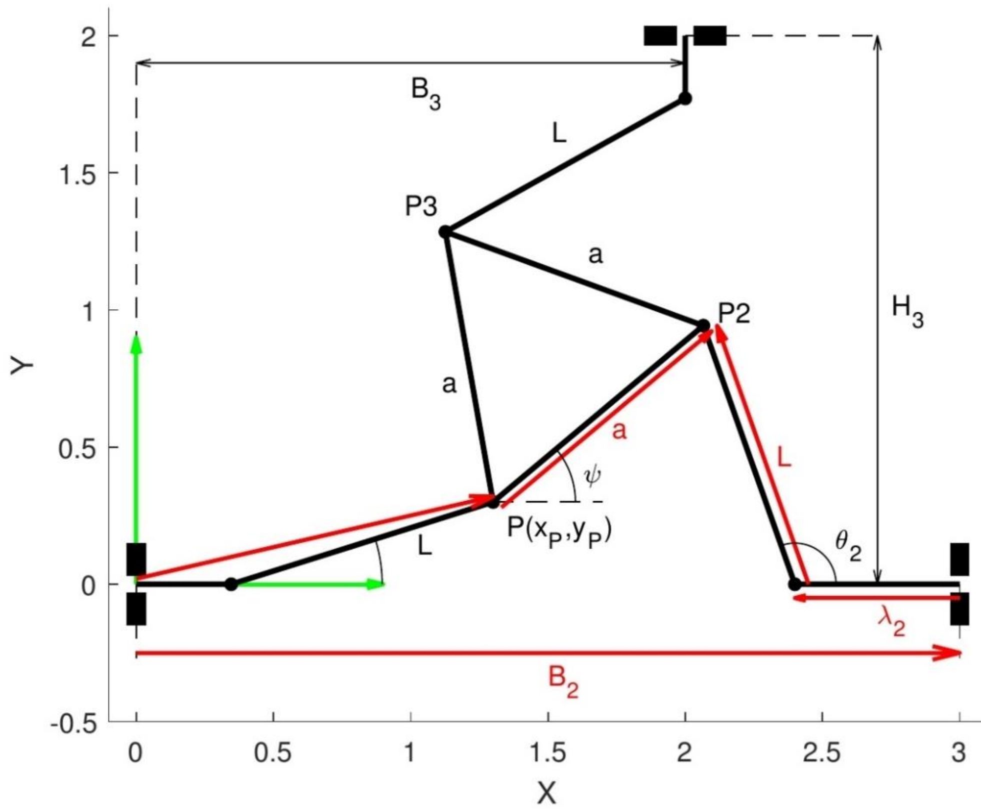


Ilustración 23. Lazo 2

$$\begin{cases} x_P = B_2 - \lambda_2 + L\cos\theta_2 - a\cos\psi \\ y_P = L\sin\theta_2 - a\sin\psi \end{cases} \quad (66)$$

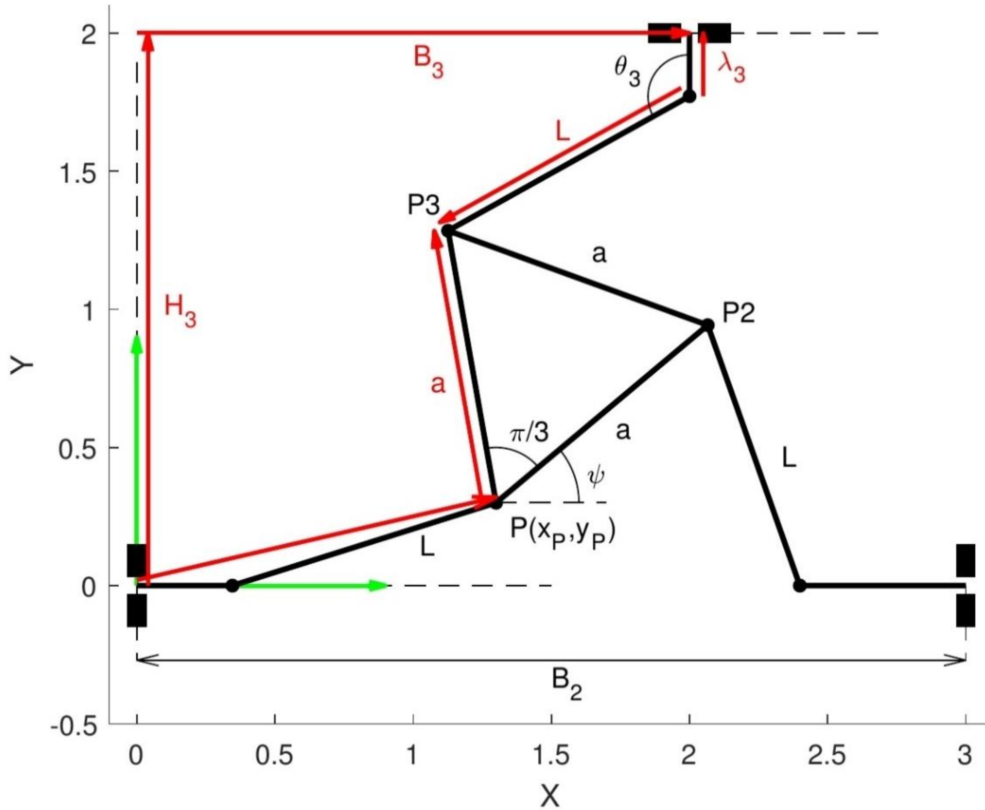


Ilustración 24. Lazo 3

$$\begin{cases} x_P = B_3 - L\sin\theta_3 - a\cos(\frac{\pi}{3} + \psi) \\ y_P = H_3 - \lambda_3 + L\cos\theta_3 - a\sin(\frac{\pi}{3} + \psi) \end{cases} \tag{67}$$

Se obtienen 6 ecuaciones con 6 incógnitas: x_P, y_P, ψ, θ_1, θ_2 y θ_3. Las variables que realmente interesan son las que definen la posición del elemento terminal, x_P, y_P y ψ, por lo que el objetivo debe ser operar para eliminar las demás. El resultado del desarrollo es el siguiente:

$$x_P = \frac{f_3(\psi)\, g_2(\psi) - f_2(\psi)\, g_3(\psi)}{f_2(\psi)\, g_1(\psi) - f_1(\psi)\, g_2(\psi)} \tag{68}$$

$$y_P = \frac{f_1(\psi)\, g_3(\psi) - f_3(\psi)\, g_1(\psi)}{f_2(\psi)\, g_1(\psi) - f_1(\psi)\, g_2(\psi)} \tag{69}$$

$$(x_P - \lambda_1)^2 + y_P^2 = L^2 \tag{70}$$

$f_1, f_2, f_3, g_1, g_2, g_3$ son funciones de ψ que vienen detalladas junto con el desarrollo matemático completo en el ANEXO I.

Al introducir las ecuaciones (68) y (69) en la ecuación (70) se obtiene un extenso polinomio en ψ igualado a 0, que se puede transformar en un polinomio en t mediante el siguiente cambio de variable:

$$t = \tan\frac{\psi}{2} \tag{71}$$

El polinomio resultante es de grado 8:

$$C_0 + C_1\, t + C_2\, t^2 + C_3\, t^3 + C_4\, t^4 + C_5\, t^5 + C_6\, t^6 + C_7\, t^7 + C_8\, t^8 = 0 \tag{72}$$

donde los coeficientes C también están detallados en el ANEXO I.

Al despejar t de la ecuación (72) se obtendrán 8 raíces, de las cuales sólo interesan las que son reales. Para deshacer el cambio de variable, resulta más sencillo, en vez de calcular directamente el valor de ψ a partir de (71), calcular su seno y su coseno, para así conocer el signo de ambos y evitar complicaciones sobre el signo o el cuadrante del ángulo resultante. Además, las funciones $f_i(\psi)$ y $g_i(\psi)$ que se han mencionado anteriormente dependen directamente del seno y el coseno de ψ.

$$\sin\psi = \frac{2t}{1+t^2} \tag{73}$$

$$\cos\psi = \frac{1-t^2}{1+t^2} \tag{74}$$

A partir de estos valores, se puede obtener fácilmente x_P, y_P según las ecuaciones que se han explicado anteriormente, y calcular el ángulo ψ como:

$$\psi = \text{atan}\frac{2t}{1-t^2} \tag{75}$$

Los valores de x_P, y_P y ψ resultantes definen la posición del elemento terminal que es solución del problema cinemático directo. Evidentemente, puede haber varias soluciones posibles, tantas como raíces t reales se hayan obtenido de la ecuación (72).

Al igual que en el problema inverso, en el directo también puede haber posiciones de singularidad. En este caso, es una posición singular aquella en la que el denominador de las expresiones de x_P, y_P, (21) y (22), se iguala a 0.

$$f_2(\psi)\, g_1(\psi) - f_1(\psi)\, g_2(\psi) = 0 \tag{76}$$

De nuevo, en la práctica resulta muy complicado que esta condición se cumpla con exactitud, así que, en lugar de igualar a 0, se considera que debe tomar un valor menor que una tolerancia previamente establecida.

11.1.3. VALIDACIÓN DE LOS CÁLCULOS

Es necesario un proceso de verificación de que los cálculos que se llevan a cabo son correctos.

En el caso del mecanismo 3PRR, se realiza en primer lugar una comprobación de que el algoritmo de cálculo del problema inverso funciona adecuadamente. Para ello, se aplica en mecanismos y

posiciones en las que, por la propia configuración geométrica, se puede conocer la solución de antemano. Es el caso de la posición que aparece en la Ilustración 25.

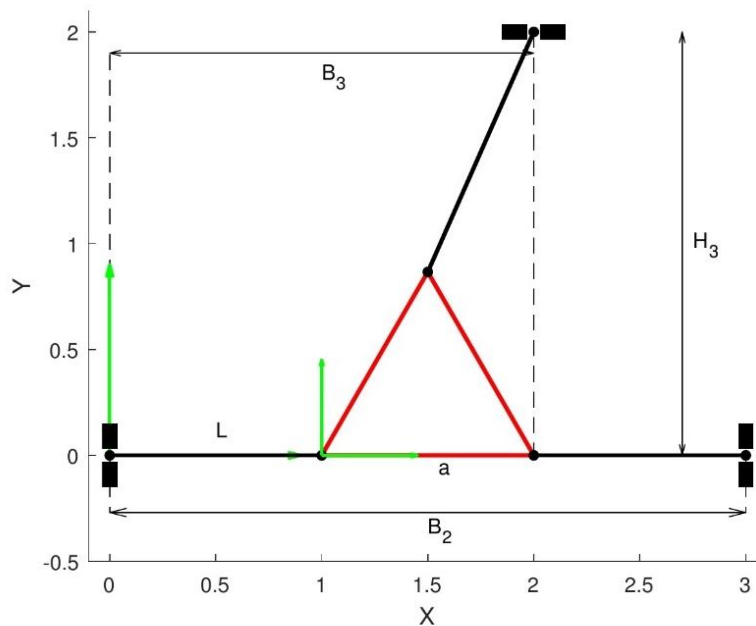


Ilustración 25. Ejemplo de posición fácilmente resoluble

Habiendo certificado que el algoritmo del problema inverso funciona correctamente, resulta sencillo comprobar el problema directo. Tan sólo hay que introducir en él las soluciones obtenidas en el inverso. El resultado debe ser las variables de entrada iniciales del problema inverso. Si tomamos unos valores de entrada al problema inverso, introducimos la solución de éste como entrada al problema directo y obtenemos como resultado los valores iniciales, prácticamente se puede afirmar que ambos algoritmos funcionan correctamente.

11.2. GENERALIDADES DEL CÁLCULO DE BARRAS FLEXIBLES

A partir de aquí, se analizan mecanismos de barras ultraflexibles. La resolución de los problemas cinemáticos de posición de este tipo de mecanismos se puede abordar de dos formas distintas, en función de cómo se resuelva el comportamiento de cada barra.

Por un lado, mediante integración numérica de las ecuaciones diferenciales que definen el comportamiento del modelo de barra de Kirchhoff en el plano, y que son las siguientes:

$$\begin{Bmatrix} p_x'(s) \\ p_y'(s) \\ \theta'(s) \\ n_x'(s) \\ n_y'(s) \\ m_z'(s) \end{Bmatrix} = \begin{Bmatrix} \cos\theta \\ \sin\theta \\ \dfrac{m_z}{EI} + u_{o,y} \\ -f_x \\ -f_y \\ n_x \sin\theta - n_y \cos\theta - l_z \end{Bmatrix} \quad (77)$$

En los casos concretos que se estudian en este proyecto, no se consideran fuerzas ni momentos aplicados a lo largo de las barras, por lo que f_x, f_y, y l_z son nulos. También se anula $u_{o,y}$ porque inicialmente las barras son totalmente rectas. El sistema que hay que resolver, por tanto, queda así:

$$\begin{Bmatrix} p_x'(s) \\ p_y'(s) \\ \theta'(s) \\ m_z'(s) \end{Bmatrix} = \begin{Bmatrix} \cos\theta \\ \sin\theta \\ \dfrac{m_z}{EI} \\ n_x \sin\theta - n_y \cos\theta \end{Bmatrix} \quad (78)$$

Es importante señalar también que, como no se consideran cargas aplicadas a lo largo de la barra, n_x y n_y son constantes.

Para la resolución por integración numérica de estas ecuaciones diferenciales se utiliza el método de Runge-Kutta de orden 4. Se trata de un método iterativo que permite integrar una función en un punto a partir de la integral calculada en el punto anterior. Sabiendo que la función que se quiere integrar es:

$$\frac{dy}{dx} = f(x,y) \quad (79)$$

Y conociendo el valor de la función $y(x)$ en el primer punto:

$$y(x_0) = y_0 \quad (80)$$

Se puede calcular el valor de la función $y(x)$ en cada punto como:

$$y_{i+1} = y_i + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4) \quad (81)$$

donde

$$k_1 = f(x_i, y_i) \quad (82)$$

$$k_2 = f(x_i + \frac{h}{2}, y_i + \frac{h}{2}k_1) \tag{83}$$

$$k_3 = f(x_i + \frac{h}{2}, y_i + \frac{h}{2}k_2) \tag{84}$$

$$k_4 = f(x_i + h, y_i + hk_3) \tag{85}$$

siendo h el paso entre puntos ($x_{i+1} = x_i + h$).

Para aplicar este método a la barra, se discretiza en toda su longitud, en n puntos separados ds. Así, conociendo el valor de p_x, p_y, θ, n_x, n_y y m_z en el primer punto, se puede calcular su valor en los puntos sucesivos hasta el extremo de la barra, que es donde realmente interesa conocer la posición, orientación y esfuerzos a los que está sometida. Es importante señalar que es imprescindible conocer los datos en ese primer punto para llevar a cabo todo el cálculo en el resto de la barra.

La otra forma de resolver el comportamiento de una barra flexible es mediante la utilización de integrales elípticas. La integración directa de las ecuaciones del sistema (77), lleva a la obtención de las siguientes expresiones para las coordenadas x e y que definen la forma de la barra deformada:

$$x = -\sqrt{\frac{EI}{R}}\cos\psi\,[2E(k,\phi_i) - 2E(k,\phi_1) - F(k,\phi_i) + F(k,\phi_1)] - \\ -\sqrt{\frac{EI}{R}}2k\sin\psi\,[\cos\phi_i - \cos\phi_1] \tag{86}$$

$$y = -\sqrt{\frac{EI}{R}}\sin\psi\,[2E(k,\phi_i) - 2E(k,\phi_1) - F(k,\phi_i) + F(k,\phi_1)] + \\ +\sqrt{\frac{EI}{R}}2k\cos\psi\,[\cos\phi_i - \cos\phi_1] \tag{87}$$

Donde $F(k,\phi)$ y $E(k,\phi)$ son las integrales elípticas de primer y segundo orden, cuyos valores están tabulados.

En principio, todas las barras de los mecanismos que se analizan son empotradas en uno de sus extremos y articuladas en el otro. Además, se utiliza un sistema de referencia local, colocado de manera que el empotramiento sea horizontal, y todos los cálculos de cada barra por separado se llevan a cabo en ese sistema asociado a ella.

Los parámetros k y ψ definen completamente el comportamiento de la barra, y permiten obtener todos los demás datos. R se calcula según la siguiente ecuación:

$$\sqrt{R} = \frac{\sqrt{EI}}{L}[F(k,\phi_2) - F(k,\phi_1)] \tag{88}$$

Sabiendo que el extremo inicial es empotrado y el otro extremo, articulado,

$$\phi_1 = \arcsin \frac{1}{k} \cos \frac{\psi}{2} \tag{89}$$

$$\phi_2 = \frac{q\pi}{2} \qquad \text{donde } q = 1,3, \dots \tag{90}$$

Con todos estos datos, k, ψ, R, ϕ_1 y ϕ_2, introduciéndolos en las ecuaciones (86) y (87)se puede calcular la posición del extremo articulado de la barra, que es el dato que más valor tiene de cara a la resolución de los problemas de posición que se van a tratar.

En los procedimientos que se van a utilizar, hay que discretizar la variable k en todo su rango. Para ello, es más útil y simplifica las cosas el empleo de una k_{rel}, puesto que tiene un rango de existencia continuo entre [-1, 1]. Se discretizará, por tanto, k_{rel} en vez de k, y para cada valor de la primera se calculará el de la segunda como:

$$k = Signo(k_{rel})k_{min} + (1 - k_{min})k_{rel} \tag{91}$$

11.3. 2RFR

El mecanismo que es objeto de estudio en esta parte del proyecto es un mecanismo de 2 grados de libertad formado por dos barras flexibles que están empotradas en uno de sus extremos a un motor de rotación, que determina su inclinación, y en el otro, articuladas entre sí. El extremo común es el elemento terminal del mecanismo, cuya posición es la salida del sistema. La entrada es la posición angular de los motores, definida por θ_1 y θ_2. En este caso, al tratarse de barras flexibles, las cargas externas toman una gran importancia en la configuración del mecanismo. Se considera una fuerza externa aplicada sobre el extremo de las barras. En general, se conocen los puntos donde se sitúan los motores, O_1 y O_2, y las dimensiones y propiedades resistentes de las barras.

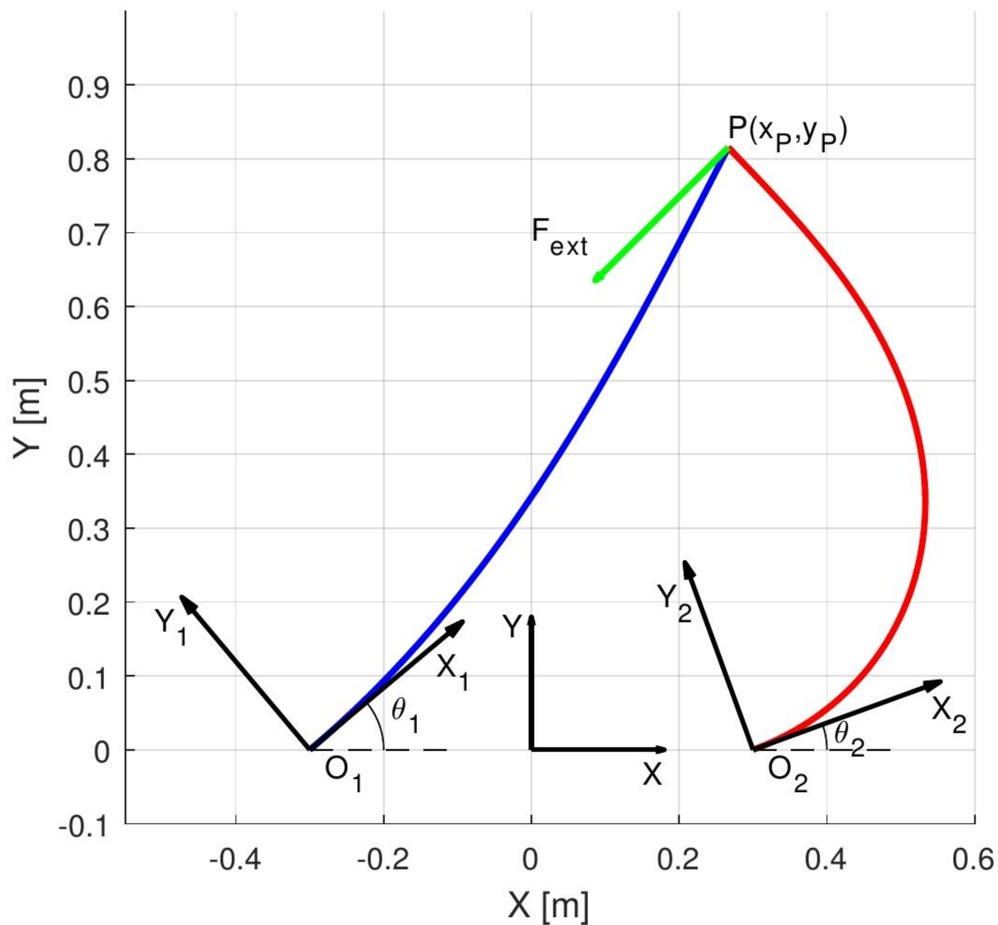


Ilustración 26. Mecanismo 2RFR

11.3.1. INTEGRACIÓN NUMÉRICA

En primer lugar, se lleva a cabo la resolución mediante la integración numérica del sistema de ecuaciones (78), tanto del problema directo como del inverso. Es importante señalar que, en este caso, el análisis que se lleva a cabo no es simplemente la resolución del problema en una única posición, sino un recorrido a lo largo de una trayectoria en el espacio de trabajo del mecanismo. El cálculo en cada punto de la trayectoria requiere conocer los datos del punto

anterior para utilizarlos como aproximación, por lo que es necesario partir en primera instancia de una posición donde todas las variables del problema sean conocidas.

11.3.1.1. PROBLEMA INVERSO

La resolución del problema inverso en este caso consiste en determinar qué posiciones de los motores, θ_1 y θ_2, son necesarias para que el elemento terminal llegue a una posición dada por (x_P, y_P), conocidos.

Como se ha indicado, el análisis se lleva a cabo a través de una trayectoria, en este caso en el espacio (x_P, y_P), y se resuelve el problema inverso en cada punto de la misma. Se plantean fundamentalmente tres tipos de trayectorias: recta, circular concéntrica y rectangular concéntrica.

Como requisitos fundamentales en la resolución, hay que tener en cuenta que ambas barras deben coincidir en su extremo, que además tiene que estar en el punto (x_P, y_P) que se ha dado inicialmente como dato del problema inverso; que ambas deben tener un momento nulo en ese extremo; y que debe haber un equilibrio de fuerzas en él. Se considera que en la discretización de las barras el punto inicial, 0, está en el extremo empotrado el motor y el punto final, n, está en el extremo articulado. Estas condiciones se traducen en un sistema de ecuaciones que se debe resolver para obtener la solución del problema inverso.

$$\begin{cases} m_{z,n}^1 = 0 \\ p_{x,n}^1 = x_P \\ p_{y,n}^1 = y_P \\ m_{z,n}^2 = 0 \\ p_{x,n}^2 = x_P \\ p_{y,n}^2 = y_P \\ n_x^1 + n_x^2 - F_x^{ext} = 0 \\ n_y^1 + n_y^2 - F_y^{ext} = 0 \end{cases} \qquad (92)$$

Para calcular la posición y los momentos de cada barra es necesario resolver el sistema de ecuaciones (78), tal y como se ha explicado anteriormente. El problema es que, para llevar a cabo esta operación, es necesario conocer el valor de p_x, p_y, θ, n_x, n_y y m_z en el extremo empotrado de cada barra, de los cuales, en principio, sólo p_x y p_y son conocidos. Por eso, se toma como aproximación inicial los datos del punto anterior de la trayectoria y se aplica un método de cálculo iterativo.

El proceso de cálculo del mecanismo en cada punto se realiza mediante el método de Newton-Raphson. Se trata de un método iterativo que permite encontrar las raíces de una función a partir de una aproximación inicial. Para un sistema de n funciones y n incógnitas, como es este caso,

$$\begin{cases} f_1(x_1, \dots, x_n) = f_1(\boldsymbol{X}) = 0 \\ \dots \\ f_n(x_1, \dots, x_n) = f_n(\boldsymbol{X}) = 0 \end{cases} \qquad (93)$$

que se puede expresar de forma compacta como vector:

$$\boldsymbol{f}(\boldsymbol{X}) = \{0\} \tag{94}$$

donde la primera aproximación es conocida,

$$\boldsymbol{f}(\boldsymbol{X}_0) = \boldsymbol{f}_0 \tag{95}$$

Cada iteración da un nuevo valor de la incógnita que es

$$\boldsymbol{X}_{i+1} = \boldsymbol{X}_i - J^{-1}(\boldsymbol{X}_i)\,\boldsymbol{f}(\boldsymbol{X}_i) \tag{96}$$

donde J es el jacobiano del sistema de ecuaciones:

$$J = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1} & \cdots & \frac{\partial f_n}{\partial x_n} \end{bmatrix} \tag{97}$$

El residuo en cada iteración es el vector $\boldsymbol{f}(\boldsymbol{X}_i)$, cuyos elementos se irán aproximando a 0 si el algoritmo converge.

En el caso que es objeto de análisis en esta parte del proyecto, el sistema está formado por 8 funciones de las cuales se desea obtener sus raíces. Estas funciones son las que se corresponden con la posición del extremo de cada barra, el momento en ese extremo y el balance de fuerzas, igual que en el sistema (92).

$$\begin{cases} f_1 = m_{z,n}^1 = 0 \\ f_2 = p_{x,n}^1 - x_P = 0 \\ f_3 = p_{y,n}^1 - y_P = 0 \\ f_4 = m_{z,n}^2 = 0 \\ f_5 = p_{x,n}^2 - x_P = 0 \\ f_6 = p_{y,n}^2 - y_P = 0 \\ f_7 = n_x^1 + n_x^2 - F_x^{ext} = 0 \\ f_8 = n_y^1 + n_y^2 - F_y^{ext} = 0 \end{cases} \tag{98}$$

El vector de incógnitas está formado por el momento, componentes de la fuerza y ángulo de orientación de cada una de las barras en sus extremos empotrados:

$$\boldsymbol{X}=\begin{bmatrix} m_{z,0}^{1} \\ n_{x,0}^{1} \\ n_{y,0}^{1} \\ \theta_{0}^{1} \\ m_{z,0}^{2} \\ n_{x,0}^{2} \\ n_{y,0}^{2} \\ \theta_{0}^{2} \end{bmatrix} \qquad (99)$$

El resultado que se alcanza mediante Newton-Raphson depende de la primera aproximación que se tome. En función de esto, el algoritmo puede converger a una solución o a otra, o no converger. Es por esto que, como ya se ha indicado, se toma como aproximación inicial los datos correspondientes a la solución del problema inverso del punto anterior de la trayectoria. Para que estas aproximaciones sean buenas, es importante que las variaciones de x_P, y_P sean pequeñas. De esta forma, al ser posiciones cercanas, se entiende que se asegura la convergencia del método a una solución compatible con la anterior.

Para el cálculo del jacobiano del sistema, las derivadas parciales de cada función se llevan a cabo mediante la definición de derivada, calculando la variación de cada función al incrementar cada variable en una pequeña cantidad ε, y dividiendo esa variación entre ε. Por ejemplo, al calcular la derivada parcial de f_1 respecto de $m_{z,0}^{1}$:

$$\frac{\partial f_1}{\partial m_{z,0}^{1}}=\frac{f_1\left(m_{z,0}^{1}+\varepsilon, n_{x,0}^{1}, n_{y,0}^{1}, \theta_0^1, m_{z,0}^{2}, n_{x,0}^{2}, n_{y,0}^{2}, \theta_0^2\right)}{\varepsilon}-\\ -\frac{f_1(m_{z,0}^{1}, n_{x,0}^{1}, n_{y,0}^{1}, \theta_0^1, m_{z,0}^{2}, n_{x,0}^{2}, n_{y,0}^{2}, \theta_0^2)}{\varepsilon} \qquad (100)$$

En cada iteración, por tanto, se va actualizando las variables según la ecuación (96) y se calcula un residuo que es el resultado de evaluar las funciones (98) en las variables obtenidas.

$$Residuo=\boldsymbol{f}(\boldsymbol{X}) \qquad (101)$$

Se considera que se ha alcanzado la solución cuando todos los elementos del vector de residuo son menores que una tolerancia dada. En este caso el algoritmo converge, pero no siempre tiene por qué ser así. El proceso de cálculo acaba cuando se alcanza la solución o cuando se detecta que el algoritmo no converge.

De todos los resultados que se obtienen, son de especial interés los ángulos de posición de los motores y los momentos que éstos tienen que aplicar sobre las barras, que se corresponden con los ángulos de inclinación y los momentos en las barras en su extremo empotrado, además del determinante del jacobiano del problema inverso, que es el jacobiano del sistema (98) respecto de las incógnitas (99). Este determinante se utiliza para identificar las posiciones de singularidad del problema inverso. Cuando el determinante es igual a 0, $|J_{IK}| = 0$, significa que se trata de una posición singular.

Una vez se ha recorrido toda la trayectoria y realizado todos los cálculos, es interesante representar gráficamente los resultados. Se lleva a cabo una representación de los valores de

los ángulos de posición de los motores, θ_1 y θ_2, y del determinante del jacobiano del problema inverso en cada punto de la trayectoria. En las siguientes figuras se puede ver un ejemplo de las gráficas resultantes en el caso de una trayectoria de circunferencias concéntricas.

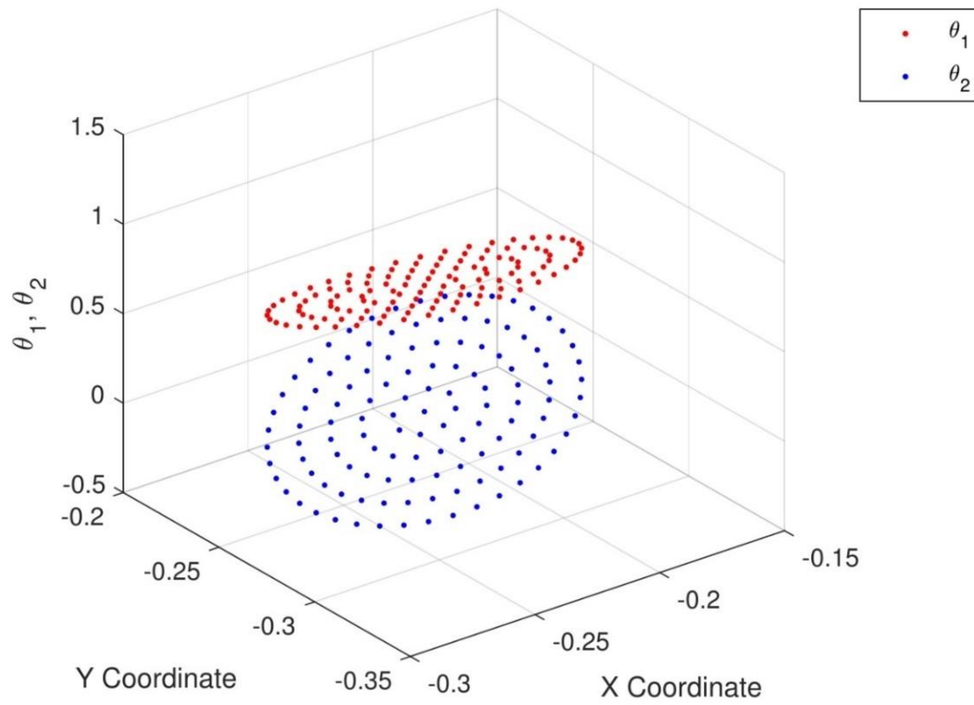


Ilustración 27. Representación de la posición angular de los motores en función de (x,y)

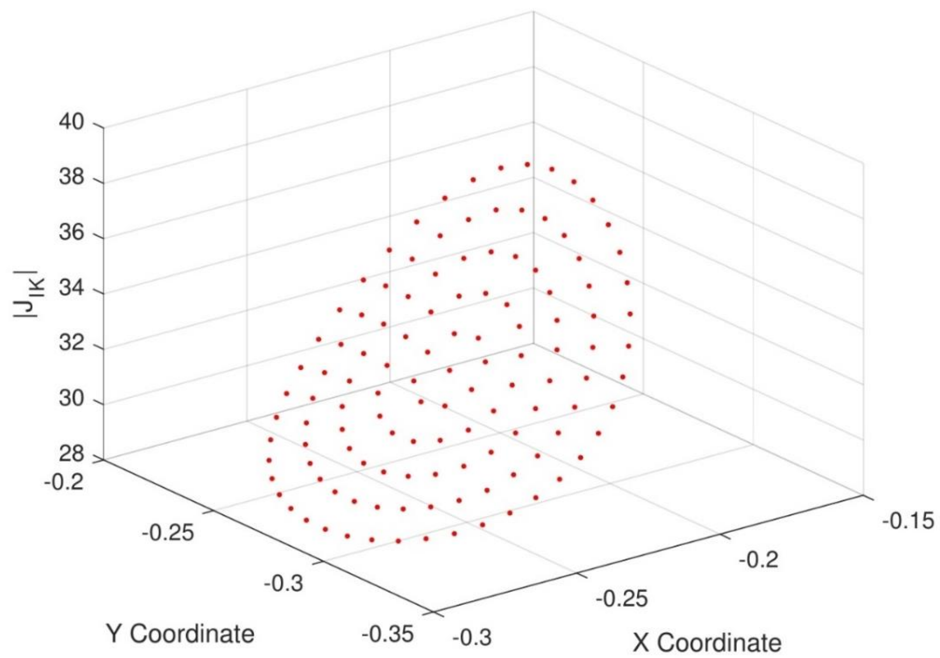


Ilustración 28. Determinante del jacobiano del problema inverso en función de (x,y)

11.3.1.2. PROBLEMA DIRECTO

En el problema directo, se trata de determinar a qué posición llega el extremo terminal del mecanismo para una posición angular concreta de los motores, definida por θ_1, θ_2 conocidos. Igual que antes, se resuelve el problema recorriendo una trayectoria, en este caso en el plano (θ_1, θ_2).

El proceso de resolución es muy similar al anterior, y las condiciones que debe cumplir el mecanismo son las mismas: los extremos de ambas barras deben estar situados en la misma posición (x_P, y_P), el momento de las barras en ellos debe ser nulo y debe haber equilibrio de fuerzas en ese punto terminal. Todo esto se traduce en el mismo sistema de ecuaciones (98). La diferencia en este caso es que las posiciones angulares de los motores, θ_1 y θ_2, son conocidas, y la posición del elemento terminal, x_P, y_P, no lo es. Por tanto, el vector de incógnitas cambia:

$$\boldsymbol{X} = \begin{bmatrix} m_{z,0}^1 \\ n_{x,0}^1 \\ n_{y,0}^1 \\ m_{z,0}^2 \\ n_{x,0}^2 \\ n_{y,0}^2 \\ x_P \\ y_P \end{bmatrix} \qquad (102)$$

Por lo demás, el método de Newton-Raphson se aplica tal y como se ha explicado en el apartado anterior.

De todos los cálculos realizados, los resultados que tienen un mayor interés son la posición del elemento terminal, (x_P, y_P), que es la solución del problema directo, los momentos que deben aplicar los motores en cada posición del mecanismo y el determinante del jacobiano del problema directo, que es el jacobiano del sistema (98) respecto de las incógnitas (102).

Se representan los resultados gráficamente en cada puno de la trayectoria seguida en el espacio (θ_1, θ_2). A continuación, se ve un ejemplo para una trayectoria circular concéntrica.

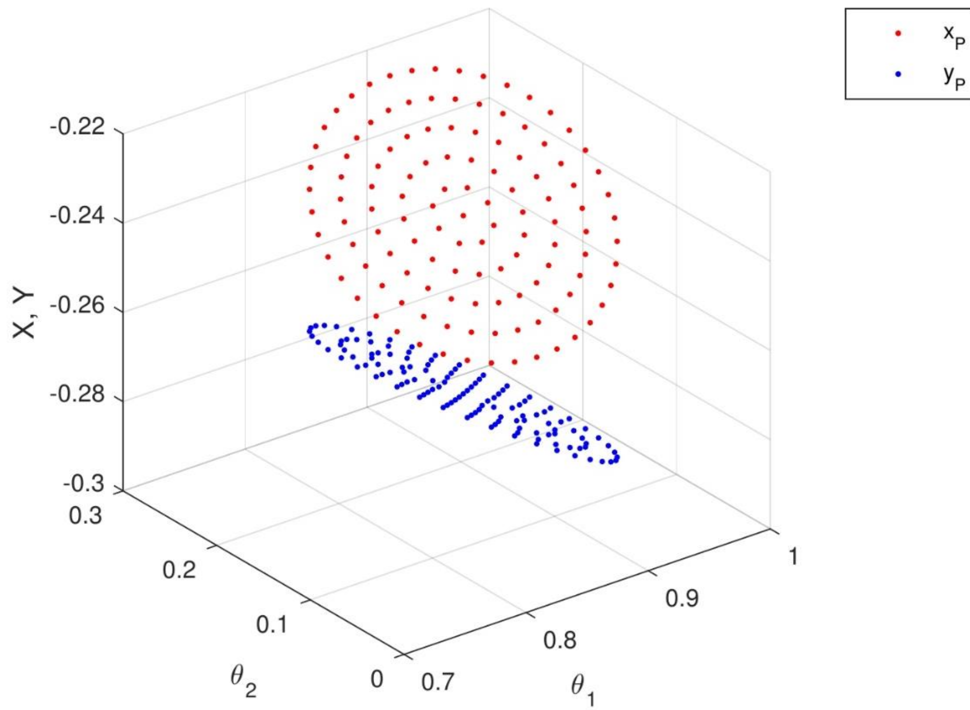


Ilustración 29. Representación de las coordenadas de P en función de (ϑ_1,ϑ_2)

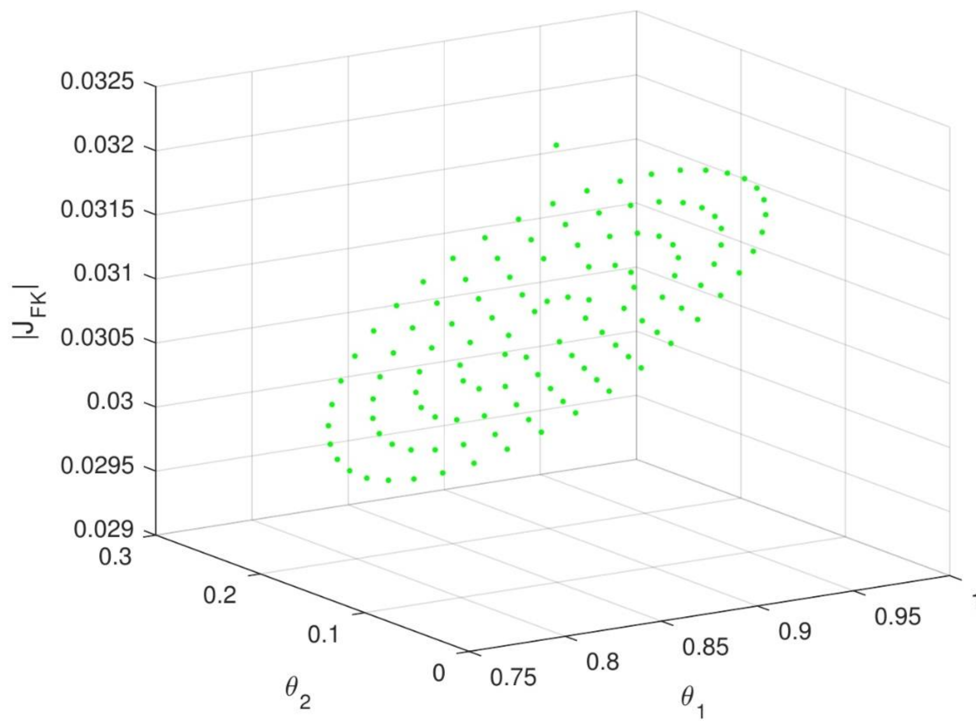


Ilustración 30. Determinante del jacobiano del problema directo en función de (ϑ_1,ϑ_2)

No obstante, quizá sea más intuitivo y visual representar los datos en función de la posición del elemento terminal. Evidentemente, la forma de la trayectoria cambia al pasar del plano (θ_1, θ_2) al (x_P, y_P). Una trayectoria circular en (θ_1, θ_2), por ejemplo, se transforma en una forma elíptica en (x_P, y_P), como se ve a continuación.

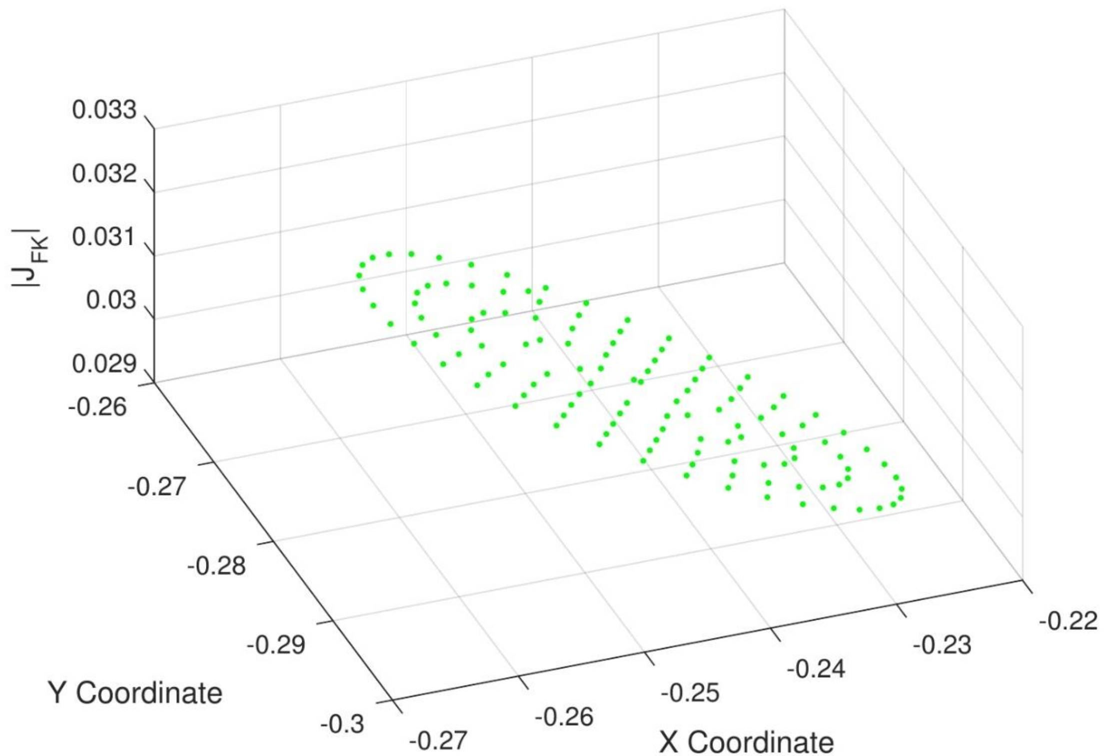


Ilustración 31. Determinante del jacobiano del problema directo en función de (x,y)

11.3.2. INTEGRALES ELÍPTICAS. PROBLEMA DIRECTO

El objetivo es, de nuevo, determinar la posición del punto P conociendo la inclinación de las barras en sus extremos empotrados. Se consideran dos sistemas de referencia locales unidos a las dos barras del mecanismo, como se ve en la Ilustración 32. Los cálculos de cada una de las barras se llevan a cabo en estos sistemas de referencia, por lo que es necesario definir las matrices de rotación de cada uno de ellos, $\boldsymbol{Rot}_1$ y $\boldsymbol{Rot}_2$. Se conocen las características geométricas y físicas del mecanismo. Además, en este caso es necesario conocer previamente el modo de pandeo de cada una de las barras, puesto que va a determinar el valor de q para el cálculo de ϕ_2. Aunque en la imagen no aparezca explícitamente, se considera también la posibilidad de que haya una fuerza externa aplicada sobre el elemento terminal P.

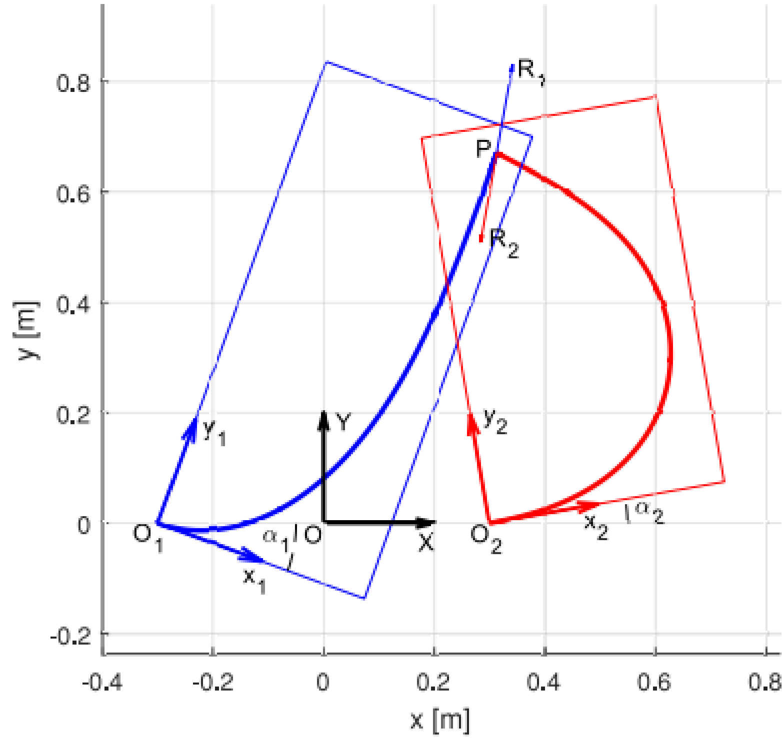


Ilustración 32. Sistemas de referencia locales en el mecanismo 2RFR

Cada barra queda perfectamente definida mediante los parámetros k y ψ. Por tanto, hay que buscar qué valores deben tomar éstos en cada una de las dos barras para que el mecanismo funcione.

Para ello, se discretizan las variables k_1 y ψ_1 de la primera barra y k_2 de la segunda en todo su rango de existencia. Como el rango de existencia de la variable k para cada ψ no es continuo, se utiliza en su lugar una k_{rel}, que tiene un rango continuo en [-1, 1]. Es lo que se puede apreciar en la Ilustración 8. La discretización de ψ se puede hacer lineal; con la de k_{rel}, en cambio, es mejor tomar más puntos cerca del -1 y el 1, por lo que se escoge una discretización exponencial.

Con cada combinación de valores de k_{rel}^1, ψ_1 y k_{rel}^2, se comprueba si pueden ser solución del problema directo. En primer lugar, hay que obtener los valores de k_1 y k_2 correspondientes mediante la ecuación (91). Con k_1 y ψ_1, se puede calcular el valor de R_1 utilizando las ecuaciones (88), (89) y (90). Hay que recordar que, en general, a cada pareja de valores de k_{rel} y ψ les corresponde un único valor de R, como se ha visto en la Ilustración 10. Con ello, se puede obtener la posición del extremo de la barra 1 en su sistema de referencia local, $(x_{1,loc}, y_{1,loc})$, aplicando (86) y (87). Es necesario pasar estas coordenadas al sistema de referencia global mediante la matriz de rotación $\boldsymbol{Rot}_1$ y la posición del punto O_1.

$$\begin{bmatrix} x_1 \\ y_1 \end{bmatrix} = \begin{bmatrix} x_{O1} \\ y_{O1} \end{bmatrix} + \boldsymbol{Rot}_1 \begin{bmatrix} x_{1,loc} \\ y_{1,loc} \end{bmatrix} \quad (103)$$

Debe haber equilibrio de fuerzas en el punto P. Expresado matemáticamente,

$$\boldsymbol{Rot}_1 \begin{bmatrix} R_1 \cos\psi_1 \\ R_1 \sin\psi_1 \end{bmatrix} + \boldsymbol{Rot}_2 \begin{bmatrix} R_2 \cos\psi_2 \\ R_2 \sin\psi_2 \end{bmatrix} + \begin{bmatrix} F_x^{ext} \\ F_y^{ext} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \quad (104)$$

De esta expresión se puede despejar el valor de ψ_2 y R_2 a partir de R_1 y ψ_1. Y con este valor de ψ_2 y k_2 obtenemos otro valor de R_2, al que llamamos R_2', además de x_2, y_2, igual que se ha hecho con la barra 1.

Una vez se han calculado todos los datos de ambas barras, hay que comprobar si realmente pueden ser solución del problema. El mecanismo debe cumplir que los extremos de las barras estén en el mismo punto y que en ese punto haya equilibrio de fuerzas. Se definen, por tanto, tres residuos:

$$\begin{cases} Res_x = x_1 - x_2 \\ Res_y = y_1 - y_2 \\ Res_F = R_2 - R_2' \end{cases} \qquad (105)$$

En principio, se considera que la solución es válida si los tres residuos son menores que una tolerancia previamente establecida. Se toma una tolerancia no demasiado pequeña, mayor cuanto más grosera haya sido la discretización inicial, para no desechar soluciones que puedan ser válidas.

Se refina la solución en un bucle de Newton-Raphson, en el que el sistema a resolver es de 4 ecuaciones (2 ecuaciones de la unión de los extremos en el punto P y 2 ecuaciones del equilibrio de fuerzas) y 4 incógnitas (k_{rel} y ψ en cada barra). De esta forma, se comprueba si la solución obtenida inicialmente es realmente válida o no, y en caso de que lo sea, se calcula un valor más exacto.

Como se puede ver, este procedimiento puede dar varias soluciones posibles, aunque no tiene por qué encontrar todas.

En este caso, k_{rel} y ψ ya definen totalmente las dos barras, por lo que podrían considerarse como la solución final del problema.

11.3.3. VALIDACIÓN DE LOS CÁLCULOS

Para comprobar que los cálculos son correctos, se aplican en mecanismos con configuraciones simétricas y sin carga, donde intuitivamente se puede deducir cuál es la solución de los problemas. Se aplica al problema directo, tanto al resuelto por integración numérica como por integrales elípticas, aunque es menos eficaz en el primer caso, donde la solución que se alcanza es única, depende de la aproximación inicial y no tiene por qué coincidir con la posición de referencia. Un ejemplo de una posición en la que se aplica esta idea es el que aparece en la Ilustración 33, donde aparece un mecanismo con configuración simétrica.

Además, se comprueban entre sí las soluciones de los diferentes programas. Los dos problemas directos, tanto el que realiza el cálculo por integración numérica como el que utiliza integrales elípticas deben llegar a una misma solución común. Además, al introducir esta solución en el problema inverso, el resultado debe ser la entrada de esos problemas directos.

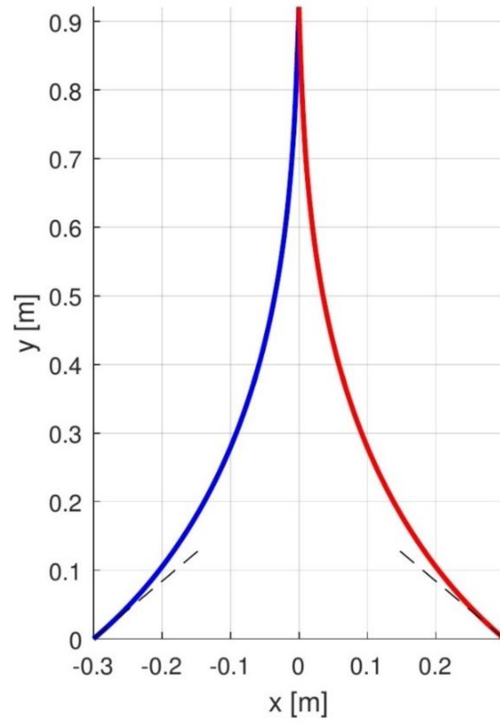


Ilustración 33. Mecanismo 2RFR simétrico

11.4. 3PFR

El mecanismo que se analiza en este apartado es el 3PFR. Se trata de un mecanismo de 3 grados de libertad que consta de 3 barras flexibles que tienen uno de sus extremos empotrado a una deslizadera que se desplaza sobre una guía lineal y el otro articulado a una plataforma triangular que es el elemento terminal del mecanismo. Se puede ver en la siguiente imagen:

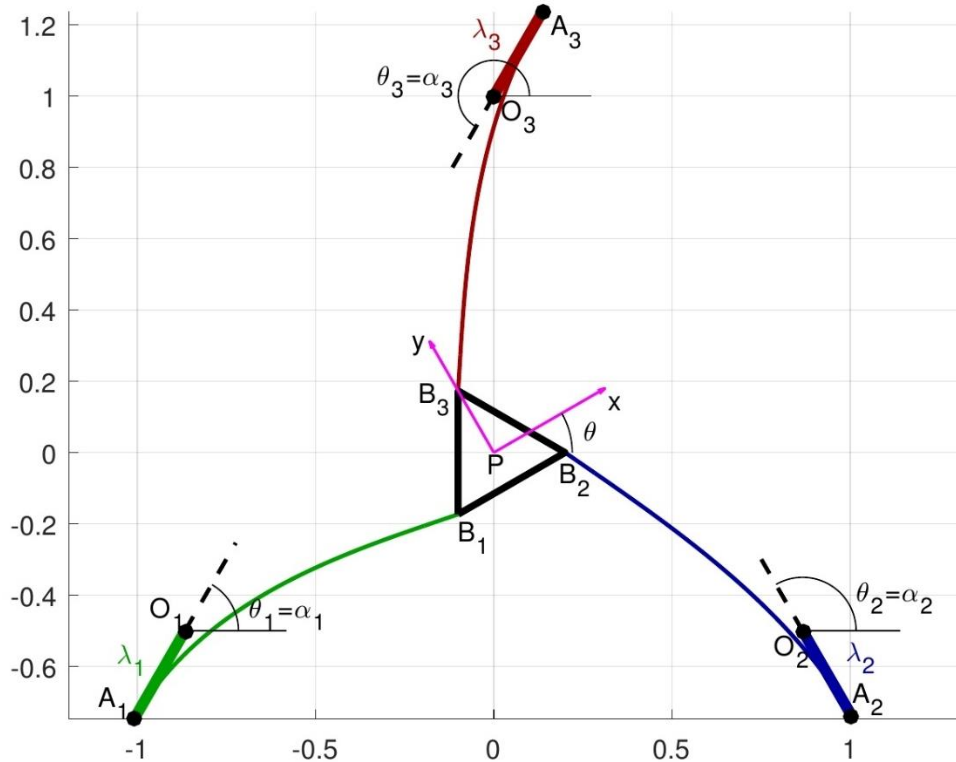


Ilustración 34. Mecanismo 3PFR

La posición del elemento terminal, que es un triángulo equilátero, está definida por las coordenadas del punto P (x_P, y_P) y el ángulo de inclinación θ. Las guías lineales pasan por los puntos O_1, O_2 y O_3 y tienen una inclinación α. En la imagen, la inclinación de las barras en el extremo empotrado a la deslizadera, θ_1, θ_2, θ_3, coincide con la inclinación de las guías, pero esto es sólo un caso particular, no tiene por qué ser así siempre. La posición de los motores que actúan sobre las deslizaderas viene dada por las longitudes λ. Se conocen todas las propiedades geométricas y resistentes de las barras. Además, se considera la posibilidad de que haya una fuerza y un momento externos aplicados sobre la plataforma.

Al igual que en el mecanismo 2RFR, la resolución de los problemas de posición se lleva a cabo de dos formas distintas: por integración numérica y por integrales elípticas.

11.4.1. INTEGRACIÓN NUMÉRICA

La resolución se lleva a cabo de la misma forma que en el mecanismo anterior, resolviendo los problemas inverso y directo en una sucesión de posiciones por el método de Newton-Raphson. Puesto que el proceso de cálculo en cada punto requiere utilizar como aproximación los datos del punto anterior, es necesario conocer una posición inicial donde todos los parámetros sean

conocidos y de donde parta la trayectoria. Hay que tener en cuenta que, al ser un mecanismo de 3 grados de libertad, cada posición está definida por 3 parámetros. Esto implica, por ejemplo, que ya no se puede concebir simplemente una trayectoria plana del elemento terminal en (x, y), sino que también hay que considerar la orientación del mismo. Las trayectorias que se consideran son en un espacio tridimensional.

11.4.1.1. PROBLEMA INVERSO

La posición del elemento terminal es conocida y se trata de determinar qué posiciones de los motores lineales llevan al mecanismo a esa posición.

Como se ve en la Ilustración 34, se utiliza un sistema de referencia local unido a la plataforma del mecanismo, que permite determinar su posición en todo momento. Como todas las dimensiones del mecanismo son conocidas, se conoce también la posición de los vértices de la plataforma en ese sistema de referencia local. Y se puede obtener las coordenadas de cada vértice en un sistema de referencia global como:

$$\begin{bmatrix} x_{B1} \\ y_{B1} \end{bmatrix} = \begin{bmatrix} x_P \\ y_P \end{bmatrix} + \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x_{B1,loc} \\ y_{B1,loc} \end{bmatrix} \quad (106)$$

$$\begin{bmatrix} x_{B2} \\ y_{B2} \end{bmatrix} = \begin{bmatrix} x_P \\ y_P \end{bmatrix} + \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x_{B2,loc} \\ y_{B2,loc} \end{bmatrix} \quad (107)$$

$$\begin{bmatrix} x_{B3} \\ y_{B3} \end{bmatrix} = \begin{bmatrix} x_P \\ y_P \end{bmatrix} + \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x_{B3,loc} \\ y_{B3,loc} \end{bmatrix} \quad (108)$$

El mecanismo debe cumplir que la posición de los extremos articulados de las barras coincidan con las coordenadas de los puntos B calculados en (106), (107) y (108), que el momento interno de las barras en ese punto sea nulo y que haya equilibrio de fuerzas y momentos en la plataforma triangular. Estas condiciones se pueden expresar matemáticamente así:

$$\begin{cases} m_{z,n}^1 = 0 \\ p_{x,n}^1 = x_{B1} \\ p_{y,n}^1 = y_{B1} \\ m_{z,n}^2 = 0 \\ p_{x,n}^2 = x_{B2} \\ p_{y,n}^2 = y_{B2} \\ m_{z,n}^3 = 0 \\ p_{x,n}^3 = x_{B3} \\ p_{y,n}^3 = y_{B3} \\ n_x^1 + n_x^2 + n_x^3 - F_x^{ext} = 0 \\ n_y^1 + n_y^2 + n_y^3 - F_y^{ext} = 0 \\ p_{x,n}^1\, n_y^1 - p_{y,n}^1\, n_x^1 + p_{x,n}^1\, n_y^1 - p_{y,n}^1\, n_x^1 + p_{x,n}^1\, n_y^1 - \\ \quad -p_{y,n}^1\, n_x^1 - x_P\, F_y^{ext} + y_P\, F_x^{ext} - M_{ext} = 0 \end{cases} \quad (109)$$

Por lo tanto, el sistema a resolver por el método de Newton-Raphson es de 12 ecuaciones con 12 incógnitas:

$$
\begin{cases}
f_1 = m_{z,n}^1 = 0 \\
f_2 = p_{x,n}^1 - x_{B1} = 0 \\
f_3 = p_{y,n}^1 - y_{B1} = 0 \\
f_4 = m_{z,n}^2 = 0 \\
f_5 = p_{x,n}^2 - x_{B2} = 0 \\
f_6 = p_{y,n}^2 - y_{B2} = 0 \\
f_7 = m_{z,n}^3 = 0 \\
f_8 = p_{x,n}^3 - x_{B3} = 0 \\
f_9 = p_{y,n}^3 - y_{B3} = 0 \\
f_{10} = n_x^1 + n_x^2 + n_x^3 - F_x^{ext} = 0 \\
f_{11} = n_y^1 + n_y^2 + n_y^3 - F_y^{ext} = 0 \\
f_{12} = p_{x,n}^1\, n_y^1 - p_{y,n}^1\, n_x^1 + p_{x,n}^1\, n_y^1 - p_{y,n}^1\, n_x^1 + p_{x,n}^1\, n_y^1 - \\
-p_{y,n}^1\, n_x^1 - x_P\, F_y^{ext} + y_P\, F_x^{ext} - M_{ext} = 0
\end{cases} \qquad (110)
$$

Las incógnitas en este caso son los momentos en el extremo empotrado, las componentes de las fuerzas internas y las longitudes λ de las tres barras. El vector de incógnitas, por tanto, es el siguiente:

$$
\boldsymbol{X} = \begin{bmatrix} m_{z,0}^1 \\ n_x^1 \\ n_y^1 \\ \lambda_1 \\ m_{z,0}^2 \\ n_x^2 \\ n_y^2 \\ \lambda_2 \\ m_{z,0}^2 \\ n_x^3 \\ n_y^3 \\ \lambda_3 \end{bmatrix} \qquad (111)
$$

Como se ha dicho, se resuelve el sistema por Newton-Raphson en una sucesión de posiciones del mecanismo, definidas por (x_P, y_P, θ), es decir, se sigue una trayectoria en el espacio (x_P, y_P, θ). En cada punto, se utilizan los datos del anterior como primera aproximación del método. Por eso, para que esa primera aproximación sea buena, es importante que las variaciones de x_P, y_P y θ no sean demasiado grandes de un punto a otro.

De esta manera, se obtienen los valores de λ_1, λ_2 y λ_3 que son la solución del problema inverso. También es interesante conocer la fuerza que debe aplicar cada motor lineal sobre la barra, que sería la proyección de la fuerza interna de la barra en el empotramiento sobre la dirección de la guía lineal; y el determinante del jacobiano del problema inverso, que es el jacobiano del sistema (110) respecto de las variables del vector (111).

En este caso, la representación gráfica de los resultados no es tan sencilla como en el mecanismo anterior de 2 grados de libertad. Sí se podría llevar a cabo de la misma forma si, por ejemplo, se considera una trayectoria con una orientación θ constante. En ese caso, el espacio (x_P, y_P, θ)

se reduce a un plano (x_P, y_P) y se pueden representar los valores de λ_1, λ_2 y λ_3 o el determinante del jacobiano, $|J_{IK}|$, en cada punto, como se ve en las siguientes imágenes:

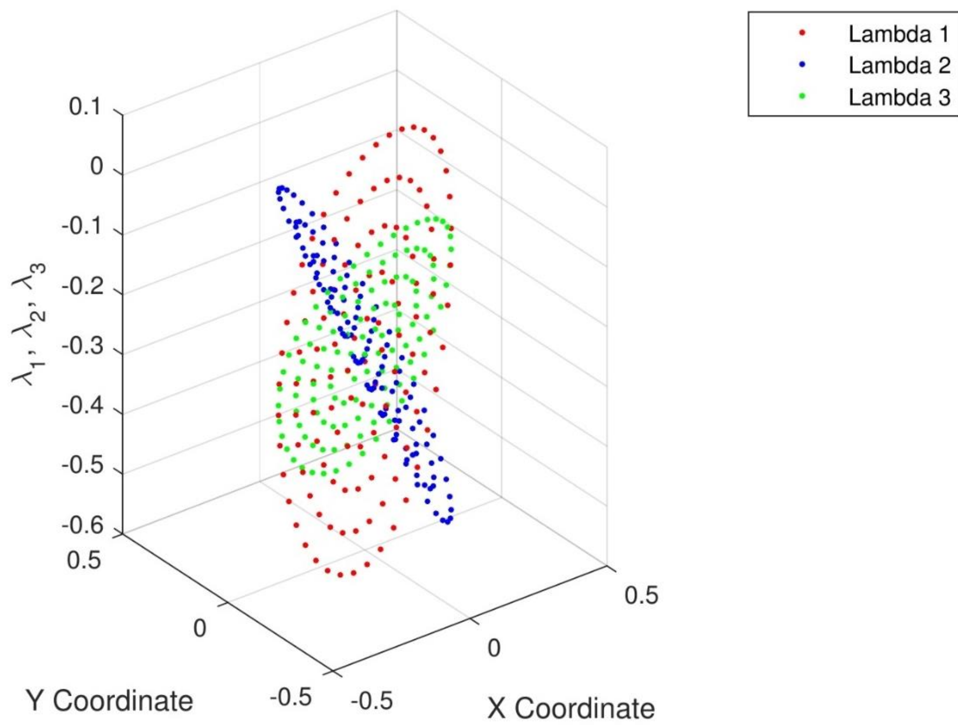


Ilustración 35. Representación de los valores de λ_1, λ_2 y λ_3 en función de (x,y)

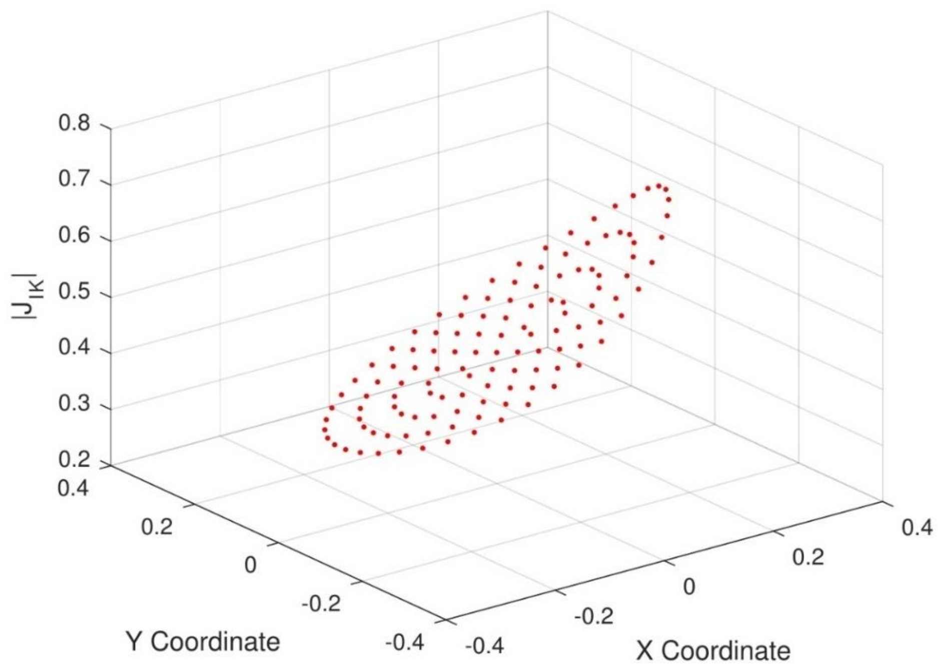


Ilustración 36. Determinante del jacobiano del problema inverso en función de (x,y)

11.4.1.2. PROBLEMA DIRECTO

La resolución del problema directo implica determinar la posición del elemento terminal conociendo las entradas del sistema, λ_1, λ_2 y λ_3. Las condiciones que debe cumplir el sistema son las mismas que en el problema inverso, por lo que el sistema (110) es igualmente válido en este caso. La única diferencia es que ahora λ_1, λ_2 y λ_3 son conocidos, y x_P, y_P y θ son incógnitas. El nuevo vector de incógnitas, por tanto, es:

$$\boldsymbol{X} = \begin{bmatrix} m_{z,0}^1 \\ n_x^1 \\ n_y^1 \\ m_{z,0}^2 \\ n_x^2 \\ n_y^2 \\ m_{z,0}^2 \\ n_x^3 \\ n_y^3 \\ x_P \\ y_P \\ \theta \end{bmatrix} \qquad (112)$$

Se lleva a cabo el cálculo en una sucesión de posiciones que están determinadas por una trayectoria en el espacio $(\lambda_1, \lambda_2, \lambda_3)$, siguiendo el mismo procedimiento que hasta ahora.

Al final, además de la posición del elemento terminal definida por x_P, y_P y θ, los datos más relevantes que se pueden obtener son la fuerza que debe ejercer cada motor sobre la barra y el determinante del jacobiano del problema directo.

Aquí, es más complicado representar gráficamente los resultados; no tiene demasiado sentido definir una trayectoria en $(\lambda_1, \lambda_2, \lambda_3)$ donde una de las tres variables sea constante, como se ha indicado en el problema inverso, y además es poco intuitivo, por lo que simplemente no se hace.

11.4.2. INTEGRALES ELÍPTICAS. PROBLEMA DIRECTO

El procedimiento a seguir es similar al que se ha explicado en el mecanismo de 2 grados de libertad. Se consideran sistemas de referencia locales en cada una de las barras, como se ve en la Ilustración 37, en los que se llevarán a cabo los cálculos de cada barra por separado. Es necesario definir sus matrices de rotación, $\boldsymbol{Rot}_1$, $\boldsymbol{Rot}_2$ y $\boldsymbol{Rot}_3$. Se conoce la inclinación de las barras en su extremo empotrado, θ_1, θ_2 y θ_3, y se trata de determinar qué forma toma el mecanismo en su conjunto con estas entradas.

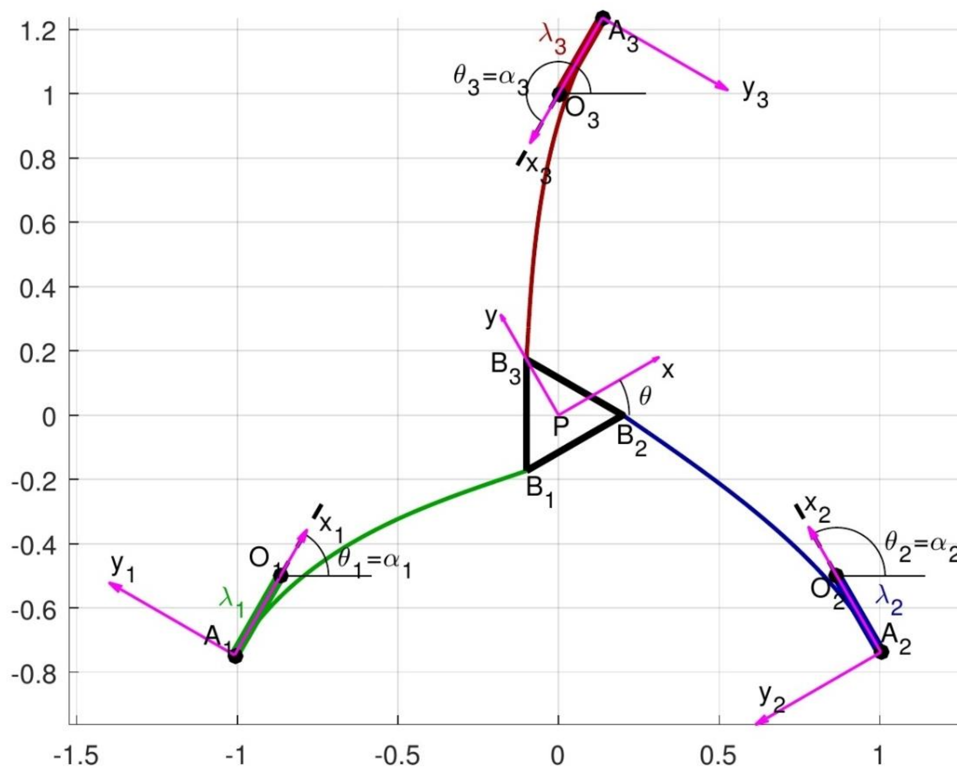


Ilustración 37. Sistemas de referencia locales en el mecanismo 3PFR

En este caso, se trata de encontrar qué valores de k_{rel} y ψ de cada una de las barras son solución del problema directo. Se discretizan las variables k_{rel}^1, ψ_1, ψ_2 y ψ_3 como se ha indicado anteriormente y se comprueba si cada posible combinación de estos 4 parámetros puede ser solución o no.

Con k_{rel}^1 y ψ_1, queda definida perfectamente la barra 1, por lo que se puede calcular el valor de R_1 y la posición del extremo en el sistema de referencia global, (x_{B1}, y_{B1}), tal y como se ha explicado en el mecanismo de 2 grados de libertad anterior.

Por otro lado, conociendo k_{rel}^1, ψ_1, ψ_2 y ψ_3, se puede despejar R_2 y R_3 de las ecuaciones del equilibrio de fuerzas en la plataforma triangular del mecanismo:

$$\boldsymbol{Rot}_1 \begin{bmatrix} R_1 \cos\psi_1 \\ R_1 \sin\psi_1 \end{bmatrix} + \boldsymbol{Rot}_2 \begin{bmatrix} R_2 \cos\psi_2 \\ R_2 \sin\psi_2 \end{bmatrix} + \boldsymbol{Rot}_3 \begin{bmatrix} R_3 \cos\psi_3 \\ R_3 \sin\psi_3 \end{bmatrix} + \begin{bmatrix} F_x^{ext} \\ F_y^{ext} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \quad (113)$$

Ahora, son conocidos R y ψ de las barras 2 y 3, por lo que se podrá obtener el valor de k_{rel} en ambas. Este cálculo no es tan sencillo, puesto que no se puede expresar k_{rel} explícitamente en función de R y ψ. Además, puede que haya varios valores de k_{rel} posibles. Lo que se hace en este caso es discretizar toda la variable k_{rel} y comprobar si cada uno de los puntos que se toman pueden corresponder con los valores de R y ψ que conocemos. Aquellos puntos que cumplan la ecuación (88) con una tolerancia aceptable, se toman como posibles valores de k_{rel}, y posteriormente se introducen en un bucle Newton-Raphson que depura esos resultados, desechando los que no son válidos y refinando los que sí. De esta manera, se obtienen todos los posibles valores de k_{rel}^2 y k_{rel}^3.

Hay que determinar si cada combinación de k_{rel}^1, ψ_1, k_{rel}^2, ψ_2, k_{rel}^3 y ψ_3 puede ser solución del problema directo. Para ello, se calcula la posición de los extremos de las barras 2 y 3, (x_{B2}, y_{B2}) y (x_{B3}, y_{B3}), como se ha hecho en ocasiones anteriores. Se definen 4 residuos, que corresponden con el balance de momentos en el elemento terminal del mecanismo y con el ensamblado de las 3 barras (los extremos de las barras deben formar un triángulo equilátero entre sí, porque esa es la forma de la plataforma, y por tanto cada uno debe distar a de los otros dos, siendo a el lado del triángulo).

$$\begin{cases} Res_M = M_1 + M_2 + M_3 + M_{ext} \\ Res_{12} = \sqrt{(x_{B2} - x_{B1})^2 + (y_{B2} - y_{B1})^2} - a \\ Res_{23} = \sqrt{(x_{B3} - x_{B2})^2 + (y_{B3} - y_{B2})^2} - a \\ Res_{31} = \sqrt{(x_{B1} - x_{B3})^2 + (y_{B1} - y_{B3})^2} - a \end{cases} \quad (114)$$

donde M_1, M_2 y M_3 son los momentos que ejercen cada barra sobre la plataforma respecto de su centro P.

$$\begin{cases} M_1 = R_{1,x}(y_{B1} - y_P) - R_{1,y}(x_{B1} - x_P) \\ M_2 = R_{2,x}(y_{B2} - y_P) - R_{2,y}(x_{B2} - x_P) \\ M_3 = R_{3,x}(y_{B3} - y_P) - R_{3,y}(x_{B3} - x_P) \end{cases} \quad (115)$$

Hay que señalar que las R_x y R_y son las proyecciones de las cargas R en el sistema de referencia global.

Se consideran que pueden ser soluciones válidas aquellas cuyos residuos son menores que una tolerancia dada. En este caso, al tratarse de magnitudes diferentes, se pueden utilizar tolerancias distintas para los residuos de longitudes y el de momentos. Se toman los datos y se depuran en un bucle de Newton-Raphson en el que se resuelve un sistema de 6 ecuaciones (3 del ensamblado geométrico del mecanismo, 2 del equilibrio de fuerzas en la plataforma y 1 del equilibrio de momentos) y 6 incógnitas (k_{rel}^1, ψ_1, k_{rel}^2, ψ_2, k_{rel}^3 y ψ_3). De esta forma, se pueden obtener soluciones más exactas y se desechan aquellas soluciones que en principio parecían válidas, pero realmente no lo son.

11.4.3. VALIDACIÓN DE LOS RESULTADOS

El procedimiento que se utiliza para comprobar el funcionamiento correcto de los algoritmos es similar al caso del mecanismo anterior. Se lleva a cabo la resolución del problema directo en posiciones en las que, por su simetría, ya se conoce, al menos, una de las soluciones. Por ejemplo, el mecanismo de la Ilustración 38, que forma un triángulo equilátero en las posiciones de sus motores.

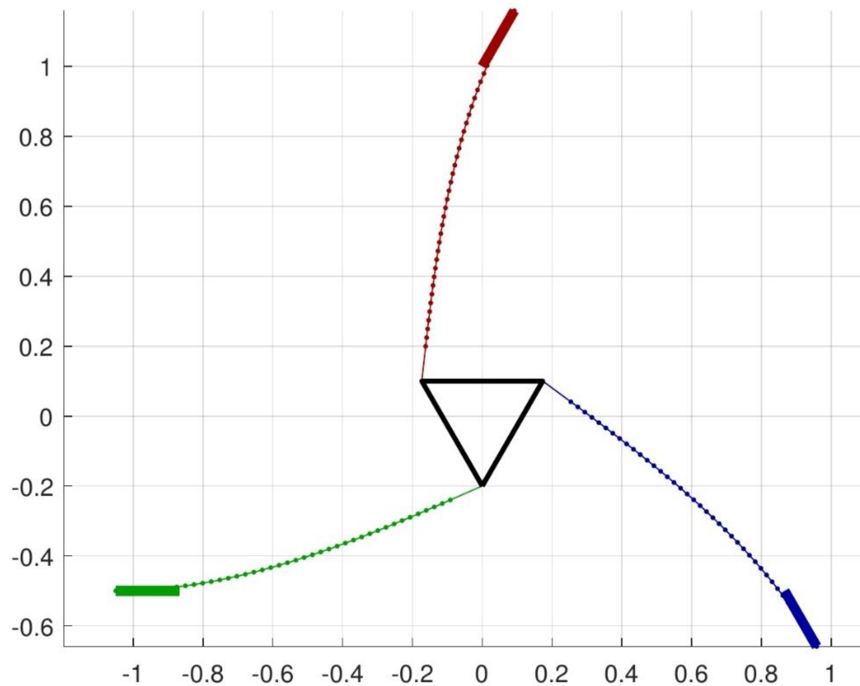


Ilustración 38. Mecanismo 3PFR simétrico

Por otro lado, se comprueba el algoritmo del problema inverso utilizando algunas soluciones obtenidas del problema directo, como se ha indicado en casos anteriores.

En el algoritmo de integrales elípticas de este método en concreto, aparece una dificultad nueva. Al tener más variables para discretizar, hay que realizar cálculos en un gran número de combinaciones, lo cual hace que el cálculo sea muy largo. Cuanto más fina sea la discretización, más preciso será el cálculo, y por tanto será necesaria una tolerancia menor, pero tardará más tiempo. Para encontrar el equilibrio adecuado entre discretización y tolerancia, de forma que el cálculo sea suficientemente preciso sin tardar demasiado tiempo, se han llevado a cabo una serie de pruebas en un mecanismo concreto. Se ha resuelto el mismo problema variando el número de puntos en que se divide la discretización de cada una de las variables ψ, n_{ψ}, y las tolerancias que se toman para los residuos de longitudes, tol_L, y de momentos, tol_M. En la Tabla 2, se puede ver el número de soluciones obtenidas en cada caso y el tiempo total de cálculo. Hay que señalar que este análisis en cuestión se lleva a cabo en un mecanismo donde las longitudes de barra son de 1 metros y la plataforma triangular tiene 0,5 metros de lado. Las tolerancias tienen unidades de metros y Newton·metro, respectivamente.

	$tol_L = 1$ $tol_M = 1$	$tol_L = 0,5$ $tol_M = 1$	$tol_L = 0,1$ $tol_M = 1$	$tol_L = 0,5$ $tol_M = 1$	$tol_L = 0,5$ $tol_M = 0,5$	$tol_L = 0,1$ $tol_M = 0,1$
$n_\psi = 15$	5 sol. t = 2030 s	5 sol. t = 1686 s	4 sol. t = 1270 s	4 sol. t = 1067 s	3 sol. t = 985 s	-
$n_\psi = 20$	6 sol. t = 4848 s	6 sol. t = 4341 s	5 sol. t = 3170 s	6 sol. t = 3534 s	5 sol. t = 3592 s	-
$n_\psi = 25$	6 sol. t = 9908 s	6 sol. t = 12176 s	5 sol. t = 7771 s	5 sol. t = 7171 s		3 sol. t = 5467 s
$n_\psi = 30$	-	-	6 sol. t = 17241 s	-	6 sol. t = 11750 s	-

Tabla 2. Comparación de grados de discretización y tolerancias

Viendo los resultados que se han obtenido en las pruebas, se decide que la mejor opción para lograr un algoritmo preciso que sea capaz de operar en un tiempo razonable es discretizar las variables ψ en 20 puntos y tomar unas tolerancias para las longitudes de 0,5 m y para los momentos, de 1 N·m.

12. DESCRIPCIÓN DE LOS RESULTADOS

Al final del proyecto, se han obtenido un conjunto de programas que aplican todos los cálculos explicados en el apartado anterior para resolver los problemas de posición de los diferentes mecanismos y representarlos gráficamente. Los códigos de todos estos programas están recogidos en el ANEXO I. Las comprobaciones finales de su correcto funcionamiento permiten asegurar que todas las soluciones que alcanzan estos programas son válidas, aunque en algunos casos puede que no sean todas las posibles.

Por un lado, en el mecanismo de barras rígidas, 3PRR, tanto en el problema directo como en el indirecto, los procedimientos de cálculo aplican directamente las ecuaciones que definen el comportamiento del mecanismo para obtener las soluciones. Por lo tanto, en este caso el programa alcanza siempre todas las soluciones posibles del problema.

Además de la mera resolución de los problemas de posición, uno de los programas comprueba si el problema inverso tiene solución en un rango de posiciones del mecanismo. Es especialmente interesante escoger un rango amplio en las variables de posición para que el estudio del espacio de trabajo sea lo más exhaustivo posible. De esta manera, el programa permite obtener todos los límites del espacio de trabajo y representarlo gráficamente, como se ha mostrado anteriormente en la Ilustración 21, lo cual ofrece una visión global del comportamiento del mecanismo.

En el análisis de los mecanismos flexibles, es necesario diferenciar las dos formas que se han planteado para resolver sus problemas de posición: por integración numérica y mediante la aplicación de integrales elípticas. Por eso, se han realizado programas diferentes que resuelven los problemas de manera distinta.

En el primer caso, el propio procedimiento de cálculo es iterativo y parte de una aproximación inicial, de la cual depende la solución a la que llega. Por eso, se realizan los cálculos en una sucesión de posiciones cercanas entre sí, para que los datos de una puedan servir como aproximación a la siguiente. Se trata de una manera muy rápida y eficiente de poder resolver los problemas directo e inverso en un conjunto de posiciones que sean de interés. Si el rango de posiciones es amplio, se puede llevar a cabo un buen análisis del espacio de trabajo del mecanismo. Sobre todo, es importante el cálculo del determinante del jacobiano de los problemas directo e inverso, puesto que permite ver qué posiciones puede alcanzar el mecanismo y dónde están las posiciones de singularidad. En el mecanismo de 2 grados de libertad, es fácil representar la evolución de estos datos en cada punto del plano, como se ha visto en la Ilustración 28. En el mecanismo de 3 grados de libertad, se puede hacer el mismo tipo de representación tomando una orientación constante de la plataforma.

El método de resolución de las integrales elípticas tan sólo se ha aplicado al cálculo del problema directo de los mecanismos. Centra su estudio en una única posición y ofrece varias soluciones posibles al problema. En los programas que se han realizado, se lleva a cabo una discretización de las variables del problema y se determina cuáles podrían ser solución según una tolerancia y se depuran. Por eso, es posible obtener varias soluciones. Ha sido necesario encontrar un equilibrio entre la discretización y la tolerancia utilizadas. Si se hace una discretización muy grosera, es necesaria una tolerancia grande para no pasar por alto posibles soluciones del problema, y aun así es probable que esto ocurra. Si la discretización es muy fina, en cambio, la tolerancia necesaria será mucho más pequeña y los resultados más exactos, pero el tiempo de

cálculo del programa aumenta considerablemente. En este sentido, el mecanismo de 3 grados de libertad ha supuesto un reto importante, puesto que tiene más variables para discretizar y, por lo tanto, muchas más combinaciones que se analizan. Ha sido necesario un análisis de distintas alternativas en cuanto a la elección de la discretización y las tolerancias para que el funcionamiento sea el más adecuado.

En definitiva, los dos métodos de resolución aportan dos visiones diferentes que permiten analizar y conocer mejor el comportamiento de los mecanismos planos ultraflexibles, lo cual es de gran interés por las grandes ventajas que tienen.

13. EJEMPLOS DE APLICACIÓN

En este apartado, se presentan varios ejemplos de funcionamiento de los programas realizados, que ofrecen una idea global de sus posibilidades.

13.1. 3PRR

13.1.1. PROBLEMA INVERSO

DEFINICIÓN DEL MECANISMO	ENTRADA	SOLUCIONES (en metros)		
L = 1 m	x = 0.6 m	λ_1 = 1.5165	λ_2 = 0.48348	λ_3 = -0.18254
a = 1 m	y = 0.4 m	λ_1 = 1.5165	λ_2 = 0.48348	λ_3 = 1.6505
B2 = 3 m	ψ = 0°	λ_1 = 1.5165	λ_2 = 2.3165	λ_3 = -0.18254
B3 = 1.5 m		λ_1 = 1.5165	λ_2 = 2.3165	λ_3 = 1.6505
H3 = 2 m		λ_1 = -0.31652	λ_2 = 0.48348	λ_3 = -0.18254
		λ_1 = -0.31652	λ_2 = 0.48348	λ_3 = 1.6505
		λ_1 = -0.31652	λ_2 = 2.3165	λ_3 = -0.18254
		λ_1 = -0.31652	λ_2 = 2.3165	λ_3 = 1.6505

Tabla 3. Ejemplo 1 de aplicación del problema inverso del mecanismo 3PRR

DEFINICIÓN DEL MECANISMO	ENTRADA	SOLUCIONES (en metros)		
L = 1 m	x = 3 m	λ_1 = 3.9539	λ_2 = -1.071	λ_3 = 1.4132
a = 0.3 m	y = 0.3 m	λ_1 = 3.9539	λ_2 = -1.071	λ_3 = 3.4072
B2 = 3 m	ψ = 45°	λ_1 = 3.9539	λ_2 = 0.64677	λ_3 = 1.4132
B3 = 3 m		λ_1 = 3.9539	λ_2 = 0.64677	λ_3 = 3.4072
H3 = 3 m		λ_1 = 2.0461	λ_2 = -1.071	λ_3 = 1.4132
		λ_1 = 2.0461	λ_2 = -1.071	λ_3 = 3.4072
		λ_1 = 2.0461	λ_2 = 0.64677	λ_3 = 1.4132
		λ_1 = 2.0461	λ_2 = 0.64677	λ_3 = 3.4072

Tabla 4. Ejemplo 2 de aplicación del problema inverso del mecanismo 3PRR

13.1.2. PROBLEMA DIRECTO

DEFINICIÓN DEL MECANISMO	ENTRADA	SOLUCIONES		
L = 1 m	λ_1 = 2 m	x = 2.2171 m	y = 0.97615 m	ψ = 345.6206°
a = 1 m	λ_2 = 0.5 m	x = 1.5162 m	y = 0.87519 m	ψ = 7.1679°
B2 = 3 m	λ_3 = 0.2 m			
B3 = 2.5 m				
H3 = 2.8 m				

Tabla 5. Ejemplo 1 de aplicación del problema directo del mecanismo 3PRR

DEFINICIÓN DEL MECANISMO	ENTRADA	SOLUCIONES		
L = 1 m	λ_1 = 0.4 m	x = 1.0535 m	y = 0.7569 m	ψ = 341.5164°
a = 1 m	λ_2 = 0.1 m	x = 1.2312 m	y = 0.55593 m	ψ = 9.9967°
B2 = 3 m	λ_3 = 0.6 m			
B3 = 2 m				
H3 = 3 m				

Tabla 6. Ejemplo 2 de aplicación del problema directo del mecanismo 3PRR

13.2. 2RFR

13.2.1. INTEGRALES ELÍPTICAS. PROBLEMA DIRECTO

Se plantea un mecanismo donde los motores 1 y 2 están situados en (-0.3, 0) y (0, 0.3), con una posición angular de 40° y 60°, respectivamente. La longitud de las barras es de 1 metro. Sobre el elemento terminal actúa una fuerza externa que tiene una proyección horizontal de 0.3 N y vertical de -0.1 N. Las soluciones del problema son las siguientes:

	SOLUCIÓN 1	SOLUCIÓN 2
ψ_1 [rad]	5.304574	1.962541
$k_{rel,1}$	0.969830	-0.004391
ψ_2 [rad]	1.825456	4.811041
$k_{rel,2}$	-0.992135	0.943413

Tabla 7. Soluciones del ejemplo del problema directo del 2RFR

Las formas que puede adoptar el mecanismo son las siguientes:

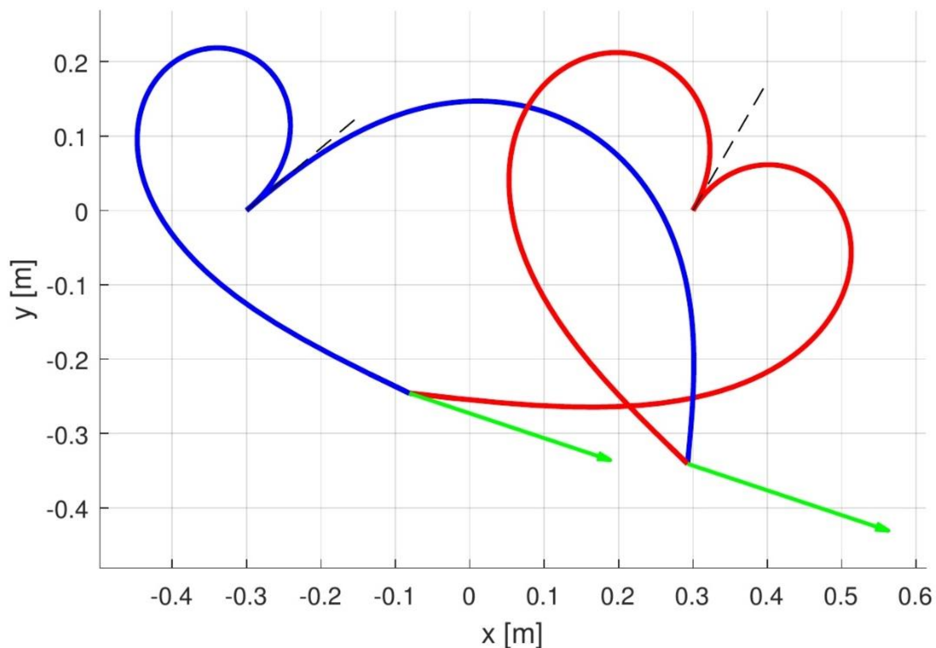


Ilustración 39. Soluciones del ejemplo del problema directo del 2RFR

13.3. 3PFR

13.3.1. INTEGRALES ELÍPTICAS. PROBLEMA DIRECTO

Se plantea un mecanismo donde las guías de los motores pasan por los puntos O_1(-0.866, -0.5), O_2(0.866, -0.5) y O_3(0, 1), y las barras tienen un ángulo de inclinación en su extremo empotrado de θ_1= 45°, θ_2= 0°, θ_3= -60°. Las longitudes de las barras son de 1 metro. Definiendo las tres variables de entrada λ iguales a 0 y una fuerza externa vertical y hacia arriba de 0.5 N aplicada en la plataforma triangular, las soluciones del problema directo son:

	SOLUCIÓN 1	SOLUCIÓN 2
ψ_1 [rad]	5.1834	5.9923
$k_{rel,1}$	-0.7267	-0.9062
ψ_2 [rad]	5.0059	5.3280
$k_{rel,2}$	-0.9286	-0.8652
ψ_3 [rad]	2.6908	2.8573
$k_{rel,3}$	-0.0952	-0.0629

Tabla 8. Soluciones del ejemplo del problema directo del 3PFR

El mecanismo podrá adoptar las siguientes configuraciones:

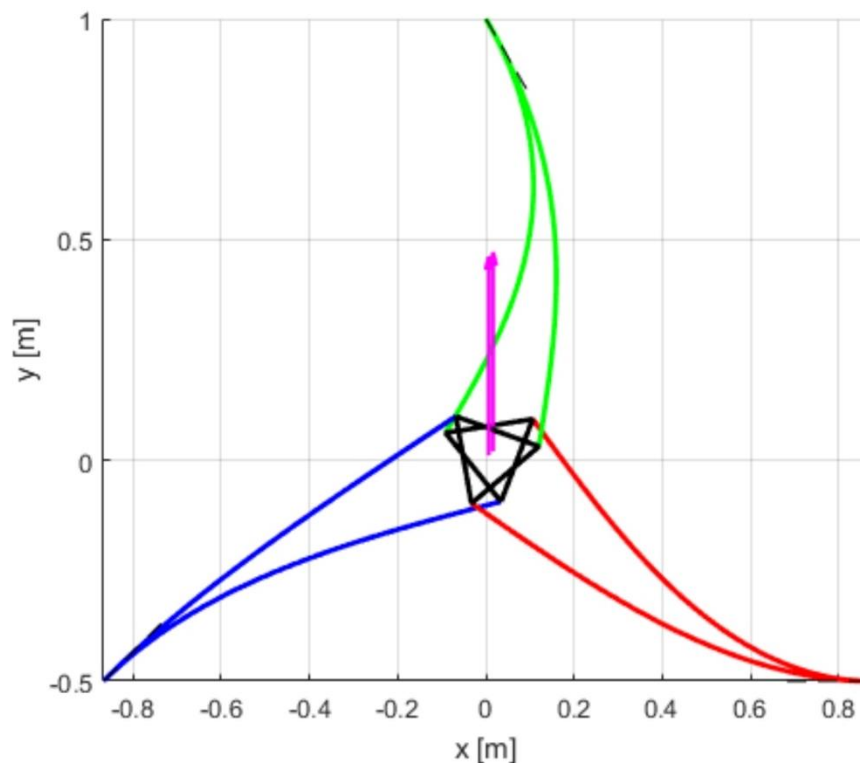


Ilustración 40. Soluciones del ejemplo del problema directo del 3PFR

14. ASPECTOS ECONÓMICOS. PRESUPUESTO

La valoración económica de este proyecto es sencilla por la cantidad reducida de recursos técnicos que se han empleado en él.

Por un lado, dos licencias de uso individual del software Matlab de duración anual, que tienen un coste de 800 € cada una. La utilización de una de estas licencias, la del alumno, ha sido exclusiva para la realización de este proyecto y, por tanto, se debe asumir su coste completo. En cambio, la licencia del profesor se emplea para otras tareas también, por lo que, estimando un uso anual de 400 horas, se calcula una amortización de 2 €/h.

Por otro lado, dos ordenadores cuyo precio de compra fue de 600 €. En este caso, no sólo se han empleado para este proyecto; se han utilizado con anterioridad y se utilizarán en el futuro para otras tareas. Por lo tanto, hay que considerar tan sólo el coste de uso correspondiente. Se les estima una vida útil de 6 años, contemplando un tiempo de uso anual de 400 horas, lo que da como resultado una amortización de 0,25 €/hora.

Hay que tener en cuenta también el coste de las horas de trabajo dedicadas al proyecto, tanto de ingeniería senior (profesor) como de ingeniería junior (alumno). En total, la dedicación ha sido de 250 horas de ingeniería junior y 50 horas de ingeniería senior, con un coste de 25 y 40 €/h, respectivamente.

En la siguiente tabla se pueden ver los costes totales desgranados:

CONCEPTO	COSTE TOTAL
Matlab	900,00 €
Ordenadores	150,00 €
Ingeniería senior	1.250,00 €
Ingeniería junior	10.000,00 €
TOTAL	**12.300,00 €**

Tabla 9. Desgrane de costes

Y para poder valorarlo de una forma más visual, en la Ilustración 41 aparece una representación gráfica de dichos costes.

Ilustración 41. Representación de de la proporción de costes totales

15. CONCLUSIONES

Los mecanismos ultraflexibles presentan grandes ventajas respecto a los de barras rígidas. Por eso, resulta interesante analizar y conocer mejor su comportamiento desde el punto de vista teórico. En ese sentido, los programas que se han realizado en este proyecto permiten obtener una visión general de estos mecanismos, puesto que son capaces de plantear diferentes situaciones de trabajo para cada uno de ellos y resolver sus problemas de posición.

Además, la aplicación de dos procedimientos de cálculo diferentes ofrece dos perspectivas distintas de los comportamientos que se están estudiando. Por un lado, un análisis del espacio de trabajo de los mecanismos a través de una trayectoria definida, que favorece una visión de conjunto; y por otro, un análisis más concreto de una posición determinada, que da como resultado varias soluciones posibles de los problemas cinemáticos.

En los procesos de cálculo, a menudo se ha utilizado una discretización de las variables del problema y un método iterativo posterior para su resolución. Al repetir los cálculos para muchos valores distintos de las variables, en ocasiones los programas tardan más tiempo del deseable en llegar a la solución. Ha sido necesario encontrar un equilibrio adecuado entre el grado de discretización y la tolerancia posterior para que los programas sean lo suficientemente precisos sin que su resolución dure demasiado.

16. BIBLIOGRAFÍA

[1] Oscar Altuzarra, Diego Caballero, Francisco Javier Campa, Charles Pinto, «*Position Analysis in Planar Parallel Continuum Mechanisms*», Departamento de Ingeniería Mecánica, Universidad del País Vasco UPV/EHU, Bilbao.

[2] Oscar Altuzarra, Diego Caballero, Francisco Javier Campa, «*Nonlinear Flexure Analysis of Mechanisms*», Departamento de Ingeniería Mecánica, Universidad del País Vasco UPV/EHU, Bilbao.

[3] Brian Hahn, Daniel T. Valentine, «*Essential MATLAB for engineers and scientists*», Elsevier Butterworth Heinemann, Amsterdam, 2007.

[4] Documentación de ayuda de Matlab, https://es.mathworks.com/help/matlab/

ANEXO I

1. DESARROLLO MATEMÁTICO DEL PROBLEMA DIRECTO DEL MECANISMO 3PRR

Partiendo de los siguientes sistemas de ecuaciones:

$$\begin{cases} x_P = \lambda_1 + L\cos\theta_1 \\ y_P = L\sin\theta_1 \end{cases} \qquad (116)$$

$$\begin{cases} x_P = B_2 - \lambda_2 + L\cos\theta_2 - a\cos\psi \\ y_P = L\sin\theta_2 - a\sin\psi \end{cases} \qquad (117)$$

$$\begin{cases} x_P = B_3 - L\sin\theta_3 - a\cos(\frac{\pi}{3} + \psi) \\ y_P = H_3 - \lambda_3 + L\cos\theta_3 - a\sin(\frac{\pi}{3} + \psi) \end{cases} \qquad (118)$$

En primer lugar, se pretende eliminar las variables θ_1, θ_2 y θ_3 de los sistemas de ecuaciones. Para ello, hay que combinar entre sí las ecuaciones de los sistemas (116), (117) y (118), teniendo en cuenta que $\sin^2\theta + \cos^2\theta = 1$. El resultado de unir las dos ecuaciones del sistema (116) es el siguiente:

$$(x_P - \lambda_1)^2 + y_P^2 = L^2 \qquad (119)$$

Para el sistema (117):

$$\begin{gathered} x_P^2 + x_P\big(2a\cos\psi + 2(\lambda_2 - B_2)\big) + y_P^2 + 2y_P\, a\sin\psi + \\ +2a(\lambda_2 - B_2)\cos\psi + a^2 + (\lambda_2 - B_2)^2 = L^2 \end{gathered} \qquad (120)$$

Y para el sistema (118):

$$\begin{gathered} x_P^2 + x_P\left(-2B_3 + 2a\cos\left(\frac{\pi}{3} + \psi\right)\right) + y_P^2 + \\ +y_P\left(2a\sin\left(\frac{\pi}{3} + \psi\right) + 2(\lambda_3 - H_3)\right) + 2a(\lambda_3 - H_3)\sin\left(\frac{\pi}{3} + \psi\right) - \\ -2aB_3\cos\left(\frac{\pi}{3} + \psi\right) + B_3^2 + a^2 + (\lambda_3 - H_3)^2 = L^2 \end{gathered} \qquad (121)$$

Combinando las ecuaciones (126) y (127) y agrupando en los elementos se obtiene:

$$\begin{gathered} x_P\,(2\lambda_1 + 2(\lambda_2 - B_2) + 2a\cos\psi) + y_P\,(2a\sin\psi) + \\ +2a(\lambda_2 - B_2)\cos\psi - \lambda_1^2 + a^2 + (\lambda_2 - B_2)^2 = 0 \end{gathered} \qquad (122)$$

Y combinando las ecuaciones (126) y (128):

$$x_P\big(2\lambda_1 - 2B_3 + a\cos\psi - a\sqrt{3}\sin\psi\big) + \\ +y_P\big(a\sqrt{3}\cos\psi + a\sin\psi + 2(\lambda_3 - H_3)\big) + \\ +\big(a\sqrt{3}(\lambda_3 - H_3) - aB_3\big)\cos\psi + \big(a(\lambda_3 - H_3) + a\sqrt{3}B_3\big)\sin\psi - \\ -\lambda_1^2 + B_3^2 + a^2 + (\lambda_3 - H_3)^2 = 0 \tag{123}$$

Se simplifica la expresión de estas ecuaciones de la siguiente forma:

$$x_P f_1(\psi) + y_P f_2(\psi) + f_3(\psi) = 0 \tag{124}$$

$$x_P g_1(\psi) + y_P g_2(\psi) + g_3(\psi) = 0 \tag{125}$$

Donde las funciones f y g son:

$$f_1(\psi) = K_1\cos\psi + K_2 \tag{126}$$

$$f_2(\psi) = K_3\sin\psi \tag{127}$$

$$f_3(\psi) = K_4\cos\psi + K_5 \tag{128}$$

$$g_1(\psi) = Q_1\sin\psi + Q_2\cos\psi + Q_3 \tag{129}$$

$$g_2(\psi) = Q_4\sin\psi + Q_5\cos\psi + Q_6 \tag{130}$$

$$g_3(\psi) = Q_7\sin\psi + Q_8\cos\psi + Q_9 \tag{131}$$

El valor de las constantes K y Q es el siguiente:

$$K_1 = 2a \tag{132}$$

$$K_2 = 2\lambda_1 + 2(\lambda_2 - B_2) \tag{133}$$

$$K_3 = 2a \tag{134}$$

$$K_4 = 2a(\lambda_2 - B_2) \tag{135}$$

$$K_5 = -\lambda_1^2 + a^2 + (\lambda_2 - B_2)^2 \tag{136}$$

$$Q_1 = -a\sqrt{3} \tag{137}$$

$$Q_2 = a \tag{138}$$

$$Q_3 = 2(\lambda_1 - B_3) \tag{139}$$

$$Q_4 = a \tag{140}$$

$$Q_5 = a\sqrt{3} \tag{141}$$

$$Q_6 = 2(\lambda_3 - H_3) \tag{142}$$

$$Q_7 = a(\lambda_3 - H_3) + a\sqrt{3}B_3 \tag{143}$$

$$Q_8 = a\sqrt{3}(\lambda_3 - H_3) - a\,B_3 \tag{144}$$

$$Q_9 = -\lambda_1^2 + B_3^2 + a^2 + (\lambda_3 - H_3)^2 \tag{145}$$

Despejando x_P y y_P de las ecuaciones (124) y (125),

$$x_P = \frac{f_3(\psi)\;g_2(\psi) - f_2(\psi)\;g_3(\psi)}{f_2(\psi)\;g_1(\psi) - f_1(\psi)\;g_2(\psi)} \tag{146}$$

$$y_P = \frac{f_1(\psi)\;g_3(\psi) - f_3(\psi)\;g_1(\psi)}{f_2(\psi)\;g_1(\psi) - f_1(\psi)\;g_2(\psi)} \tag{147}$$

Al introducir de nuevo las ecuaciones (146) y (147) en la ecuación (119) se obtiene:

$$\begin{gathered}\big(f_3g_2 - f_2g_3 - \lambda_1(f_2g_1 - f_1g_2)\big)^2 + (f_1g_3 - f_3g_1)^2 - \\ -L^2(f_2g_1 - f_1g_2)^2 = 0\end{gathered} \tag{148}$$

Se trata de un polinomio en ψ igualado a 0. Sin embargo, es difícil despejar ψ de una ecuación donde la incógnita está dentro de senos y cosenos. Para resolver este problema, se realiza un cambio de variable:

$$t = \tan\frac{\psi}{2} \tag{149}$$

$$\sin\psi = \frac{2t}{1+t^2} \tag{150}$$

$$\cos\psi = \frac{1-t^2}{1+t^2} \tag{151}$$

El resultado es una ecuación que es un polinomio en t igualado a 0.

$$C_0 + C_1\, t + C_2\, t^2 + C_3\, t^3 + C_4\, t^4 + C_5\, t^5 + C_6\, t^6 + C_7\, t^7 + C_8\, t^8 = 0 \qquad (152)$$

Donde los coeficientes son:

C8 = - K1^2*L^2*Q5^2 + 2*K1^2*L^2*Q5*Q6 - K1^2*L^2*Q6^2 + K1^2*Q5^2*λ1^2 - 2*K1^2*Q5*Q6*λ1^2 + K1^2*Q6^2*λ1^2 + K1^2*Q8^2 - 2*K1^2*Q8*Q9 + K1^2*Q9^2 + 2*K1*K2*L^2*Q5^2 - 4*K1*K2*L^2*Q5*Q6 + 2*K1*K2*L^2*Q6^2 - 2*K1*K2*Q5^2*λ1^2 + 4*K1*K2*Q5*Q6*λ1^2 - 2*K1*K2*Q6^2*λ1^2 - 2*K1*K2*Q8^2 + 4*K1*K2*Q8*Q9 - 2*K1*K2*Q9^2 - 2*K1*K4*Q2*Q8 + 2*K1*K4*Q2*Q9 + 2*K1*K4*Q3*Q8 - 2*K1*K4*Q3*Q9 + 2*K1*K4*Q5^2*λ1 - 4*K1*K4*Q5*Q6*λ1 + 2*K1*K4*Q6^2*λ1 + 2*K1*K5*Q2*Q8 - 2*K1*K5*Q2*Q9 - 2*K1*K5*Q3*Q8 + 2*K1*K5*Q3*Q9 - 2*K1*K5*Q5^2*λ1 + 4*K1*K5*Q5*Q6*λ1 - 2*K1*K5*Q6^2*λ1 - K2^2*L^2*Q5^2 + 2*K2^2*L^2*Q5*Q6 - K2^2*L^2*Q6^2 + K2^2*Q5^2*λ1^2 - 2*K2^2*Q5*Q6*λ1^2 + K2^2*Q6^2*λ1^2 + K2^2*Q8^2 - 2*K2^2*Q8*Q9 + K2^2*Q9^2 + 2*K2*K4*Q2*Q8 - 2*K2*K4*Q2*Q9 - 2*K2*K4*Q3*Q8 + 2*K2*K4*Q3*Q9 - 2*K2*K4*Q5^2*λ1 + 4*K2*K4*Q5*Q6*λ1 - 2*K2*K4*Q6^2*λ1 - 2*K2*K5*Q2*Q8 + 2*K2*K5*Q2*Q9 + 2*K2*K5*Q3*Q8 - 2*K2*K5*Q3*Q9 + 2*K2*K5*Q5^2*λ1 - 4*K2*K5*Q5*Q6*λ1 + 2*K2*K5*Q6^2*λ1 + K4^2*Q2^2 - 2*K4^2*Q2*Q3 + K4^2*Q3^2 + K4^2*Q5^2 - 2*K4^2*Q5*Q6 + K4^2*Q6^2 - 2*K4*K5*Q2^2 + 4*K4*K5*Q2*Q3 - 2*K4*K5*Q3^2 - 2*K4*K5*Q5^2 + 4*K4*K5*Q5*Q6 - 2*K4*K5*Q6^2 + K5^2*Q2^2 - 2*K5^2*Q2*Q3 + K5^2*Q3^2 + K5^2*Q5^2 - 2*K5^2*Q5*Q6 + K5^2*Q6^2

C7 = 4*K4^2*Q1*Q3 - 4*K4^2*Q1*Q2 - 4*K5^2*Q1*Q2 + 4*K5^2*Q1*Q3 - 4*K4^2*Q4*Q5 + 4*K4^2*Q4*Q6 - 4*K5^2*Q4*Q5 + 4*K5^2*Q4*Q6 - 4*K1^2*Q7*Q8 + 4*K1^2*Q7*Q9 - 4*K2^2*Q7*Q8 + 4*K2^2*Q7*Q9 + 4*K1^2*L^2*Q4*Q5 - 4*K1^2*L^2*Q4*Q6 + 4*K2^2*L^2*Q4*Q5 - 4*K2^2*L^2*Q4*Q6 - 4*K1^2*Q4*Q5*λ1^2 + 4*K1^2*Q4*Q6*λ1^2 - 4*K2^2*Q4*Q5*λ1^2 + 4*K2^2*Q4*Q6*λ1^2 + 8*K4*K5*Q1*Q2 - 8*K4*K5*Q1*Q3 + 4*K1*K4*Q1*Q8 + 4*K1*K4*Q2*Q7 - 4*K1*K4*Q1*Q9 - 4*K1*K4*Q3*Q7 - 4*K1*K5*Q1*Q8 - 4*K1*K5*Q2*Q7 - 4*K2*K4*Q1*Q8 - 4*K2*K4*Q2*Q7 + 4*K1*K5*Q1*Q9 + 4*K1*K5*Q3*Q7 + 4*K2*K4*Q1*Q9 + 4*K2*K4*Q3*Q7 + 4*K2*K5*Q1*Q8 + 4*K2*K5*Q2*Q7 - 4*K2*K5*Q1*Q9 - 4*K2*K5*Q3*Q7 + 8*K1*K2*Q7*Q8 + 8*K4*K5*Q4*Q5 - 8*K1*K2*Q7*Q9 - 8*K4*K5*Q4*Q6 + 4*K3*K4*Q5*Q8 - 4*K3*K4*Q5*Q9 - 4*K3*K4*Q6*Q8 - 4*K3*K5*Q5*Q8 + 4*K3*K4*Q6*Q9 + 4*K3*K5*Q5*Q9 + 4*K3*K5*Q6*Q8 - 4*K3*K5*Q6*Q9 - 8*K1*K4*Q4*Q5*λ1 + 4*K3*K4*Q2*Q5*λ1 + 8*K1*K4*Q4*Q6*λ1 + 8*K1*K5*Q4*Q5*λ1 + 8*K2*K4*Q4*Q5*λ1 - 4*K3*K4*Q2*Q6*λ1 - 4*K3*K4*Q3*Q5*λ1 - 4*K3*K5*Q2*Q5*λ1 - 8*K1*K5*Q4*Q6*λ1 - 8*K2*K4*Q4*Q6*λ1 - 8*K2*K5*Q4*Q5*λ1 + 4*K3*K4*Q3*Q6*λ1 + 4*K3*K5*Q2*Q6*λ1 + 4*K3*K5*Q3*Q5*λ1 + 4*K1*K3*Q5*Q8*λ1 + 8*K2*K5*Q4*Q6*λ1 - 4*K3*K5*Q3*Q6*λ1 - 4*K1*K3*Q5*Q9*λ1 - 4*K1*K3*Q6*Q8*λ1 - 4*K2*K3*Q5*Q8*λ1 + 4*K1*K3*Q6*Q9*λ1 + 4*K2*K3*Q5*Q9*λ1 + 4*K2*K3*Q6*Q8*λ1 - 4*K2*K3*Q6*Q9*λ1 - 4*K1*K3*L^2*Q2*Q5 - 8*K1*K2*L^2*Q4*Q5 + 4*K1*K3*L^2*Q2*Q6 + 4*K1*K3*L^2*Q3*Q5 + 4*K2*K3*L^2*Q2*Q5 + 8*K1*K2*L^2*Q4*Q6 - 4*K1*K3*L^2*Q3*Q6 - 4*K2*K3*L^2*Q2*Q6 - 4*K2*K3*L^2*Q3*Q5 + 4*K2*K3*L^2*Q3*Q6 + 4*K1*K3*Q2*Q5*λ1^2 + 8*K1*K2*Q4*Q5*λ1^2 - 4*K1*K3*Q2*Q6*λ1^2 - 4*K1*K3*Q3*Q5*λ1^2 - 4*K2*K3*Q2*Q5*λ1^2 - 8*K1*K2*Q4*Q6*λ1^2 + 4*K1*K3*Q3*Q6*λ1^2 + 4*K2*K3*Q2*Q6*λ1^2 + 4*K2*K3*Q3*Q5*λ1^2 - 4*K2*K3*Q3*Q6*λ1^2

C6 = - 4*K1^2*L^2*Q4^2 + 4*K1^2*L^2*Q5^2 - 4*K1^2*L^2*Q5*Q6 + 4*K1^2*Q4^2*λ1^2 - 4*K1^2*Q5^2*λ1^2 + 4*K1^2*Q5*Q6*λ1^2 + 4*K1^2*Q7^2 - 4*K1^2*Q8^2 + 4*K1^2*Q8*Q9 + 8*K1*K2*L^2*Q4^2 - 4*K1*K2*L^2*Q5^2 + 4*K1*K2*L^2*Q6^2 - 8*K1*K2*Q4^2*λ1^2 + 4*K1*K2*Q5^2*λ1^2 - 4*K1*K2*Q6^2*λ1^2 - 8*K1*K2*Q7^2 + 4*K1*K2*Q8^2 - 4*K1*K2*Q9^2 + 8*K1*K3*L^2*Q1*Q5 - 8*K1*K3*L^2*Q1*Q6 + 8*K1*K3*L^2*Q2*Q4 - 8*K1*K3*L^2*Q3*Q4 - 8*K1*K3*Q1*Q5*λ1^2 + 8*K1*K3*Q1*Q6*λ1^2 - 8*K1*K3*Q2*Q4*λ1^2 + 8*K1*K3*Q3*Q4*λ1^2 - 8*K1*K3*Q4*Q8*λ1 + 8*K1*K3*Q4*Q9*λ1 - 8*K1*K3*Q5*Q7*λ1 + 8*K1*K3*Q6*Q7*λ1 - 8*K1*K4*Q1*Q7 + 8*K1*K4*Q2*Q8 - 4*K1*K4*Q2*Q9 - 4*K1*K4*Q3*Q8 + 8*K1*K4*Q4^2*λ1 - 8*K1*K4*Q5^2*λ1 + 8*K1*K4*Q5*Q6*λ1 + 8*K1*K5*Q1*Q7 - 4*K1*K5*Q2*Q8 + 4*K1*K5*Q3*Q9 - 8*K1*K5*Q4^2*λ1 + 4*K1*K5*Q5^2*λ1 - 4*K1*K5*Q6^2*λ1 - 4*K2^2*L^2*Q4^2 + 4*K2^2*L^2*Q5*Q6 - 4*K2^2*L^2*Q6^2 + 4*K2^2*Q4^2*λ1^2 - 4*K2^2*Q5*Q6*λ1^2 + 4*K2^2*Q6^2*λ1^2 +

4*K2^2*Q7^2 - 4*K2^2*Q8*Q9 + 4*K2^2*Q9^2 - 8*K2*K3*L^2*Q1*Q5 + 8*K2*K3*L^2*Q1*Q6 - 8*K2*K3*L^2*Q2*Q4 + 8*K2*K3*L^2*Q3*Q4 + 8*K2*K3*Q1*Q5*λ1^2 - 8*K2*K3*Q1*Q6*λ1^2 + 8*K2*K3*Q2*Q4*λ1^2 - 8*K2*K3*Q3*Q4*λ1^2 + 8*K2*K3*Q4*Q8*λ1 - 8*K2*K3*Q4*Q9*λ1 + 8*K2*K3*Q5*Q7*λ1 - 8*K2*K3*Q6*Q7*λ1 + 8*K2*K4*Q1*Q7 - 4*K2*K4*Q2*Q8 + 4*K2*K4*Q3*Q9 - 8*K2*K4*Q4^2*λ1 + 4*K2*K4*Q5^2*λ1 - 4*K2*K4*Q6^2*λ1 - 8*K2*K5*Q1*Q7 + 4*K2*K5*Q2*Q9 + 4*K2*K5*Q3*Q8 - 8*K2*K5*Q3*Q9 + 8*K2*K5*Q4^2*λ1 - 8*K2*K5*Q5*Q6*λ1 + 8*K2*K5*Q6^2*λ1 - 4*K3^2*L^2*Q2^2 + 8*K3^2*L^2*Q2*Q3 - 4*K3^2*L^2*Q3^2 + 4*K3^2*Q2^2*λ1^2 - 8*K3^2*Q2*Q3*λ1^2 + 8*K3^2*Q2*Q8*λ1 - 8*K3^2*Q2*Q9*λ1 + 4*K3^2*Q3^2*λ1^2 - 8*K3^2*Q3*Q8*λ1 + 8*K3^2*Q3*Q9*λ1 + 4*K3^2*Q8^2 - 8*K3^2*Q8*Q9 + 4*K3^2*Q9^2 - 8*K3*K4*Q1*Q5*λ1 + 8*K3*K4*Q1*Q6*λ1 - 8*K3*K4*Q2*Q4*λ1 + 8*K3*K4*Q3*Q4*λ1 - 8*K3*K4*Q4*Q8 + 8*K3*K4*Q4*Q9 - 8*K3*K4*Q5*Q7 + 8*K3*K4*Q6*Q7 + 8*K3*K5*Q1*Q5*λ1 - 8*K3*K5*Q1*Q6*λ1 + 8*K3*K5*Q2*Q4*λ1 - 8*K3*K5*Q3*Q4*λ1 + 8*K3*K5*Q4*Q8 - 8*K3*K5*Q4*Q9 + 8*K3*K5*Q5*Q7 - 8*K3*K5*Q6*Q7 + 4*K4^2*Q1^2 - 4*K4^2*Q2^2 + 4*K4^2*Q2*Q3 + 4*K4^2*Q4^2 - 4*K4^2*Q5^2 + 4*K4^2*Q5*Q6 - 8*K4*K5*Q1^2 + 4*K4*K5*Q2^2 - 4*K4*K5*Q3^2 - 8*K4*K5*Q4^2 + 4*K4*K5*Q5^2 - 4*K4*K5*Q6^2 + 4*K5^2*Q1^2 - 4*K5^2*Q2*Q3 + 4*K5^2*Q3^2 + 4*K5^2*Q4^2 - 4*K5^2*Q5*Q6 + 4*K5^2*Q6^2

C5 = 12*K4^2*Q1*Q2 - 4*K4^2*Q1*Q3 - 4*K5^2*Q1*Q2 + 12*K5^2*Q1*Q3 + 12*K4^2*Q4*Q5 - 4*K4^2*Q4*Q6 - 4*K5^2*Q4*Q5 + 12*K5^2*Q4*Q6 + 12*K1^2*Q7*Q8 - 4*K1^2*Q7*Q9 - 4*K2^2*Q7*Q8 + 12*K2^2*Q7*Q9 - 16*K3^2*Q7*Q8 + 16*K3^2*Q7*Q9 - 16*K3^2*Q1*Q8*λ1 - 16*K3^2*Q2*Q7*λ1 + 16*K3^2*Q1*Q9*λ1 + 16*K3^2*Q3*Q7*λ1 + 16*K3^2*L^2*Q1*Q2 - 16*K3^2*L^2*Q1*Q3 - 12*K1^2*L^2*Q4*Q5 + 4*K1^2*L^2*Q4*Q6 + 4*K2^2*L^2*Q4*Q5 - 12*K2^2*L^2*Q4*Q6 - 16*K3^2*Q1*Q2*λ1^2 + 16*K3^2*Q1*Q3*λ1^2 + 12*K1^2*Q4*Q5*λ1^2 - 4*K1^2*Q4*Q6*λ1^2 - 4*K2^2*Q4*Q5*λ1^2 + 12*K2^2*Q4*Q6*λ1^2 - 8*K4*K5*Q1*Q2 - 8*K4*K5*Q1*Q3 - 12*K1*K4*Q1*Q8 - 12*K1*K4*Q2*Q7 + 4*K1*K4*Q1*Q9 + 4*K1*K4*Q3*Q7 + 4*K1*K5*Q1*Q8 + 4*K1*K5*Q2*Q7 + 4*K2*K4*Q1*Q8 + 4*K2*K4*Q2*Q7 + 4*K1*K5*Q1*Q9 + 4*K1*K5*Q3*Q7 + 4*K2*K4*Q1*Q9 + 4*K2*K4*Q3*Q7 + 4*K2*K5*Q1*Q8 + 4*K2*K5*Q2*Q7 - 12*K2*K5*Q1*Q9 - 12*K2*K5*Q3*Q7 - 8*K1*K2*Q7*Q8 + 16*K3*K4*Q4*Q7 - 8*K4*K5*Q4*Q5 - 8*K1*K2*Q7*Q9 - 16*K3*K5*Q4*Q7 - 8*K4*K5*Q4*Q6 - 12*K3*K4*Q5*Q8 + 4*K3*K4*Q5*Q9 + 4*K3*K4*Q6*Q8 + 4*K3*K5*Q5*Q8 + 4*K3*K4*Q6*Q9 + 4*K3*K5*Q5*Q9 + 4*K3*K5*Q6*Q8 - 12*K3*K5*Q6*Q9 + 16*K3*K4*Q1*Q4*λ1 - 16*K3*K5*Q1*Q4*λ1 + 24*K1*K4*Q4*Q5*λ1 - 12*K3*K4*Q2*Q5*λ1 + 16*K1*K3*Q4*Q7*λ1 - 8*K1*K4*Q4*Q6*λ1 - 8*K1*K5*Q4*Q5*λ1 - 8*K2*K4*Q4*Q5*λ1 + 4*K3*K4*Q2*Q6*λ1 + 4*K3*K4*Q3*Q5*λ1 + 4*K3*K5*Q2*Q5*λ1 - 8*K1*K5*Q4*Q6*λ1 - 16*K2*K3*Q4*Q7*λ1 - 8*K2*K4*Q4*Q6*λ1 - 8*K2*K5*Q4*Q5*λ1 + 4*K3*K4*Q3*Q6*λ1 + 4*K3*K5*Q2*Q6*λ1 + 4*K3*K5*Q3*Q5*λ1 - 12*K1*K3*Q5*Q8*λ1 + 24*K2*K5*Q4*Q6*λ1 - 12*K3*K5*Q3*Q6*λ1 + 4*K1*K3*Q5*Q9*λ1 + 4*K1*K3*Q6*Q8*λ1 + 4*K2*K3*Q5*Q8*λ1 + 4*K1*K3*Q6*Q9*λ1 + 4*K2*K3*Q5*Q9*λ1 + 4*K2*K3*Q6*Q8*λ1 - 12*K2*K3*Q6*Q9*λ1 - 16*K1*K3*L^2*Q1*Q4 + 16*K2*K3*L^2*Q1*Q4 + 12*K1*K3*L^2*Q2*Q5 + 8*K1*K2*L^2*Q4*Q5 - 4*K1*K3*L^2*Q2*Q6 - 4*K1*K3*L^2*Q3*Q5 - 4*K2*K3*L^2*Q2*Q5 + 8*K1*K2*L^2*Q4*Q6 - 4*K1*K3*L^2*Q3*Q6 - 4*K2*K3*L^2*Q2*Q6 - 4*K2*K3*L^2*Q3*Q5 + 12*K2*K3*L^2*Q3*Q6 + 16*K1*K3*Q1*Q4*λ1^2 - 16*K2*K3*Q1*Q4*λ1^2 - 12*K1*K3*Q2*Q5*λ1^2 - 8*K1*K2*Q4*Q5*λ1^2 + 4*K1*K3*Q2*Q6*λ1^2 + 4*K1*K3*Q3*Q5*λ1^2 + 4*K2*K3*Q2*Q5*λ1^2 - 8*K1*K2*Q4*Q6*λ1^2 + 4*K1*K3*Q3*Q6*λ1^2 + 4*K2*K3*Q2*Q6*λ1^2 + 4*K2*K3*Q3*Q5*λ1^2 - 12*K2*K3*Q3*Q6*λ1^2

C4 = 8*K1^2*L^2*Q4^2 - 6*K1^2*L^2*Q5^2 + 2*K1^2*L^2*Q6^2 - 8*K1^2*Q4^2*λ1^2 + 6*K1^2*Q5^2*λ1^2 - 2*K1^2*Q6^2*λ1^2 - 8*K1^2*Q7^2 + 6*K1^2*Q8^2 - 2*K1^2*Q9^2 + 8*K1*K2*L^2*Q5*Q6 - 8*K1*K2*Q5*Q6*λ1^2 - 8*K1*K2*Q8*Q9 - 16*K1*K3*L^2*Q1*Q5 - 16*K1*K3*L^2*Q2*Q4 + 16*K1*K3*Q1*Q5*λ1^2 + 16*K1*K3*Q2*Q4*λ1^2 + 16*K1*K3*Q4*Q8*λ1 + 16*K1*K3*Q5*Q7*λ1 + 16*K1*K4*Q1*Q7 - 12*K1*K4*Q2*Q8 + 4*K1*K4*Q3*Q9 - 16*K1*K4*Q4^2*λ1 + 12*K1*K4*Q5^2*λ1 - 4*K1*K4*Q6^2*λ1 + 4*K1*K5*Q2*Q9 + 4*K1*K5*Q3*Q8 - 8*K1*K5*Q5*Q6*λ1 - 8*K2^2*L^2*Q4^2 + 2*K2^2*L^2*Q5^2 - 6*K2^2*L^2*Q6^2 + 8*K2^2*Q4^2*λ1^2 - 2*K2^2*Q5^2*λ1^2 + 6*K2^2*Q6^2*λ1^2 + 8*K2^2*Q7^2 - 2*K2^2*Q8^2 + 6*K2^2*Q9^2 + 16*K2*K3*L^2*Q1*Q6 + 16*K2*K3*L^2*Q3*Q4 - 16*K2*K3*Q1*Q6*λ1^2 - 16*K2*K3*Q3*Q4*λ1^2 - 16*K2*K3*Q4*Q9*λ1 - 16*K2*K3*Q6*Q7*λ1 + 4*K2*K4*Q2*Q9 + 4*K2*K4*Q3*Q8 - 8*K2*K4*Q5*Q6*λ1 - 16*K2*K5*Q1*Q7 + 4*K2*K5*Q2*Q8 - 12*K2*K5*Q3*Q9 + 16*K2*K5*Q4^2*λ1 -

4*K2*K5*Q5^2*λ1 + 12*K2*K5*Q6^2*λ1 - 16*K3^2*L^2*Q1^2 + 8*K3^2*L^2*Q2^2 -
8*K3^2*L^2*Q3^2 + 16*K3^2*Q1^2*λ1^2 + 32*K3^2*Q1*Q7*λ1 - 8*K3^2*Q2^2*λ1^2 -
16*K3^2*Q2*Q8*λ1 + 8*K3^2*Q3^2*λ1^2 + 16*K3^2*Q3*Q9*λ1 + 16*K3^2*Q7^2 - 8*K3^2*Q8^2 +
8*K3^2*Q9^2 + 16*K3*K4*Q1*Q5*λ1 + 16*K3*K4*Q2*Q4*λ1 + 16*K3*K4*Q4*Q8 + 16*K3*K4*Q5*Q7
- 16*K3*K5*Q1*Q6*λ1 - 16*K3*K5*Q3*Q4*λ1 - 16*K3*K5*Q4*Q9 - 16*K3*K5*Q6*Q7 - 8*K4^2*Q1^2
+ 6*K4^2*Q2^2 - 2*K4^2*Q3^2 - 8*K4^2*Q4^2 + 6*K4^2*Q5^2 - 2*K4^2*Q6^2 - 8*K4*K5*Q2*Q3 -
8*K4*K5*Q5*Q6 + 8*K5^2*Q1^2 - 2*K5^2*Q2^2 + 6*K5^2*Q3^2 + 8*K5^2*Q4^2 - 2*K5^2*Q5^2 +
6*K5^2*Q6^2

C3 = 4*K5^2*Q1*Q2 - 4*K4^2*Q1*Q3 - 12*K4^2*Q1*Q2 + 12*K5^2*Q1*Q3 - 12*K4^2*Q4*Q5 -
4*K4^2*Q4*Q6 + 4*K5^2*Q4*Q5 + 12*K5^2*Q4*Q6 - 12*K1^2*Q7*Q8 - 4*K1^2*Q7*Q9 +
4*K2^2*Q7*Q8 + 12*K2^2*Q7*Q9 + 16*K3^2*Q7*Q8 + 16*K3^2*Q7*Q9 + 16*K3^2*Q1*Q8*λ1 +
16*K3^2*Q2*Q7*λ1 + 16*K3^2*Q1*Q9*λ1 + 16*K3^2*Q3*Q7*λ1 - 16*K3^2*L^2*Q1*Q2 -
16*K3^2*L^2*Q1*Q3 + 12*K1^2*L^2*Q4*Q5 + 4*K1^2*L^2*Q4*Q6 - 4*K2^2*L^2*Q4*Q5 -
12*K2^2*L^2*Q4*Q6 + 16*K3^2*Q1*Q2*λ1^2 + 16*K3^2*Q1*Q3*λ1^2 - 12*K1^2*Q4*Q5*λ1^2 -
4*K1^2*Q4*Q6*λ1^2 + 4*K2^2*Q4*Q5*λ1^2 + 12*K2^2*Q4*Q6*λ1^2 - 8*K4*K5*Q1*Q2 +
8*K4*K5*Q1*Q3 + 12*K1*K4*Q1*Q8 + 12*K1*K4*Q2*Q7 + 4*K1*K4*Q1*Q9 + 4*K1*K4*Q3*Q7 +
4*K1*K5*Q1*Q8 + 4*K1*K5*Q2*Q7 + 4*K2*K4*Q1*Q8 + 4*K2*K4*Q2*Q7 - 4*K1*K5*Q1*Q9 -
4*K1*K5*Q3*Q7 - 4*K2*K4*Q1*Q9 - 4*K2*K4*Q3*Q7 - 4*K2*K5*Q1*Q8 - 4*K2*K5*Q2*Q7 -
12*K2*K5*Q1*Q9 - 12*K2*K5*Q3*Q7 - 8*K1*K2*Q7*Q8 - 16*K3*K4*Q4*Q7 - 8*K4*K5*Q4*Q5 +
8*K1*K2*Q7*Q9 - 16*K3*K5*Q4*Q7 + 8*K4*K5*Q4*Q6 + 12*K3*K4*Q5*Q8 + 4*K3*K4*Q5*Q9 +
4*K3*K4*Q6*Q8 + 4*K3*K5*Q5*Q8 - 4*K3*K4*Q6*Q9 - 4*K3*K5*Q5*Q9 - 4*K3*K5*Q6*Q8 -
12*K3*K5*Q6*Q9 - 16*K3*K4*Q1*Q4*λ1 - 16*K3*K5*Q1*Q4*λ1 - 24*K1*K4*Q4*Q5*λ1 +
12*K3*K4*Q2*Q5*λ1 - 16*K1*K3*Q4*Q7*λ1 - 8*K1*K4*Q4*Q6*λ1 - 8*K1*K5*Q4*Q5*λ1 -
8*K2*K4*Q4*Q5*λ1 + 4*K3*K4*Q2*Q6*λ1 + 4*K3*K4*Q3*Q5*λ1 + 4*K3*K5*Q2*Q5*λ1 +
8*K1*K5*Q4*Q6*λ1 - 16*K2*K3*Q4*Q7*λ1 + 8*K2*K4*Q4*Q6*λ1 + 8*K2*K5*Q4*Q5*λ1 -
4*K3*K4*Q3*Q6*λ1 - 4*K3*K5*Q2*Q6*λ1 - 4*K3*K5*Q3*Q5*λ1 + 12*K1*K3*Q5*Q8*λ1 +
24*K2*K5*Q4*Q6*λ1 - 12*K3*K5*Q3*Q6*λ1 + 4*K1*K3*Q5*Q9*λ1 + 4*K1*K3*Q6*Q8*λ1 +
4*K2*K3*Q5*Q8*λ1 - 4*K1*K3*Q6*Q9*λ1 - 4*K2*K3*Q5*Q9*λ1 - 4*K2*K3*Q6*Q8*λ1 -
12*K2*K3*Q6*Q9*λ1 + 16*K1*K3*L^2*Q1*Q4 + 16*K2*K3*L^2*Q1*Q4 - 12*K1*K3*L^2*Q2*Q5 +
8*K1*K2*L^2*Q4*Q5 - 4*K1*K3*L^2*Q2*Q6 - 4*K1*K3*L^2*Q3*Q5 - 4*K2*K3*L^2*Q2*Q5 -
8*K1*K2*L^2*Q4*Q6 + 4*K1*K3*L^2*Q3*Q6 + 4*K2*K3*L^2*Q2*Q6 + 4*K2*K3*L^2*Q3*Q5 +
12*K2*K3*L^2*Q3*Q6 - 16*K1*K3*Q1*Q4*λ1^2 - 16*K2*K3*Q1*Q4*λ1^2 + 12*K1*K3*Q2*Q5*λ1^2 -
8*K1*K2*Q4*Q5*λ1^2 + 4*K1*K3*Q2*Q6*λ1^2 + 4*K1*K3*Q3*Q5*λ1^2 + 4*K2*K3*Q2*Q5*λ1^2 +
8*K1*K2*Q4*Q6*λ1^2 - 4*K1*K3*Q3*Q6*λ1^2 - 4*K2*K3*Q2*Q6*λ1^2 - 4*K2*K3*Q3*Q5*λ1^2 -
12*K2*K3*Q3*Q6*λ1^2

C2 = - 4*K1^2*L^2*Q4^2 + 4*K1^2*L^2*Q5^2 + 4*K1^2*L^2*Q5*Q6 + 4*K1^2*Q4^2*λ1^2 -
4*K1^2*Q5^2*λ1^2 - 4*K1^2*Q5*Q6*λ1^2 + 4*K1^2*Q7^2 - 4*K1^2*Q8^2 - 4*K1^2*Q8*Q9 -
8*K1*K2*L^2*Q4^2 + 4*K1*K2*L^2*Q5^2 - 4*K1*K2*L^2*Q6^2 + 8*K1*K2*Q4^2*λ1^2 -
4*K1*K2*Q5^2*λ1^2 + 4*K1*K2*Q6^2*λ1^2 + 8*K1*K2*Q7^2 - 4*K1*K2*Q8^2 + 4*K1*K2*Q9^2 +
8*K1*K3*L^2*Q1*Q5 + 8*K1*K3*L^2*Q1*Q6 + 8*K1*K3*L^2*Q2*Q4 + 8*K1*K3*L^2*Q3*Q4 -
8*K1*K3*Q1*Q5*λ1^2 - 8*K1*K3*Q1*Q6*λ1^2 - 8*K1*K3*Q2*Q4*λ1^2 - 8*K1*K3*Q3*Q4*λ1^2 -
8*K1*K3*Q4*Q8*λ1 - 8*K1*K3*Q4*Q9*λ1 - 8*K1*K3*Q5*Q7*λ1 - 8*K1*K3*Q6*Q7*λ1 -
8*K1*K4*Q1*Q7 + 8*K1*K4*Q2*Q8 + 4*K1*K4*Q2*Q9 + 4*K1*K4*Q3*Q8 + 8*K1*K4*Q4^2*λ1 -
8*K1*K4*Q5^2*λ1 - 8*K1*K4*Q5*Q6*λ1 - 8*K1*K5*Q1*Q7 + 4*K1*K5*Q2*Q8 - 4*K1*K5*Q3*Q9 +
8*K1*K5*Q4^2*λ1 - 4*K1*K5*Q5^2*λ1 + 4*K1*K5*Q6^2*λ1 - 4*K2^2*L^2*Q4^2 - 4*K2^2*L^2*Q5*Q6
- 4*K2^2*L^2*Q6^2 + 4*K2^2*Q4^2*λ1^2 + 4*K2^2*Q5*Q6*λ1^2 + 4*K2^2*Q6^2*λ1^2 +
4*K2^2*Q7^2 + 4*K2^2*Q8*Q9 + 4*K2^2*Q9^2 + 8*K2*K3*L^2*Q1*Q5 + 8*K2*K3*L^2*Q1*Q6 +
8*K2*K3*L^2*Q2*Q4 + 8*K2*K3*L^2*Q3*Q4 - 8*K2*K3*Q1*Q5*λ1^2 - 8*K2*K3*Q1*Q6*λ1^2 -
8*K2*K3*Q2*Q4*λ1^2 - 8*K2*K3*Q3*Q4*λ1^2 - 8*K2*K3*Q4*Q8*λ1 - 8*K2*K3*Q4*Q9*λ1 -
8*K2*K3*Q5*Q7*λ1 - 8*K2*K3*Q6*Q7*λ1 - 8*K2*K4*Q1*Q7 + 4*K2*K4*Q2*Q8 - 4*K2*K4*Q3*Q9 +
8*K2*K4*Q4^2*λ1 - 4*K2*K4*Q5^2*λ1 + 4*K2*K4*Q6^2*λ1 - 8*K2*K5*Q1*Q7 - 4*K2*K5*Q2*Q9 -
4*K2*K5*Q3*Q8 - 8*K2*K5*Q3*Q9 + 8*K2*K5*Q4^2*λ1 + 8*K2*K5*Q5*Q6*λ1 + 8*K2*K5*Q6^2*λ1 -
4*K3^2*L^2*Q2^2 - 8*K3^2*L^2*Q2*Q3 - 4*K3^2*L^2*Q3^2 + 4*K3^2*Q2^2*λ1^2 +
8*K3^2*Q2*Q3*λ1^2 + 8*K3^2*Q2*Q8*λ1 + 8*K3^2*Q2*Q9*λ1 + 4*K3^2*Q3^2*λ1^2 +

8*K3^2*Q3*Q8*λ1 + 8*K3^2*Q3*Q9*λ1 + 4*K3^2*Q8^2 + 8*K3^2*Q8*Q9 + 4*K3^2*Q9^2 -
8*K3*K4*Q1*Q5*λ1 - 8*K3*K4*Q1*Q6*λ1 - 8*K3*K4*Q2*Q4*λ1 - 8*K3*K4*Q3*Q4*λ1 -
8*K3*K4*Q4*Q8 - 8*K3*K4*Q4*Q9 - 8*K3*K4*Q5*Q7 - 8*K3*K4*Q6*Q7 - 8*K3*K5*Q1*Q5*λ1 -
8*K3*K5*Q1*Q6*λ1 - 8*K3*K5*Q2*Q4*λ1 - 8*K3*K5*Q3*Q4*λ1 - 8*K3*K5*Q4*Q8 - 8*K3*K5*Q4*Q9
- 8*K3*K5*Q5*Q7 - 8*K3*K5*Q6*Q7 + 4*K4^2*Q1^2 - 4*K4^2*Q2^2 - 4*K4^2*Q2*Q3 + 4*K4^2*Q4^2
- 4*K4^2*Q5^2 - 4*K4^2*Q5*Q6 + 8*K4*K5*Q1^2 - 4*K4*K5*Q2^2 + 4*K4*K5*Q3^2 + 8*K4*K5*Q4^2 -
4*K4*K5*Q5^2 + 4*K4*K5*Q6^2 + 4*K5^2*Q1^2 + 4*K5^2*Q2*Q3 + 4*K5^2*Q3^2 + 4*K5^2*Q4^2 +
4*K5^2*Q5*Q6 + 4*K5^2*Q6^2

C1 = 4*K4^2*Q1*Q2 + 4*K4^2*Q1*Q3 + 4*K5^2*Q1*Q2 + 4*K5^2*Q1*Q3 + 4*K4^2*Q4*Q5 +
4*K4^2*Q4*Q6 + 4*K5^2*Q4*Q5 + 4*K5^2*Q4*Q6 + 4*K1^2*Q7*Q8 + 4*K1^2*Q7*Q9 +
4*K2^2*Q7*Q8 + 4*K2^2*Q7*Q9 - 4*K1^2*L^2*Q4*Q5 - 4*K1^2*L^2*Q4*Q6 - 4*K2^2*L^2*Q4*Q5 -
4*K2^2*L^2*Q4*Q6 + 4*K1^2*Q4*Q5*λ1^2 + 4*K1^2*Q4*Q6*λ1^2 + 4*K2^2*Q4*Q5*λ1^2 +
4*K2^2*Q4*Q6*λ1^2 + 8*K4*K5*Q1*Q2 + 8*K4*K5*Q1*Q3 - 4*K1*K4*Q1*Q8 - 4*K1*K4*Q2*Q7 -
4*K1*K4*Q1*Q9 - 4*K1*K4*Q3*Q7 - 4*K1*K5*Q1*Q8 - 4*K1*K5*Q2*Q7 - 4*K2*K4*Q1*Q8 -
4*K2*K4*Q2*Q7 - 4*K1*K5*Q1*Q9 - 4*K1*K5*Q3*Q7 - 4*K2*K4*Q1*Q9 - 4*K2*K4*Q3*Q7 -
4*K2*K5*Q1*Q8 - 4*K2*K5*Q2*Q7 - 4*K2*K5*Q1*Q9 - 4*K2*K5*Q3*Q7 + 8*K1*K2*Q7*Q8 +
8*K4*K5*Q4*Q5 + 8*K1*K2*Q7*Q9 + 8*K4*K5*Q4*Q6 - 4*K3*K4*Q5*Q8 - 4*K3*K4*Q5*Q9 -
4*K3*K4*Q6*Q8 - 4*K3*K5*Q5*Q8 - 4*K3*K4*Q6*Q9 - 4*K3*K5*Q5*Q9 - 4*K3*K5*Q6*Q8 -
4*K3*K5*Q6*Q9 + 8*K1*K4*Q4*Q5*λ1 - 4*K3*K4*Q2*Q5*λ1 + 8*K1*K4*Q4*Q6*λ1 +
8*K1*K5*Q4*Q5*λ1 + 8*K2*K4*Q4*Q5*λ1 - 4*K3*K4*Q2*Q6*λ1 - 4*K3*K4*Q3*Q5*λ1 -
4*K3*K5*Q2*Q5*λ1 + 8*K1*K5*Q4*Q6*λ1 + 8*K2*K4*Q4*Q6*λ1 + 8*K2*K5*Q4*Q5*λ1 -
4*K3*K4*Q3*Q6*λ1 - 4*K3*K5*Q2*Q6*λ1 - 4*K3*K5*Q3*Q5*λ1 - 4*K1*K3*Q5*Q8*λ1 +
8*K2*K5*Q4*Q6*λ1 - 4*K3*K5*Q3*Q6*λ1 - 4*K1*K3*Q5*Q9*λ1 - 4*K1*K3*Q6*Q8*λ1 -
4*K2*K3*Q5*Q8*λ1 - 4*K1*K3*Q6*Q9*λ1 - 4*K2*K3*Q5*Q9*λ1 - 4*K2*K3*Q6*Q8*λ1 -
4*K2*K3*Q6*Q9*λ1 + 4*K1*K3*L^2*Q2*Q5 - 8*K1*K2*L^2*Q4*Q5 + 4*K1*K3*L^2*Q2*Q6 +
4*K1*K3*L^2*Q3*Q5 + 4*K2*K3*L^2*Q2*Q5 - 8*K1*K2*L^2*Q4*Q6 + 4*K1*K3*L^2*Q3*Q6 +
4*K2*K3*L^2*Q2*Q6 + 4*K2*K3*L^2*Q3*Q5 + 4*K2*K3*L^2*Q3*Q6 - 4*K1*K3*Q2*Q5*λ1^2 +
8*K1*K2*Q4*Q5*λ1^2 - 4*K1*K3*Q2*Q6*λ1^2 - 4*K1*K3*Q3*Q5*λ1^2 - 4*K2*K3*Q2*Q5*λ1^2 +
8*K1*K2*Q4*Q6*λ1^2 - 4*K1*K3*Q3*Q6*λ1^2 - 4*K2*K3*Q2*Q6*λ1^2 - 4*K2*K3*Q3*Q5*λ1^2 -
4*K2*K3*Q3*Q6*λ1^2

C0 = - K1^2*L^2*Q5^2 - 2*K1^2*L^2*Q5*Q6 - K1^2*L^2*Q6^2 + K1^2*Q5^2*λ1^2 +
2*K1^2*Q5*Q6*λ1^2 + K1^2*Q6^2*λ1^2 + K1^2*Q8^2 + 2*K1^2*Q8*Q9 + K1^2*Q9^2 -
2*K1*K2*L^2*Q5^2 - 4*K1*K2*L^2*Q5*Q6 - 2*K1*K2*L^2*Q6^2 + 2*K1*K2*Q5^2*λ1^2 +
4*K1*K2*Q5*Q6*λ1^2 + 2*K1*K2*Q6^2*λ1^2 + 2*K1*K2*Q8^2 + 4*K1*K2*Q8*Q9 + 2*K1*K2*Q9^2 -
2*K1*K4*Q2*Q8 - 2*K1*K4*Q2*Q9 - 2*K1*K4*Q3*Q8 - 2*K1*K4*Q3*Q9 + 2*K1*K4*Q5^2*λ1 +
4*K1*K4*Q5*Q6*λ1 + 2*K1*K4*Q6^2*λ1 - 2*K1*K5*Q2*Q8 - 2*K1*K5*Q2*Q9 - 2*K1*K5*Q3*Q8 -
2*K1*K5*Q3*Q9 + 2*K1*K5*Q5^2*λ1 + 4*K1*K5*Q5*Q6*λ1 + 2*K1*K5*Q6^2*λ1 - K2^2*L^2*Q5^2 -
2*K2^2*L^2*Q5*Q6 - K2^2*L^2*Q6^2 + K2^2*Q5^2*λ1^2 + 2*K2^2*Q5*Q6*λ1^2 + K2^2*Q6^2*λ1^2
+ K2^2*Q8^2 + 2*K2^2*Q8*Q9 + K2^2*Q9^2 - 2*K2*K4*Q2*Q8 - 2*K2*K4*Q2*Q9 - 2*K2*K4*Q3*Q8 -
2*K2*K4*Q3*Q9 + 2*K2*K4*Q5^2*λ1 + 4*K2*K4*Q5*Q6*λ1 + 2*K2*K4*Q6^2*λ1 - 2*K2*K5*Q2*Q8 -
2*K2*K5*Q2*Q9 - 2*K2*K5*Q3*Q8 - 2*K2*K5*Q3*Q9 + 2*K2*K5*Q5^2*λ1 + 4*K2*K5*Q5*Q6*λ1 +
2*K2*K5*Q6^2*λ1 + K4^2*Q2^2 + 2*K4^2*Q2*Q3 + K4^2*Q3^2 + K4^2*Q5^2 + 2*K4^2*Q5*Q6 +
K4^2*Q6^2 + 2*K4*K5*Q2^2 + 4*K4*K5*Q2*Q3 + 2*K4*K5*Q3^2 + 2*K4*K5*Q5^2 + 4*K4*K5*Q5*Q6
+ 2*K4*K5*Q6^2 + K5^2*Q2^2 + 2*K5^2*Q2*Q3 + K5^2*Q3^2 + K5^2*Q5^2 + 2*K5^2*Q5*Q6 +
K5^2*Q6^2;

ANEXO II

1. CÓDIGO DEL PROGRAMA DE BARRA SIMPLEMENTE EMPOTRADA

1.1. INTEGRACIÓN NUMÉRICA. PROBLEMA INVERSO

```
% Algorithm that solves de Inverse Kinematic position problem of a rod
that
% is clamped at one extreme and is pinned on the other.

%%%%%%%%%%%%% GEOMETRIC DEFINITION AND MECHANICAL PROPERTIES
%%%%%%%%%%%%%
L     = 1 ;               % Rod length [m]
E     = 210e9 ;           % Young's modulus [Pa]

diam = 2e-3 ;             % Diameter of the cross-section of the rod
radi = 0.5*diam ;
I     = 0.25*pi*radi^4 ;  % Interia of the cross-section of the rod

N     = 61 ;              % Number of points in which the rod is
discretized



%%%%%%%%%%%%%%%%% POSITION AND ORIENTATION AT CLAMPED END
%%%%%%%%%%%%%%%%%
theta0 = 0 ;              % Angle of the tangent of the rod at its
clamped end [rad]
p0     = [0; 0] ;         % Position of the clamped end of the rod [m]

%%%%%%%%%%%%%%%%% POSITION AT END-POINT %%%%%%%%%%%%%%%%%
p_L    = [0.8; 0.1] ;    % Position of the pinned end of the rod [m]



%%%%%%%%%%%%%% CREATION OF THE STRUCTURE WITH ALL DATA NEEDED
%%%%%%%%%%%%%
IN.L      = L ;
IN.EI     = E*I ;
IN.N      = N ;
IN.px_0   = p0(1) ;
IN.py_0   = p0(2) ;
IN.px_end = p_L(1) ;
IN.py_end = p_L(2) ;
IN.theta0 = theta0 ;

% Guess values
IN.m0 = 0.1 ;            % Bening moment at the clamped end [N·m]
IN.n0 = [0.1; 0.1] ;      % Internal force in the rod [N]



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% SHOOTING METHOD
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
[ IN, OUT ] = IK_NewRaph( IN ) ;


if OUT.solution == 0      % If algorithm does not converge
```

```
    msgbox('Convergence problem: solution has not been reached');
    return;
end

%%%%%%%%%%%%%%%%%%%%%%%%%% GRAFIC REPRESENTATION
%%%%%%%%%%%%%%%%%%%%%%%%%%
aux1 = - OUT.m.*sin(OUT.theta) ;
aux2 =   OUT.m.*cos(OUT.theta) ;

% Deformation of the rod
figure;
subplot(3,2,[1 3 5])
axis equal; grid;
xlabel('x [m]'); ylabel('y [m]') ;
title('Inverse Kinematic position problem') ;
hold on ;
plot(OUT.px, OUT.py, '.-b', 'linewidth', 3) ;
quiver(OUT.px, OUT.py, aux1, aux2, '.-r') ;
quiver(OUT.px(end), OUT.py(end), IN.n0(1)/norm(IN.n0)*0.25,
IN.n0(2)/norm(IN.n0)*0.25, 'k') ;
legend('Deformation of the rod', 'Bending moment', 'Force direction')
;

% Evolution of the value of n_x
subplot(3,2,2) ;
grid; hold on;
xlabel('iterations');
title('Evolution n_x [N]') ;
plot(OUT.GUESS(1,:), '.-b') ;

% Evolution of the value of n_y
subplot(3,2,4) ;
grid; hold on;
xlabel('iterations');
title('Evolution n_y [N]') ;
plot(OUT.GUESS(2,:), '.-b') ;

% Evolution of the value of m0
subplot(3,2,6) ;
grid; hold on;
xlabel('iterations');
title('Evolution m_0 [N·m]') ;
plot(OUT.GUESS(3,:), '.-b') ;
```

1.2. INTEGRACIÓN NUMÉRICA. FUNCIONES ASOCIADAS

```
function [ IN, OUT ] = IK_NewRaph( IN )
% Shooting method to obtain the solution of the Inverse Kinematic
Position
% problem of a rod that is clamped at both one extreme (at it base)

% Absolute and relative tolerance of the error (residue)
tol = 1e-7 ;

% Runge-Kutta method of order 4 with the initial guess values
OUT = RK4_rod( IN ) ;

% Residue vector for the initial guess values
resid = [ OUT.px(end) - IN.px_end ;
          OUT.py(end) - IN.py_end ;
          OUT.m(end) ] ;

resid0 = zeros(size(resid)) ;

GUESS = [ IN.n0 ;
          IN.m0 ] ;

while max(abs(resid)) > tol && max(abs(resid-resid0)) > tol

    resid0 = resid ;

    % Jacobian of the residue vector respect to the variables of the
    % problme (guess values)
    J = Jacobian( IN, OUT ) ;

    % Increment of the variable to get to the new solutions with less
    % residue
    delta = -J\resid0 ;

    % Updating the value of the guess values to reach smaller values
of
    % residue vector
    IN.n0 = IN.n0 + delta(1:2) ;
    IN.m0 = IN.m0 + delta(3) ;

    % Runge-Kutta method of order 4 wiht the updated guess values
    OUT = RK4_rod( IN ) ;

    % Residue vector for the updates guess values
    resid = [ OUT.px(end) - IN.px_end ;
              OUT.py(end) - IN.py_end ;
              OUT.m(end) ] ;

    % In case the increment (delta) is very hight
    while max(abs(resid))>max(abs(resid0)) && max(abs(delta))>tol*1e-3
        delta = 0.5*delta ;

        IN.n0 = IN.n0 - delta(1:2) ;
        IN.m0 = IN.m0 - delta(3) ;

        OUT = RK4_rod( IN ) ;

        resid = [ OUT.px(end) - IN.px_end ;
```

```
                  OUT.py(end) - IN.py_end ;
                  OUT.m(end) ] ;
    end

    aux = [ IN.n0 ; IN.m0 ] ;
    GUESS = [ GUESS aux ] ;
end

% Checking if the final solution has been reached
if max(abs(resid))>tol || not(sum(abs(resid))>=0)
    OUT.solution = 0 ;
else
    OUT.solution = 1 ;
    OUT.GUESS = GUESS ;
end
```

```
function [ Jacob ] = Jacobian( IN, OUT )
% Calculation of the Jacobian of the residue vector respect to the
% variables that are treated as guess values
Jacob = zeros(3) ;


% Increment of the variables to calculate de Jacobian numerically
epsilon = 1e-10 ;

% Small increment of the x component of the internal force
IN.n0(1) = IN.n0(1) + epsilon ;
OUT_eps = RK4_rod( IN ) ;
Jacob(:,1) = [ OUT_eps.px(end) - OUT.px(end) ;
               OUT_eps.py(end) - OUT.py(end) ;
               OUT_eps.m(end)  - OUT.m(end) ] ;
IN.n0(1) = IN.n0(1) - epsilon ;

% Small increment of the y component of the internal force
IN.n0(2) = IN.n0(2) + epsilon ;
OUT_eps = RK4_rod( IN ) ;
Jacob(:,2) = [ OUT_eps.px(end) - OUT.px(end) ;
               OUT_eps.py(end) - OUT.py(end) ;
               OUT_eps.m(end)  - OUT.m(end) ] ;
IN.n0(2) = IN.n0(2) - epsilon ;

% Small increment of the bending moment
IN.m0 = IN.m0 + epsilon ;
OUT_eps = RK4_rod( IN ) ;
Jacob(:,3) = [ OUT_eps.px(end) - OUT.px(end) ;
               OUT_eps.py(end) - OUT.py(end) ;
               OUT_eps.m(end)  - OUT.m(end) ] ;

Jacob = Jacob/epsilon ;
```

```
function [ OUT ] = RK4_rod ( IN )
% Functions that solves de initial value problem using the method of
% Runge-Kutta of order four.
%   IN     Structure in which are geometric and mechanical parameters
%                IN.L      length of the rod
%                IN.n0      number of nodes of the rod
%                IN.K_SE   array (3x1) with the stiff for the linear
deformation
%                IN.K_BT   array (3x1) with the stiff for the angular
deformation
%                IN.m0     initial bending moment
%                IN.n0      vectir (2x1) that represents the force
applied to the rod
%   OUT    Structure in which are saved the solution
%                OUT.py  array (1xIN.n0) with the y component of the
centroid position in each node of the rod
%                OUT.pz  array (1xIN.n0) with the y component of the
centroid position in each node of the rod
%                OUT.m   array (1xIN.n0) with the bending moment in
each node of the rod


% Variables to save the solution
OUT.px     = zeros(1,IN.N) ;        % Array with the component y of the
centroid position vector
OUT.py     = zeros(1,IN.N) ;        % Array with the component z of the
centroid position vector
OUT.m      = zeros(1,IN.N) ;        % Array with the values of bending
moment through the lenght of the rod
OUT.theta = zeros(1,IN.N) ;         % Array with the values of the
angle between the x axis and the tangent to the rod

% Values of the storage variable at arc-length s=0 (clamped end=
OUT.px(1)     = IN.px_0 ;
OUT.py(1)     = IN.py_0 ;
OUT.theta(1) = IN.theta0 ;
OUT.m(1)      = IN.m0 ;


% Vector with the values of the dependent variables at the clamped end
var = [ IN.px_0 ;       % px at the clamped end
        IN.py_0 ;       % py at the clamped end
        IN.theta0 ;     % angle of the rod at the clamped end
        IN.m0 ] ;       % guess value of the moment at the clamped end

% Increment of length
ds = IN.L/(IN.N-1) ;


% "For" loop to integrate the system of differential equations
for ii = 2:IN.N

    % Rugne-Kutta of order four
    k1 = ds *  Right_function( IN.EI, IN.n0, var ) ;
    k2 = ds *  Right_function( IN.EI, IN.n0, var + 0.5*k1 ) ;
    k3 = ds *  Right_function( IN.EI, IN.n0, var + 0.5*k2 ) ;
    k4 = ds *  Right_function( IN.EI, IN.n0, var + k3 ) ;

    var = var + (k1 + 2*k2 + 2*k3 + k4)/6 ;
```

```
    OUT.px(ii)     = var(1) ;    % x component of centroid position
vector in node ii
    OUT.py(ii)     = var(2) ;    % y component of centroid position
vector in node ii
    OUT.m(ii)      = var(4) ;    % bending moment in node ii
    OUT.theta(ii) = var(3) ;    % angle in node ii

end
```

```
function [ func ] = Right_function( EI, n, var )
% Value of the right part of the system of differential equations
% var(1)    px
% var(2)    py
% var(3)    theta
% var(4)    moment

func = [ cos(var(3)) ;
         sin(var(3)) ;
         var(4)/EI ;
         n(1)*sin(var(3)) - n(2)*cos(var(3)) ] ;
```

2. CÓDIGO DE LOS PROGRAMAS DEL MECANISMO 3PRR

2.1. PROBLEMA INVERSO 3PRR

```
%Introduciendo geometría y posición del mecanismo paralelo de 3gdl,
calcula
%la solución del problema inverso, si la tiene
clc;

%% Introducción de datos

% Posición del elemento triangular (xp,yp,psi)
xp = 3 ;
yp = 0.3 ;
psigrad = 45 ;           % Ángulo en grados
psi = psigrad*pi/180;    % Ángulo en rad para trabajar



% Datos geométricos (L,a,B2,B3,H3)
L  = 1;
a  = 0.3;
B2 = 3;
B3 = 3;
H3 = 3;

disp('MECANISMO PARALELO DE 3 GDL');
disp(' ');

disp('Parámetros geométricos del mecanismo:');
disp(['Longitud de las barras, L(m): ',num2str(L)]);
disp(['Lados del elemento triangular, a(m): ',num2str(a)]);
disp(['Posición en x del motor 2, B2(m): ',num2str(B2)]);
disp(['Posición en x del motor 3, B3(m): ',num2str(B3)]);
disp(['Posición en y del motor 3, H3(m): ',num2str(H3)]);
disp(' ');

disp('Posición del punto P del elemento triangular:');
disp(['Coordenada x (m): ',num2str(xp)]);
disp(['Coordenada y (m): ',num2str(yp)]);
disp(['Ángulo de inclinación (grados): ',num2str(psigrad)]);
disp(' ');



%% ¿Existe solución?
sol = Comprobar_Problema_Inverso_3PRR(L,a,B3,xp,yp,psi);


%% Si no existe solución (sol=0)
if sol == 0
    disp('El problema no tiene solución real');
end


%% Si existe solución (sol=1)
if sol == 1

    %Resolución problema inverso
```

```
    lambda = Calcular_Problema_Inverso_3PRR(L,a,B2,B3,H3,xp,yp,psi);

    % ¿POSICIÓN SINGULAR?
    if size(lambda,1) < 8
        disp('POSICIÓN SINGULAR');
    end

    %Mostrar soluciones
    disp('Combinaciones posibles de posiciones de los motores:');
    for i = 1:size(lambda,1)
        disp(['Lambda 1 = ' num2str(lambda(i,1))]);
        disp(['Lambda 2 = ' num2str(lambda(i,2))]);
        disp(['Lambda 3 = ' num2str(lambda(i,3))]);
        disp(' ');
    end


    %Representación
    fig = figure;
    figure(fig);

    if size(lambda,1) == 1    % Todas las soluciones iguales
        Plot_3PRR(a,B2,B3,H3,lambda(1),lambda(2),lambda(3),xp,yp,psi);
    else
        nsol = size(lambda,1);
        for i = 1:nsol
            subplot(nsol/2,2,i);
Plot_3PRR(a,B2,B3,H3,lambda(i,1),lambda(i,2),lambda(i,3),xp,yp,psi);
        end
    end

end




%En una región del plano dada, analiza para qué posiciones del
mecanismo el
%problema inverso tiene solución

clc;
disp('ESTUDIO DEL PLANO DE TRABAJO');
disp(' ');

%% Introducción de datos

% Espacio de trabajo
x0 = -3;
xn = 6;
n1 = 50;
y0 = -3;
yn = 6;
n2 = 50;
psi0grad = 30;
psi0     = psi0grad*pi/180;   %Ángulo en rad para trabajar


% Datos geométricos (L,a,B2,B3,H3)
```

```
L  = 1;
a  = 1;
B2 = 3;
B3 = 2;
H3 = 2;

disp('Parámetros geométricos del mecanismo:');
disp(['Longitud de las barras, L(m): ',num2str(L)]);
disp(['Lados del elemento triangular, a(m): ',num2str(a)]);
disp(['Posición en x del motor 2, B2(m): ',num2str(B2)]);
disp(['Posición en x del motor 3, B3(m): ',num2str(B3)]);
disp(['Posición en y del motor 3, H3(m): ',num2str(H3)]);
disp(' ');

disp('Región del plano a estudiar:');
disp('Coordenada x:');
disp(['Desde: ' num2str(x0)]);
disp(['Hasta: ' num2str(xn)]);
disp(['N° de divisiones: ' num2str(n1)]);
disp('Coordenada y:');
disp(['Desde: ' num2str(y0)]);
disp(['Hasta: ' num2str(yn)]);
disp(['N° de divisiones: ' num2str(n2)]);
disp(' ');
disp(['Ángulo de orientación (grados): ' num2str(psi0grad)]);
disp(' ');




%% Discretización

[X,Y] = meshgrid(x0:(xn-x0)/n1:xn,y0:(yn-y0)/n2:yn);
psi = 0:0.15*pi:2*pi;   %Vector de ángulos de orientación del elemento
triangular
psigrad = (psi.*180)./pi;


%% Representación

%En cada pto de la red, y para cada orientación, comprobar si el
problema
%inverso tiene solución, y si tiene, representar dicho pto

fig1 = figure;
figure(fig1);

%Representación en 3 dimensiones del espacio de trabajo para cada
orientación
xrep1 = [];    %Inicializar vectores de ptos representados
yrep1 = [];
psirep1 = [];

% subplot(3,2,[1 3 5]);
for i = 1:size(X,1)
    for j = 1:size(X,2)
        for k = 1:length(psi)
            sol =
Comprobar_Problema_Inverso_3PRR(L,a,B3,X(i,j),Y(i,j),psi(k));
```

```
            if sol == 1
                xrep1  = [xrep1 X(i,j)];    %Añadir el pto al vector
de representación
                yrep1  = [yrep1 Y(i,j)];
                psirep1 = [psirep1 psigrad(k)];
            end
        end
    end
end
plot3(xrep1,yrep1,psirep1,'.r');
hold on;
% k1 = boundary(xrep1.',yrep1.',firep1.',1);
% trisurf(k1,xrep1,yrep1,firep1,'Facecolor','green','FaceAlpha',0.4);
daspect([1 1 100]);
grid;
xlabel('X');
ylabel('Y');
zlabel('Orientación (\psi)');
hold off;


%Representación del espacio de trabajo para una orientación fi0
xrep2 = [];  %Inicializar vectores de ptos representados
yrep2 = [];


fig2 = figure ;
figure(fig2) ;
% subplot(3,2,4);
for i = 1:size(X,1)
    for j = 1:size(X,2)
        sol =
Comprobar_Problema_Inverso_3PRR(L,a,B3,X(i,j),Y(i,j),psi0);
        if sol == 1
            xrep2 = [xrep2 X(i,j)];   %Añadir el pto al vector de
representación
            yrep2 = [yrep2 Y(i,j)];
        end
    end
end
plot(xrep2,yrep2,'.r');
% k2 = boundary(xrep2.',yrep2.');    %Frontera
hold on;
% plot(xrep2(k2),yrep2(k2),'g');
title(['\psi (grados)=',num2str(psi0grad)]);
axis equal;
xlabel('X');
ylabel('Y');
hold off;
```

2.2. PROBLEMA DIRECTO 3PRR

```
clear; clc;

%% Entrada: lambda1, lambda2, lambda3
lambda1 = 0.4 ;
lambda2 = 0.1 ;
lambda3 = 0.6 ;

disp('Posición de los motores');
disp(['Lambda 1: ', num2str(lambda1)]);
disp(['Lambda 2: ', num2str(lambda2)]);
disp(['Lambda 3: ', num2str(lambda3)]);

% meto una solución del problema para evaluar el residuo del polinomio
% univariante
% Psi = input('Psi prueba (grados): ');
% T=tan(Psi*pi/360);


%% Datos geomÃ©tricos: a, L, B2, B3, H3
L = 1;
a = 1;
B2 = 3;
B3 = 2;
H3 = 3;




%% Desarrollo matemÃ¡tico

% Llamo spsi y cpsi a los senos y cosenos de psi

% Cambio de variable
% spsi = 2 * t/(1 + t^2);      %sin(psi)
% cpsi = (1 - t^2)/(1 + t^2);      %cos(psi)

K1 = 2 * a;
K2 = 2 * lambda1 + 2*(lambda2 - B2);
K3 = 2 * a;
K4 = 2 * a * (lambda2 - B2);
K5 = -lambda1^2 + a^2 + (lambda2 - B2)^2;

% f1 = K1 * cpsi + K2;
% f2 = K3 * spsi;
% f3 = K4 * cpsi + K5;

Q1 = -a * sqrt(3);
Q2 = a;
Q3 = 2 * lambda1 - 2 * B3;
Q4 = a;
Q5 = a * sqrt(3);
Q6 = 2 * (lambda3 - H3);
Q7 = a * (lambda3 - H3) + a * sqrt(3) * B3;
Q8 = a * sqrt(3) * (lambda3 - H3) - a * B3;
Q9 = -lambda1^2 + B3^2 + a^2 + (lambda3 - H3)^2;

% g1 = Q1 * spsi + Q2 * cpsi + Q3;
```

```
% g2 = Q4 * spsi + Q5 * cpsi + Q6;
% g3 = Q7 * spsi + Q8 * cpsi + Q9;

% xp = h1 = (f3 * g2 - f2 * g3)/(f2 * g1 - f1 * g2);
% yp = h2 = (f1 * g3 - f3 * g1)/(f2 * g1 - f1 * g2);


% EcuaciÃ³n a resolver: (h1 - lambda1)^2 + h2^2 - L^2 = 0

% Polinomio en t: C0 + C1*t + C2*t^2 + C3*t^3 + C4*t^4 + C5*t^5 +
C6*t^6 +
% C7*t^7 + C8*t^8 (desarrollo en un script aparte)

C8 = - K1^2*L^2*Q5^2 + 2*K1^2*L^2*Q5*Q6 - K1^2*L^2*Q6^2 +
K1^2*Q5^2*lambda1^2 - 2*K1^2*Q5*Q6*lambda1^2 + K1^2*Q6^2*lambda1^2 +
K1^2*Q8^2 - 2*K1^2*Q8*Q9 + K1^2*Q9^2 + 2*K1*K2*L^2*Q5^2 -
4*K1*K2*L^2*Q5*Q6 + 2*K1*K2*L^2*Q6^2 - 2*K1*K2*Q5^2*lambda1^2 +
4*K1*K2*Q5*Q6*lambda1^2 - 2*K1*K2*Q6^2*lambda1^2 - 2*K1*K2*Q8^2 +
4*K1*K2*Q8*Q9 - 2*K1*K2*Q9^2 - 2*K1*K4*Q2*Q8 + 2*K1*K4*Q2*Q9 +
2*K1*K4*Q3*Q8 - 2*K1*K4*Q3*Q9 + 2*K1*K4*Q5^2*lambda1 -
4*K1*K4*Q5*Q6*lambda1 + 2*K1*K4*Q6^2*lambda1 + 2*K1*K5*Q2*Q8 -
2*K1*K5*Q2*Q9 - 2*K1*K5*Q3*Q8 + 2*K1*K5*Q3*Q9 - 2*K1*K5*Q5^2*lambda1 +
4*K1*K5*Q5*Q6*lambda1 - 2*K1*K5*Q6^2*lambda1 - K2^2*L^2*Q5^2 +
2*K2^2*L^2*Q5*Q6 - K2^2*L^2*Q6^2 + K2^2*Q5^2*lambda1^2 -
2*K2^2*Q5*Q6*lambda1^2 + K2^2*Q6^2*lambda1^2 + K2^2*Q8^2 -
2*K2^2*Q8*Q9 + K2^2*Q9^2 + 2*K2*K4*Q2*Q8 - 2*K2*K4*Q2*Q9 -
2*K2*K4*Q3*Q8 + 2*K2*K4*Q3*Q9 - 2*K2*K4*Q5^2*lambda1 +
4*K2*K4*Q5*Q6*lambda1 - 2*K2*K4*Q6^2*lambda1 - 2*K2*K5*Q2*Q8 +
2*K2*K5*Q2*Q9 + 2*K2*K5*Q3*Q8 - 2*K2*K5*Q3*Q9 + 2*K2*K5*Q5^2*lambda1 -
4*K2*K5*Q5*Q6*lambda1 + 2*K2*K5*Q6^2*lambda1 + K4^2*Q2^2 -
2*K4^2*Q2*Q3 + K4^2*Q3^2 + K4^2*Q5^2 - 2*K4^2*Q5*Q6 + K4^2*Q6^2 -
2*K4*K5*Q2^2 + 4*K4*K5*Q2*Q3 - 2*K4*K5*Q3^2 - 2*K4*K5*Q5^2 +
4*K4*K5*Q5*Q6 - 2*K4*K5*Q6^2 + K5^2*Q2^2 - 2*K5^2*Q2*Q3 + K5^2*Q3^2 +
K5^2*Q5^2 - 2*K5^2*Q5*Q6 + K5^2*Q6^2;
C7 = 4*K4^2*Q1*Q3 - 4*K4^2*Q1*Q2 - 4*K5^2*Q1*Q2 + 4*K5^2*Q1*Q3 -
4*K4^2*Q4*Q5 + 4*K4^2*Q4*Q6 - 4*K5^2*Q4*Q5 + 4*K5^2*Q4*Q6 -
4*K1^2*Q7*Q8 + 4*K1^2*Q7*Q9 - 4*K2^2*Q7*Q8 + 4*K2^2*Q7*Q9 +
4*K1^2*L^2*Q4*Q5 - 4*K1^2*L^2*Q4*Q6 + 4*K2^2*L^2*Q4*Q5 -
4*K2^2*L^2*Q4*Q6 - 4*K1^2*Q4*Q5*lambda1^2 + 4*K1^2*Q4*Q6*lambda1^2 -
4*K2^2*Q4*Q5*lambda1^2 + 4*K2^2*Q4*Q6*lambda1^2 + 8*K4*K5*Q1*Q2 -
8*K4*K5*Q1*Q3 + 4*K1*K4*Q1*Q8 + 4*K1*K4*Q2*Q7 - 4*K1*K4*Q1*Q9 -
4*K1*K4*Q3*Q7 - 4*K1*K5*Q1*Q8 - 4*K1*K5*Q2*Q7 - 4*K2*K4*Q1*Q8 -
4*K2*K4*Q2*Q7 + 4*K1*K5*Q1*Q9 + 4*K1*K5*Q3*Q7 + 4*K2*K4*Q1*Q9 +
4*K2*K4*Q3*Q7 + 4*K2*K5*Q1*Q8 + 4*K2*K5*Q2*Q7 - 4*K2*K5*Q1*Q9 -
4*K2*K5*Q3*Q7 + 8*K1*K2*Q7*Q8 + 8*K4*K5*Q4*Q5 - 8*K1*K2*Q7*Q9 -
8*K4*K5*Q4*Q6 + 4*K3*K4*Q5*Q8 - 4*K3*K4*Q5*Q9 - 4*K3*K4*Q6*Q8 -
4*K3*K5*Q5*Q8 + 4*K3*K4*Q6*Q9 + 4*K3*K5*Q5*Q9 + 4*K3*K5*Q6*Q8 -
4*K3*K5*Q6*Q9 - 8*K1*K4*Q4*Q5*lambda1 + 4*K3*K4*Q2*Q5*lambda1 +
8*K1*K4*Q4*Q6*lambda1 + 8*K1*K5*Q4*Q5*lambda1 + 8*K2*K4*Q4*Q5*lambda1
- 4*K3*K4*Q2*Q6*lambda1 - 4*K3*K4*Q3*Q5*lambda1 -
4*K3*K5*Q2*Q5*lambda1 - 8*K1*K5*Q4*Q6*lambda1 - 8*K2*K4*Q4*Q6*lambda1
- 8*K2*K5*Q4*Q5*lambda1 + 4*K3*K4*Q3*Q6*lambda1 +
4*K3*K5*Q2*Q6*lambda1 + 4*K3*K5*Q3*Q5*lambda1 + 4*K1*K3*Q5*Q8*lambda1
+ 8*K2*K5*Q4*Q6*lambda1 - 4*K3*K5*Q3*Q6*lambda1 -
4*K1*K3*Q5*Q9*lambda1 - 4*K1*K3*Q6*Q8*lambda1 - 4*K2*K3*Q5*Q8*lambda1
+ 4*K1*K3*Q6*Q9*lambda1 + 4*K2*K3*Q5*Q9*lambda1 +
4*K2*K3*Q6*Q8*lambda1 - 4*K2*K3*Q6*Q9*lambda1 - 4*K1*K3*L^2*Q2*Q5 -
8*K1*K2*L^2*Q4*Q5 + 4*K1*K3*L^2*Q2*Q6 + 4*K1*K3*L^2*Q3*Q5 +
4*K2*K3*L^2*Q2*Q5 + 8*K1*K2*L^2*Q4*Q6 - 4*K1*K3*L^2*Q3*Q6 -
4*K2*K3*L^2*Q2*Q6 - 4*K2*K3*L^2*Q3*Q5 + 4*K2*K3*L^2*Q3*Q6 +
```

```
4*K1*K3*Q2*Q5*lambda1^2 + 8*K1*K2*Q4*Q5*lambda1^2 -
4*K1*K3*Q2*Q6*lambda1^2 - 4*K1*K3*Q3*Q5*lambda1^2 -
4*K2*K3*Q2*Q5*lambda1^2 - 8*K1*K2*Q4*Q6*lambda1^2 +
4*K1*K3*Q3*Q6*lambda1^2 + 4*K2*K3*Q2*Q6*lambda1^2 +
4*K2*K3*Q3*Q5*lambda1^2 - 4*K2*K3*Q3*Q6*lambda1^2;
C6 = - 4*K1^2*L^2*Q4^2 + 4*K1^2*L^2*Q5^2 - 4*K1^2*L^2*Q5*Q6 +
4*K1^2*Q4^2*lambda1^2 - 4*K1^2*Q5^2*lambda1^2 + 4*K1^2*Q5*Q6*lambda1^2
+ 4*K1^2*Q7^2 - 4*K1^2*Q8^2 + 4*K1^2*Q8*Q9 + 8*K1*K2*L^2*Q4^2 -
4*K1*K2*L^2*Q5^2 + 4*K1*K2*L^2*Q6^2 - 8*K1*K2*Q4^2*lambda1^2 +
4*K1*K2*Q5^2*lambda1^2 - 4*K1*K2*Q6^2*lambda1^2 - 8*K1*K2*Q7^2 +
4*K1*K2*Q8^2 - 4*K1*K2*Q9^2 + 8*K1*K3*L^2*Q1*Q5 - 8*K1*K3*L^2*Q1*Q6 +
8*K1*K3*L^2*Q2*Q4 - 8*K1*K3*L^2*Q3*Q4 - 8*K1*K3*Q1*Q5*lambda1^2 +
8*K1*K3*Q1*Q6*lambda1^2 - 8*K1*K3*Q2*Q4*lambda1^2 +
8*K1*K3*Q3*Q4*lambda1^2 - 8*K1*K3*Q4*Q8*lambda1 +
8*K1*K3*Q4*Q9*lambda1 - 8*K1*K3*Q5*Q7*lambda1 + 8*K1*K3*Q6*Q7*lambda1
- 8*K1*K4*Q1*Q7 + 8*K1*K4*Q2*Q8 - 4*K1*K4*Q2*Q9 - 4*K1*K4*Q3*Q8 +
8*K1*K4*Q4^2*lambda1 - 8*K1*K4*Q5^2*lambda1 + 8*K1*K4*Q5*Q6*lambda1 +
8*K1*K5*Q1*Q7 - 4*K1*K5*Q2*Q8 + 4*K1*K5*Q3*Q9 - 8*K1*K5*Q4^2*lambda1 +
4*K1*K5*Q5^2*lambda1 - 4*K1*K5*Q6^2*lambda1 - 4*K2^2*L^2*Q4^2 +
4*K2^2*L^2*Q5*Q6 - 4*K2^2*L^2*Q6^2 + 4*K2^2*Q4^2*lambda1^2 -
4*K2^2*Q5*Q6*lambda1^2 + 4*K2^2*Q6^2*lambda1^2 + 4*K2^2*Q7^2 -
4*K2^2*Q8*Q9 + 4*K2^2*Q9^2 - 8*K2*K3*L^2*Q1*Q5 + 8*K2*K3*L^2*Q1*Q6 -
8*K2*K3*L^2*Q2*Q4 + 8*K2*K3*L^2*Q3*Q4 + 8*K2*K3*Q1*Q5*lambda1^2 -
8*K2*K3*Q1*Q6*lambda1^2 + 8*K2*K3*Q2*Q4*lambda1^2 -
8*K2*K3*Q3*Q4*lambda1^2 + 8*K2*K3*Q4*Q8*lambda1 -
8*K2*K3*Q4*Q9*lambda1 + 8*K2*K3*Q5*Q7*lambda1 - 8*K2*K3*Q6*Q7*lambda1
+ 8*K2*K4*Q1*Q7 - 4*K2*K4*Q2*Q8 + 4*K2*K4*Q3*Q9 - 8*K2*K4*Q4^2*lambda1
+ 4*K2*K4*Q5^2*lambda1 - 4*K2*K4*Q6^2*lambda1 - 8*K2*K5*Q1*Q7 +
4*K2*K5*Q2*Q9 + 4*K2*K5*Q3*Q8 - 8*K2*K5*Q3*Q9 + 8*K2*K5*Q4^2*lambda1 -
8*K2*K5*Q5*Q6*lambda1 + 8*K2*K5*Q6^2*lambda1 - 4*K3^2*L^2*Q2^2 +
8*K3^2*L^2*Q2*Q3 - 4*K3^2*L^2*Q3^2 + 4*K3^2*Q2^2*lambda1^2 -
8*K3^2*Q2*Q3*lambda1^2 + 8*K3^2*Q2*Q8*lambda1 - 8*K3^2*Q2*Q9*lambda1 +
4*K3^2*Q3^2*lambda1^2 - 8*K3^2*Q3*Q8*lambda1 + 8*K3^2*Q3*Q9*lambda1 +
4*K3^2*Q8^2 - 8*K3^2*Q8*Q9 + 4*K3^2*Q9^2 - 8*K3*K4*Q1*Q5*lambda1 +
8*K3*K4*Q1*Q6*lambda1 - 8*K3*K4*Q2*Q4*lambda1 + 8*K3*K4*Q3*Q4*lambda1
- 8*K3*K4*Q4*Q8 + 8*K3*K4*Q4*Q9 - 8*K3*K4*Q5*Q7 + 8*K3*K4*Q6*Q7 +
8*K3*K5*Q1*Q5*lambda1 - 8*K3*K5*Q1*Q6*lambda1 + 8*K3*K5*Q2*Q4*lambda1
- 8*K3*K5*Q3*Q4*lambda1 + 8*K3*K5*Q4*Q8 - 8*K3*K5*Q4*Q9 +
8*K3*K5*Q5*Q7 - 8*K3*K5*Q6*Q7 + 4*K4^2*Q1^2 - 4*K4^2*Q2^2 +
4*K4^2*Q2*Q3 + 4*K4^2*Q4^2 - 4*K4^2*Q5^2 + 4*K4^2*Q5*Q6 - 8*K4*K5*Q1^2
+ 4*K4*K5*Q2^2 - 4*K4*K5*Q3^2 - 8*K4*K5*Q4^2 + 4*K4*K5*Q5^2 -
4*K4*K5*Q6^2 + 4*K5^2*Q1^2 - 4*K5^2*Q2*Q3 + 4*K5^2*Q3^2 + 4*K5^2*Q4^2
- 4*K5^2*Q5*Q6 + 4*K5^2*Q6^2;
C5 = 12*K4^2*Q1*Q2 - 4*K4^2*Q1*Q3 - 4*K5^2*Q1*Q2 + 12*K5^2*Q1*Q3 +
12*K4^2*Q4*Q5 - 4*K4^2*Q4*Q6 - 4*K5^2*Q4*Q5 + 12*K5^2*Q4*Q6 +
12*K1^2*Q7*Q8 - 4*K1^2*Q7*Q9 - 4*K2^2*Q7*Q8 + 12*K2^2*Q7*Q9 -
16*K3^2*Q7*Q8 + 16*K3^2*Q7*Q9 - 16*K3^2*Q1*Q8*lambda1 -
16*K3^2*Q2*Q7*lambda1 + 16*K3^2*Q1*Q9*lambda1 + 16*K3^2*Q3*Q7*lambda1
+ 16*K3^2*L^2*Q1*Q2 - 16*K3^2*L^2*Q1*Q3 - 12*K1^2*L^2*Q4*Q5 +
4*K1^2*L^2*Q4*Q6 + 4*K2^2*L^2*Q4*Q5 - 12*K2^2*L^2*Q4*Q6 -
16*K3^2*Q1*Q2*lambda1^2 + 16*K3^2*Q1*Q3*lambda1^2 +
12*K1^2*Q4*Q5*lambda1^2 - 4*K1^2*Q4*Q6*lambda1^2 -
4*K2^2*Q4*Q5*lambda1^2 + 12*K2^2*Q4*Q6*lambda1^2 - 8*K4*K5*Q1*Q2 -
8*K4*K5*Q1*Q3 - 12*K1*K4*Q1*Q8 - 12*K1*K4*Q2*Q7 + 4*K1*K4*Q1*Q9 +
4*K1*K4*Q3*Q7 + 4*K1*K5*Q1*Q8 + 4*K1*K5*Q2*Q7 + 4*K2*K4*Q1*Q8 +
4*K2*K4*Q2*Q7 + 4*K1*K5*Q1*Q9 + 4*K1*K5*Q3*Q7 + 4*K2*K4*Q1*Q9 +
4*K2*K4*Q3*Q7 + 4*K2*K5*Q1*Q8 + 4*K2*K5*Q2*Q7 - 12*K2*K5*Q1*Q9 -
12*K2*K5*Q3*Q7 - 8*K1*K2*Q7*Q8 + 16*K3*K4*Q4*Q7 - 8*K4*K5*Q4*Q5 -
8*K1*K2*Q7*Q9 - 16*K3*K5*Q4*Q7 - 8*K4*K5*Q4*Q6 - 12*K3*K4*Q5*Q8 +
4*K3*K4*Q5*Q9 + 4*K3*K4*Q6*Q8 + 4*K3*K5*Q5*Q8 + 4*K3*K4*Q6*Q9 +
4*K3*K5*Q5*Q9 + 4*K3*K5*Q6*Q8 - 12*K3*K5*Q6*Q9 +
```

```
16*K3*K4*Q1*Q4*lambda1 - 16*K3*K5*Q1*Q4*lambda1 +
24*K1*K4*Q4*Q5*lambda1 - 12*K3*K4*Q2*Q5*lambda1 +
16*K1*K3*Q4*Q7*lambda1 - 8*K1*K4*Q4*Q6*lambda1 - 8*K1*K5*Q4*Q5*lambda1
- 8*K2*K4*Q4*Q5*lambda1 + 4*K3*K4*Q2*Q6*lambda1 +
4*K3*K4*Q3*Q5*lambda1 + 4*K3*K5*Q2*Q5*lambda1 - 8*K1*K5*Q4*Q6*lambda1
- 16*K2*K3*Q4*Q7*lambda1 - 8*K2*K4*Q4*Q6*lambda1 -
8*K2*K5*Q4*Q5*lambda1 + 4*K3*K4*Q3*Q6*lambda1 + 4*K3*K5*Q2*Q6*lambda1
+ 4*K3*K5*Q3*Q5*lambda1 - 12*K1*K3*Q5*Q8*lambda1 +
24*K2*K5*Q4*Q6*lambda1 - 12*K3*K5*Q3*Q6*lambda1 +
4*K1*K3*Q5*Q9*lambda1 + 4*K1*K3*Q6*Q8*lambda1 + 4*K2*K3*Q5*Q8*lambda1
+ 4*K1*K3*Q6*Q9*lambda1 + 4*K2*K3*Q5*Q9*lambda1 +
4*K2*K3*Q6*Q8*lambda1 - 12*K2*K3*Q6*Q9*lambda1 - 16*K1*K3*L^2*Q1*Q4 +
16*K2*K3*L^2*Q1*Q4 + 12*K1*K3*L^2*Q2*Q5 + 8*K1*K2*L^2*Q4*Q5 -
4*K1*K3*L^2*Q2*Q6 - 4*K1*K3*L^2*Q3*Q5 - 4*K2*K3*L^2*Q2*Q5 +
8*K1*K2*L^2*Q4*Q6 - 4*K1*K3*L^2*Q3*Q6 - 4*K2*K3*L^2*Q2*Q6 -
4*K2*K3*L^2*Q3*Q5 + 12*K2*K3*L^2*Q3*Q6 + 16*K1*K3*Q1*Q4*lambda1^2 -
16*K2*K3*Q1*Q4*lambda1^2 - 12*K1*K3*Q2*Q5*lambda1^2 -
8*K1*K2*Q4*Q5*lambda1^2 + 4*K1*K3*Q2*Q6*lambda1^2 +
4*K1*K3*Q3*Q5*lambda1^2 + 4*K2*K3*Q2*Q5*lambda1^2 -
8*K1*K2*Q4*Q6*lambda1^2 + 4*K1*K3*Q3*Q6*lambda1^2 +
4*K2*K3*Q2*Q6*lambda1^2 + 4*K2*K3*Q3*Q5*lambda1^2 -
12*K2*K3*Q3*Q6*lambda1^2;
C4 = 8*K1^2*L^2*Q4^2 - 6*K1^2*L^2*Q5^2 + 2*K1^2*L^2*Q6^2 -
8*K1^2*Q4^2*lambda1^2 + 6*K1^2*Q5^2*lambda1^2 - 2*K1^2*Q6^2*lambda1^2
- 8*K1^2*Q7^2 + 6*K1^2*Q8^2 - 2*K1^2*Q9^2 + 8*K1*K2*L^2*Q5*Q6 -
8*K1*K2*Q5*Q6*lambda1^2 - 8*K1*K2*Q8*Q9 - 16*K1*K3*L^2*Q1*Q5 -
16*K1*K3*L^2*Q2*Q4 + 16*K1*K3*Q1*Q5*lambda1^2 +
16*K1*K3*Q2*Q4*lambda1^2 + 16*K1*K3*Q4*Q8*lambda1 +
16*K1*K3*Q5*Q7*lambda1 + 16*K1*K4*Q1*Q7 - 12*K1*K4*Q2*Q8 +
4*K1*K4*Q3*Q9 - 16*K1*K4*Q4^2*lambda1 + 12*K1*K4*Q5^2*lambda1 -
4*K1*K4*Q6^2*lambda1 + 4*K1*K5*Q2*Q9 + 4*K1*K5*Q3*Q8 -
8*K1*K5*Q5*Q6*lambda1 - 8*K2^2*L^2*Q4^2 + 2*K2^2*L^2*Q5^2 -
6*K2^2*L^2*Q6^2 + 8*K2^2*Q4^2*lambda1^2 - 2*K2^2*Q5^2*lambda1^2 +
6*K2^2*Q6^2*lambda1^2 + 8*K2^2*Q7^2 - 2*K2^2*Q8^2 + 6*K2^2*Q9^2 +
16*K2*K3*L^2*Q1*Q6 + 16*K2*K3*L^2*Q3*Q4 - 16*K2*K3*Q1*Q6*lambda1^2 -
16*K2*K3*Q3*Q4*lambda1^2 - 16*K2*K3*Q4*Q9*lambda1 -
16*K2*K3*Q6*Q7*lambda1 + 4*K2*K4*Q2*Q9 + 4*K2*K4*Q3*Q8 -
8*K2*K4*Q5*Q6*lambda1 - 16*K2*K5*Q1*Q7 + 4*K2*K5*Q2*Q8 -
12*K2*K5*Q3*Q9 + 16*K2*K5*Q4^2*lambda1 - 4*K2*K5*Q5^2*lambda1 +
12*K2*K5*Q6^2*lambda1 - 16*K3^2*L^2*Q1^2 + 8*K3^2*L^2*Q2^2 -
8*K3^2*L^2*Q3^2 + 16*K3^2*Q1^2*lambda1^2 + 32*K3^2*Q1*Q7*lambda1 -
8*K3^2*Q2^2*lambda1^2 - 16*K3^2*Q2*Q8*lambda1 + 8*K3^2*Q3^2*lambda1^2
+ 16*K3^2*Q3*Q9*lambda1 + 16*K3^2*Q7^2 - 8*K3^2*Q8^2 + 8*K3^2*Q9^2 +
16*K3*K4*Q1*Q5*lambda1 + 16*K3*K4*Q2*Q4*lambda1 + 16*K3*K4*Q4*Q8 +
16*K3*K4*Q5*Q7 - 16*K3*K5*Q1*Q6*lambda1 - 16*K3*K5*Q3*Q4*lambda1 -
16*K3*K5*Q4*Q9 - 16*K3*K5*Q6*Q7 - 8*K4^2*Q1^2 + 6*K4^2*Q2^2 -
2*K4^2*Q3^2 - 8*K4^2*Q4^2 + 6*K4^2*Q5^2 - 2*K4^2*Q6^2 - 8*K4*K5*Q2*Q3
- 8*K4*K5*Q5*Q6 + 8*K5^2*Q1^2 - 2*K5^2*Q2^2 + 6*K5^2*Q3^2 +
8*K5^2*Q4^2 - 2*K5^2*Q5^2 + 6*K5^2*Q6^2;
C3 = 4*K5^2*Q1*Q2 - 4*K4^2*Q1*Q3 - 12*K4^2*Q1*Q2 + 12*K5^2*Q1*Q3 -
12*K4^2*Q4*Q5 - 4*K4^2*Q4*Q6 + 4*K5^2*Q4*Q5 + 12*K5^2*Q4*Q6 -
12*K1^2*Q7*Q8 - 4*K1^2*Q7*Q9 + 4*K2^2*Q7*Q8 + 12*K2^2*Q7*Q9 +
16*K3^2*Q7*Q8 + 16*K3^2*Q7*Q9 + 16*K3^2*Q1*Q8*lambda1 +
16*K3^2*Q2*Q7*lambda1 + 16*K3^2*Q1*Q9*lambda1 + 16*K3^2*Q3*Q7*lambda1
- 16*K3^2*L^2*Q1*Q2 - 16*K3^2*L^2*Q1*Q3 + 12*K1^2*L^2*Q4*Q5 +
4*K1^2*L^2*Q4*Q6 - 4*K2^2*L^2*Q4*Q5 - 12*K2^2*L^2*Q4*Q6 +
16*K3^2*Q1*Q2*lambda1^2 + 16*K3^2*Q1*Q3*lambda1^2 -
12*K1^2*Q4*Q5*lambda1^2 - 4*K1^2*Q4*Q6*lambda1^2 +
4*K2^2*Q4*Q5*lambda1^2 + 12*K2^2*Q4*Q6*lambda1^2 - 8*K4*K5*Q1*Q2 +
8*K4*K5*Q1*Q3 + 12*K1*K4*Q1*Q8 + 12*K1*K4*Q2*Q7 + 4*K1*K4*Q1*Q9 +
4*K1*K4*Q3*Q7 + 4*K1*K5*Q1*Q8 + 4*K1*K5*Q2*Q7 + 4*K2*K4*Q1*Q8 +
```

```
4*K2*K4*Q2*Q7 - 4*K1*K5*Q1*Q9 - 4*K1*K5*Q3*Q7 - 4*K2*K4*Q1*Q9 -
4*K2*K4*Q3*Q7 - 4*K2*K5*Q1*Q8 - 4*K2*K5*Q2*Q7 - 12*K2*K5*Q1*Q9 -
12*K2*K5*Q3*Q7 - 8*K1*K2*Q7*Q8 - 16*K3*K4*Q4*Q7 - 8*K4*K5*Q4*Q5 +
8*K1*K2*Q7*Q9 - 16*K3*K5*Q4*Q7 + 8*K4*K5*Q4*Q6 + 12*K3*K4*Q5*Q8 +
4*K3*K4*Q5*Q9 + 4*K3*K4*Q6*Q8 + 4*K3*K5*Q5*Q8 - 4*K3*K4*Q6*Q9 -
4*K3*K5*Q5*Q9 - 4*K3*K5*Q6*Q8 - 12*K3*K5*Q6*Q9 -
16*K3*K4*Q1*Q4*lambda1 - 16*K3*K5*Q1*Q4*lambda1 -
24*K1*K4*Q4*Q5*lambda1 + 12*K3*K4*Q2*Q5*lambda1 -
16*K1*K3*Q4*Q7*lambda1 - 8*K1*K4*Q4*Q6*lambda1 - 8*K1*K5*Q4*Q5*lambda1
- 8*K2*K4*Q4*Q5*lambda1 + 4*K3*K4*Q2*Q6*lambda1 +
4*K3*K4*Q3*Q5*lambda1 + 4*K3*K5*Q2*Q5*lambda1 + 8*K1*K5*Q4*Q6*lambda1
- 16*K2*K3*Q4*Q7*lambda1 + 8*K2*K4*Q4*Q6*lambda1 +
8*K2*K5*Q4*Q5*lambda1 - 4*K3*K4*Q3*Q6*lambda1 - 4*K3*K5*Q2*Q6*lambda1
- 4*K3*K5*Q3*Q5*lambda1 + 12*K1*K3*Q5*Q8*lambda1 +
24*K2*K5*Q4*Q6*lambda1 - 12*K3*K5*Q3*Q6*lambda1 +
4*K1*K3*Q5*Q9*lambda1 + 4*K1*K3*Q6*Q8*lambda1 + 4*K2*K3*Q5*Q8*lambda1
- 4*K1*K3*Q6*Q9*lambda1 - 4*K2*K3*Q5*Q9*lambda1 -
4*K2*K3*Q6*Q8*lambda1 - 12*K2*K3*Q6*Q9*lambda1 + 16*K1*K3*L^2*Q1*Q4 +
16*K2*K3*L^2*Q1*Q4 - 12*K1*K3*L^2*Q2*Q5 + 8*K1*K2*L^2*Q4*Q5 -
4*K1*K3*L^2*Q2*Q6 - 4*K1*K3*L^2*Q3*Q5 - 4*K2*K3*L^2*Q2*Q5 -
8*K1*K2*L^2*Q4*Q6 + 4*K1*K3*L^2*Q3*Q6 + 4*K2*K3*L^2*Q2*Q6 +
4*K2*K3*L^2*Q3*Q5 + 12*K2*K3*L^2*Q3*Q6 - 16*K1*K3*Q1*Q4*lambda1^2 -
16*K2*K3*Q1*Q4*lambda1^2 + 12*K1*K3*Q2*Q5*lambda1^2 -
8*K1*K2*Q4*Q5*lambda1^2 + 4*K1*K3*Q2*Q6*lambda1^2 +
4*K1*K3*Q3*Q5*lambda1^2 + 4*K2*K3*Q2*Q5*lambda1^2 +
8*K1*K2*Q4*Q6*lambda1^2 - 4*K1*K3*Q3*Q6*lambda1^2 -
4*K2*K3*Q2*Q6*lambda1^2 - 4*K2*K3*Q3*Q5*lambda1^2 -
12*K2*K3*Q3*Q6*lambda1^2;
C2 = - 4*K1^2*L^2*Q4^2 + 4*K1^2*L^2*Q5^2 + 4*K1^2*L^2*Q5*Q6 +
4*K1^2*Q4^2*lambda1^2 - 4*K1^2*Q5^2*lambda1^2 - 4*K1^2*Q5*Q6*lambda1^2
+ 4*K1^2*Q7^2 - 4*K1^2*Q8^2 - 4*K1^2*Q8*Q9 - 8*K1*K2*L^2*Q4^2 +
4*K1*K2*L^2*Q5^2 - 4*K1*K2*L^2*Q6^2 + 8*K1*K2*Q4^2*lambda1^2 -
4*K1*K2*Q5^2*lambda1^2 + 4*K1*K2*Q6^2*lambda1^2 + 8*K1*K2*Q7^2 -
4*K1*K2*Q8^2 + 4*K1*K2*Q9^2 + 8*K1*K3*L^2*Q1*Q5 + 8*K1*K3*L^2*Q1*Q6 +
8*K1*K3*L^2*Q2*Q4 + 8*K1*K3*L^2*Q3*Q4 - 8*K1*K3*Q1*Q5*lambda1^2 -
8*K1*K3*Q1*Q6*lambda1^2 - 8*K1*K3*Q2*Q4*lambda1^2 -
8*K1*K3*Q3*Q4*lambda1^2 - 8*K1*K3*Q4*Q8*lambda1 -
8*K1*K3*Q4*Q9*lambda1 - 8*K1*K3*Q5*Q7*lambda1 - 8*K1*K3*Q6*Q7*lambda1
- 8*K1*K4*Q1*Q7 + 8*K1*K4*Q2*Q8 + 4*K1*K4*Q2*Q9 + 4*K1*K4*Q3*Q8 +
8*K1*K4*Q4^2*lambda1 - 8*K1*K4*Q5^2*lambda1 - 8*K1*K4*Q5*Q6*lambda1 -
8*K1*K5*Q1*Q7 + 4*K1*K5*Q2*Q8 - 4*K1*K5*Q3*Q9 + 8*K1*K5*Q4^2*lambda1 -
4*K1*K5*Q5^2*lambda1 + 4*K1*K5*Q6^2*lambda1 - 4*K2^2*L^2*Q4^2 -
4*K2^2*L^2*Q5*Q6 - 4*K2^2*L^2*Q6^2 + 4*K2^2*Q4^2*lambda1^2 +
4*K2^2*Q5*Q6*lambda1^2 + 4*K2^2*Q6^2*lambda1^2 + 4*K2^2*Q7^2 +
4*K2^2*Q8*Q9 + 4*K2^2*Q9^2 + 8*K2*K3*L^2*Q1*Q5 + 8*K2*K3*L^2*Q1*Q6 +
8*K2*K3*L^2*Q2*Q4 + 8*K2*K3*L^2*Q3*Q4 - 8*K2*K3*Q1*Q5*lambda1^2 -
8*K2*K3*Q1*Q6*lambda1^2 - 8*K2*K3*Q2*Q4*lambda1^2 -
8*K2*K3*Q3*Q4*lambda1^2 - 8*K2*K3*Q4*Q8*lambda1 -
8*K2*K3*Q4*Q9*lambda1 - 8*K2*K3*Q5*Q7*lambda1 - 8*K2*K3*Q6*Q7*lambda1
- 8*K2*K4*Q1*Q7 + 4*K2*K4*Q2*Q8 - 4*K2*K4*Q3*Q9 + 8*K2*K4*Q4^2*lambda1
- 4*K2*K4*Q5^2*lambda1 + 4*K2*K4*Q6^2*lambda1 - 8*K2*K5*Q1*Q7 -
4*K2*K5*Q2*Q9 - 4*K2*K5*Q3*Q8 - 8*K2*K5*Q3*Q9 + 8*K2*K5*Q4^2*lambda1 +
8*K2*K5*Q5*Q6*lambda1 + 8*K2*K5*Q6^2*lambda1 - 4*K3^2*L^2*Q2^2 -
8*K3^2*L^2*Q2*Q3 - 4*K3^2*L^2*Q3^2 + 4*K3^2*Q2^2*lambda1^2 +
8*K3^2*Q2*Q3*lambda1^2 + 8*K3^2*Q2*Q8*lambda1 + 8*K3^2*Q2*Q9*lambda1 +
4*K3^2*Q3^2*lambda1^2 + 8*K3^2*Q3*Q8*lambda1 + 8*K3^2*Q3*Q9*lambda1 +
4*K3^2*Q8^2 + 8*K3^2*Q8*Q9 + 4*K3^2*Q9^2 - 8*K3*K4*Q1*Q5*lambda1 -
8*K3*K4*Q1*Q6*lambda1 - 8*K3*K4*Q2*Q4*lambda1 - 8*K3*K4*Q3*Q4*lambda1
- 8*K3*K4*Q4*Q8 - 8*K3*K4*Q4*Q9 - 8*K3*K4*Q5*Q7 - 8*K3*K4*Q6*Q7 -
8*K3*K5*Q1*Q5*lambda1 - 8*K3*K5*Q1*Q6*lambda1 - 8*K3*K5*Q2*Q4*lambda1
- 8*K3*K5*Q3*Q4*lambda1 - 8*K3*K5*Q4*Q8 - 8*K3*K5*Q4*Q9 -
```

```
8*K3*K5*Q5*Q7 - 8*K3*K5*Q6*Q7 + 4*K4^2*Q1^2 - 4*K4^2*Q2^2 -
4*K4^2*Q2*Q3 + 4*K4^2*Q4^2 - 4*K4^2*Q5^2 - 4*K4^2*Q5*Q6 + 8*K4*K5*Q1^2
- 4*K4*K5*Q2^2 + 4*K4*K5*Q3^2 + 8*K4*K5*Q4^2 - 4*K4*K5*Q5^2 +
4*K4*K5*Q6^2 + 4*K5^2*Q1^2 + 4*K5^2*Q2*Q3 + 4*K5^2*Q3^2 + 4*K5^2*Q4^2
+ 4*K5^2*Q5*Q6 + 4*K5^2*Q6^2;
C1 = 4*K4^2*Q1*Q2 + 4*K4^2*Q1*Q3 + 4*K5^2*Q1*Q2 + 4*K5^2*Q1*Q3 +
4*K4^2*Q4*Q5 + 4*K4^2*Q4*Q6 + 4*K5^2*Q4*Q5 + 4*K5^2*Q4*Q6 +
4*K1^2*Q7*Q8 + 4*K1^2*Q7*Q9 + 4*K2^2*Q7*Q8 + 4*K2^2*Q7*Q9 -
4*K1^2*L^2*Q4*Q5 - 4*K1^2*L^2*Q4*Q6 - 4*K2^2*L^2*Q4*Q5 -
4*K2^2*L^2*Q4*Q6 + 4*K1^2*Q4*Q5*lambda1^2 + 4*K1^2*Q4*Q6*lambda1^2 +
4*K2^2*Q4*Q5*lambda1^2 + 4*K2^2*Q4*Q6*lambda1^2 + 8*K4*K5*Q1*Q2 +
8*K4*K5*Q1*Q3 - 4*K1*K4*Q1*Q8 - 4*K1*K4*Q2*Q7 - 4*K1*K4*Q1*Q9 -
4*K1*K4*Q3*Q7 - 4*K1*K5*Q1*Q8 - 4*K1*K5*Q2*Q7 - 4*K2*K4*Q1*Q8 -
4*K2*K4*Q2*Q7 - 4*K1*K5*Q1*Q9 - 4*K1*K5*Q3*Q7 - 4*K2*K4*Q1*Q9 -
4*K2*K4*Q3*Q7 - 4*K2*K5*Q1*Q8 - 4*K2*K5*Q2*Q7 - 4*K2*K5*Q1*Q9 -
4*K2*K5*Q3*Q7 + 8*K1*K2*Q7*Q8 + 8*K4*K5*Q4*Q5 + 8*K1*K2*Q7*Q9 +
8*K4*K5*Q4*Q6 - 4*K3*K4*Q5*Q8 - 4*K3*K4*Q5*Q9 - 4*K3*K4*Q6*Q8 -
4*K3*K5*Q5*Q8 - 4*K3*K4*Q6*Q9 - 4*K3*K5*Q5*Q9 - 4*K3*K5*Q6*Q8 -
4*K3*K5*Q6*Q9 + 8*K1*K4*Q4*Q5*lambda1 - 4*K3*K4*Q2*Q5*lambda1 +
8*K1*K4*Q4*Q6*lambda1 + 8*K1*K5*Q4*Q5*lambda1 + 8*K2*K4*Q4*Q5*lambda1
- 4*K3*K4*Q2*Q6*lambda1 - 4*K3*K4*Q3*Q5*lambda1 -
4*K3*K5*Q2*Q5*lambda1 + 8*K1*K5*Q4*Q6*lambda1 + 8*K2*K4*Q4*Q6*lambda1
+ 8*K2*K5*Q4*Q5*lambda1 - 4*K3*K4*Q3*Q6*lambda1 -
4*K3*K5*Q2*Q6*lambda1 - 4*K3*K5*Q3*Q5*lambda1 - 4*K1*K3*Q5*Q8*lambda1
+ 8*K2*K5*Q4*Q6*lambda1 - 4*K3*K5*Q3*Q6*lambda1 -
4*K1*K3*Q5*Q9*lambda1 - 4*K1*K3*Q6*Q8*lambda1 - 4*K2*K3*Q5*Q8*lambda1
- 4*K1*K3*Q6*Q9*lambda1 - 4*K2*K3*Q5*Q9*lambda1 -
4*K2*K3*Q6*Q8*lambda1 - 4*K2*K3*Q6*Q9*lambda1 + 4*K1*K3*L^2*Q2*Q5 -
8*K1*K2*L^2*Q4*Q5 + 4*K1*K3*L^2*Q2*Q6 + 4*K1*K3*L^2*Q3*Q5 +
4*K2*K3*L^2*Q2*Q5 - 8*K1*K2*L^2*Q4*Q6 + 4*K1*K3*L^2*Q3*Q6 +
4*K2*K3*L^2*Q2*Q6 + 4*K2*K3*L^2*Q3*Q5 + 4*K2*K3*L^2*Q3*Q6 -
4*K1*K3*Q2*Q5*lambda1^2 + 8*K1*K2*Q4*Q5*lambda1^2 -
4*K1*K3*Q2*Q6*lambda1^2 - 4*K1*K3*Q3*Q5*lambda1^2 -
4*K2*K3*Q2*Q5*lambda1^2 + 8*K1*K2*Q4*Q6*lambda1^2 -
4*K1*K3*Q3*Q6*lambda1^2 - 4*K2*K3*Q2*Q6*lambda1^2 -
4*K2*K3*Q3*Q5*lambda1^2 - 4*K2*K3*Q3*Q6*lambda1^2;
C0 = - K1^2*L^2*Q5^2 - 2*K1^2*L^2*Q5*Q6 - K1^2*L^2*Q6^2 +
K1^2*Q5^2*lambda1^2 + 2*K1^2*Q5*Q6*lambda1^2 + K1^2*Q6^2*lambda1^2 +
K1^2*Q8^2 + 2*K1^2*Q8*Q9 + K1^2*Q9^2 - 2*K1*K2*L^2*Q5^2 -
4*K1*K2*L^2*Q5*Q6 - 2*K1*K2*L^2*Q6^2 + 2*K1*K2*Q5^2*lambda1^2 +
4*K1*K2*Q5*Q6*lambda1^2 + 2*K1*K2*Q6^2*lambda1^2 + 2*K1*K2*Q8^2 +
4*K1*K2*Q8*Q9 + 2*K1*K2*Q9^2 - 2*K1*K4*Q2*Q8 - 2*K1*K4*Q2*Q9 -
2*K1*K4*Q3*Q8 - 2*K1*K4*Q3*Q9 + 2*K1*K4*Q5^2*lambda1 +
4*K1*K4*Q5*Q6*lambda1 + 2*K1*K4*Q6^2*lambda1 - 2*K1*K5*Q2*Q8 -
2*K1*K5*Q2*Q9 - 2*K1*K5*Q3*Q8 - 2*K1*K5*Q3*Q9 + 2*K1*K5*Q5^2*lambda1 +
4*K1*K5*Q5*Q6*lambda1 + 2*K1*K5*Q6^2*lambda1 - K2^2*L^2*Q5^2 -
2*K2^2*L^2*Q5*Q6 - K2^2*L^2*Q6^2 + K2^2*Q5^2*lambda1^2 +
2*K2^2*Q5*Q6*lambda1^2 + K2^2*Q6^2*lambda1^2 + K2^2*Q8^2 +
2*K2^2*Q8*Q9 + K2^2*Q9^2 - 2*K2*K4*Q2*Q8 - 2*K2*K4*Q2*Q9 -
2*K2*K4*Q3*Q8 - 2*K2*K4*Q3*Q9 + 2*K2*K4*Q5^2*lambda1 +
4*K2*K4*Q5*Q6*lambda1 + 2*K2*K4*Q6^2*lambda1 - 2*K2*K5*Q2*Q8 -
2*K2*K5*Q2*Q9 - 2*K2*K5*Q3*Q8 - 2*K2*K5*Q3*Q9 + 2*K2*K5*Q5^2*lambda1 +
4*K2*K5*Q5*Q6*lambda1 + 2*K2*K5*Q6^2*lambda1 + K4^2*Q2^2 +
2*K4^2*Q2*Q3 + K4^2*Q3^2 + K4^2*Q5^2 + 2*K4^2*Q5*Q6 + K4^2*Q6^2 +
2*K4*K5*Q2^2 + 4*K4*K5*Q2*Q3 + 2*K4*K5*Q3^2 + 2*K4*K5*Q5^2 +
4*K4*K5*Q5*Q6 + 2*K4*K5*Q6^2 + K5^2*Q2^2 + 2*K5^2*Q2*Q3 + K5^2*Q3^2 +
K5^2*Q5^2 + 2*K5^2*Q5*Q6 + K5^2*Q6^2;

% Vector de coeficientes C
C = [C8 C7 C6 C5 C4 C3 C2 C1 C0];
```

```
%% ResoluciÃ³n ecuaciÃ³n

% Mostrar coef. del polinomio
% format long;
% disp('Coef. polinomio:');
% disp(C.');
% format

% evaluar el polinommio con la solución aportada para ver si se
cumple,
% mostrar el residuo
% res=polyval(C,T);
% disp('Residuo:');
% disp(res);

% Raíces del polinomio
t = roots(C);

% Mostrar las raíces
% disp('Raíces del polinomio: ');
% disp(t);

% Tomar solo las raÃces reales
treal = [];
for i = 1:length(t)
    % una tolerancia para la parte imaginaria DEFINIR QUÉ VALOR Y
CUANTOS
    % DECIMALES USAR
    if abs(imag(t(i))) < 1e-4
        % asegurarse de coger número real
        treal = [treal; real(t(i))];
    end
end

% Mostrar las raices reales seleccionadas
% disp('Raíces t reales del polinomio: ');
% disp(treal);

%% Cálculo de la posición

% Sin, cos y tan de psi en función de t
% Sabemos que t = tan(psi/2)
sinpsi = (2.*treal)./(1 + treal.^2);
cospsi = (1 - treal.^2)./(1 + treal.^2);
tanpsi = (2.*treal)./(1-treal.^2);

f1 = K1 .* cospsi + K2;
f2 = K3 .* sinpsi;
f3 = K4 .* cospsi + K5;

g1 = Q1 .* sinpsi + Q2 .* cospsi + Q3;
g2 = Q4 .* sinpsi + Q5 .* cospsi + Q6;
g3 = Q7 .* sinpsi + Q8 .* cospsi + Q9;

h1 = (f3 .* g2 - f2 .* g3)./(f2 .* g1 - f1 .* g2);
h2 = (f1 .* g3 - f3 .* g1)./(f2 .* g1 - f1 .* g2);
```

```
%% Posición singular

% Verificación denominador: si es cero exacto genera coordenadas NaN
% POSICION SINGULAR
den = f2 .* g1 - f1 .* g2;
disp('denominador:');
disp(den);

% El vector possing identifica las posiciones singulares asignándoles
1 y a
% las demás 0
possing = zeros(length(den),1);
for i = 1:length(den)
    if abs(den(i)) < L * 1e-3
        possing(i) = 1;
    end
end


%% Ángulo psi
psi = atan(tanpsi);

% Para cada valor de psi, si el coseno es negativo, psi = psi + pi.
Esto es
% porque la arcotangente tiene dos soluciones, pero la función atan da
sólo
% aquella que está en el intervalo [-pi/2 , pi/2].

for i = 1:length(psi)
    if cospsi(i) < 0
        psi(i) = psi(i) + pi;
    end
end

% Ángulo positivo
for i = 1:length(psi)
    if psi(i) < 0
        psi(i) = psi(i) + 2 * pi;
    end
end

% Ángulo en [0,2*pi]
for i = 1:length(psi)
    if psi(i) > 2 * pi
        psi(i) = psi(i) - 2 * pi;
    end
end

% en grados para poder validar más fácil
psigrad = psi*180/pi;


%% Coordenadas xp, yp

% xp = h1
% yp = h2

xp = h1;
yp = h2;
```

```
%% Mostrar resultados

disp(' ');
disp(' ');
for i = 1:length(psi)
    disp(['SOLUCIÓN ' num2str(i)]);

    if possing(i) == 1    % Si es posición singular
        disp('POSICIÓN SINGULAR');
        % Ángulo psi
        disp(['Ángulo en grados: ' num2str(psigrad(i))]);

    else                 % Si NO es posición singular
        % Ángulo psi
        disp(['Ángulo en grados: ' num2str(psigrad(i))]);
        % Coordenada xp
        disp(['Coordenada xp: ' num2str(xp(i))]);
        % Coordenada yp
        disp(['Coordenada yp: ' num2str(yp(i))]);
    end
    disp(' ');
end
```

2.3. FUNCIONES ASOCIADAS

```
function [lambda] =
Calcular_Problema_Inverso_3PRR(L,a,B2,B3,H3,xp,yp,psi)
%Con los parámetros geométricos y la posición del elemento triangular,
%calcula todas las posibles soluciones del problema inverso

%Var. entrada: L: longitud barras (m)
%               a: lado triángulo (m)
%               B2: posición x motor 2 (m)
%               B3: posición x motor 3 (m)
%               H3: posición y motor 3 (m)
%               xp,yp,psi: posición y orientación elemento triangular

%Var. salida: lambda: matriz de soluciones



%% Vértices P2 y P3
xp2 = xp+a*cos(psi);
yp2 = yp+a*sin(psi);
xp3 = xp+(a/2)*(cos(psi)-sqrt(3)*sin(psi));
yp3 = yp+(a/2)*(sin(psi)+sqrt(3)*cos(psi));


%% Valores posibles de lambda1, 2 y 3
lambda1 = [xp+sqrt(L^2-yp^2)            xp-sqrt(L^2-yp^2)];
lambda2 = [B2-xp2-sqrt(L^2-yp2^2)       B2-xp2+sqrt(L^2-yp2^2)];
lambda3 = [H3-yp3-sqrt(L^2-(B3-xp3)^2)  H3-yp3+sqrt(L^2-(B3-xp3)^2)];


%% Comprobar tangencia/solución doble(tolerancia de L * 1e-6 m)
POSICIÓN SINGULAR

possing = [0 0 0];

if abs(lambda1(1)-lambda1(2)) <= L * 1e-6
    lambda1(2) = lambda1(1);
    possing(1) = 1;
end
if abs(lambda2(1)-lambda2(2)) <= L * 1e-6
    lambda2(2) = lambda2(1);
    possing(2) = 1;
end
if abs(lambda3(1)-lambda3(2)) <= L * 1e-6
    lambda3(2) = lambda3(1);
    possing(3) = 1;
end


%% Matriz lambda
%Cada fila es una combinación de soluciones

if possing == [0 0 0]      % NO POSICION SINGULAR
lambda = [lambda1(1) lambda2(1) lambda3(1)
          lambda1(1) lambda2(1) lambda3(2)
          lambda1(1) lambda2(2) lambda3(1)
          lambda1(1) lambda2(2) lambda3(2)
          lambda1(2) lambda2(1) lambda3(1)
```

```
            lambda1(2) lambda2(1) lambda3(2)
            lambda1(2) lambda2(2) lambda3(1)
            lambda1(2) lambda2(2) lambda3(2)];
end

if possing == [1 0 0]     % POSICION SINGULAR
lambda = [lambda1(1) lambda2(1) lambda3(1)
            lambda1(1) lambda2(1) lambda3(2)
            lambda1(1) lambda2(2) lambda3(1)
            lambda1(1) lambda2(2) lambda3(2)];
end

if possing == [0 1 0]     % POSICION SINGULAR
lambda = [lambda1(1) lambda2(1) lambda3(1)
            lambda1(1) lambda2(1) lambda3(2)
            lambda1(2) lambda2(1) lambda3(1)
            lambda1(2) lambda2(1) lambda3(2)];
end


if possing == [0 0 1]     % POSICION SINGULAR
lambda = [lambda1(1) lambda2(1) lambda3(1)
            lambda1(1) lambda2(2) lambda3(1)
            lambda1(2) lambda2(1) lambda3(1)
            lambda1(2) lambda2(2) lambda3(1)];
end


if possing == [1 1 0]     % POSICION SINGULAR
lambda = [lambda1(1) lambda2(1) lambda3(1)
            lambda1(1) lambda2(1) lambda3(2)];
end

if possing == [1 0 1]     % POSICION SINGULAR
lambda = [lambda1(1) lambda2(1) lambda3(1)
            lambda1(1) lambda2(2) lambda3(1)];
end

if possing == [0 1 1]     % POSICION SINGULAR
lambda = [lambda1(1) lambda2(1) lambda3(1)
            lambda1(2) lambda2(1) lambda3(1)];
end

if possing == [1 1 1]     % POSICION SINGULAR
lambda = [lambda1(1) lambda2(1) lambda3(1)];
end

end
function [sol] = Comprobar_Problema_Inverso_3PRR(L,a,B3,xp,yp,psi)
%Comprueba que el problema inverso tiene solución.
%Var. entrada: L: longitud barras (m)
%              a: lado triángulo (m)
%              B3: posición x motor 3 (m)
%              xp,yp,psi: posición y orientación elemento triangular
%Var. salida: sol: parámetro que vale 1 si el problema iverso tiene
%                  solución y 0 si no la tiene

%% Vértices P2 y P3
yp2 = yp + a * sin(psi);
```

```
xp3 = xp + (a/2) * (cos(psi) - sqrt(3) * sin(psi));


%% ¿Solución real?
%Si los radicandos de las expresiones para calcular lambda1, 2 y 3 son
%mayores o iguales que 0, la solución del problema será real
if L^2-yp^2 >= 0 && L^2-yp2^2 >= 0 && L^2-(B3-xp3)^2 >= 0
    sol = 1;
else
    sol = 0;
end


end



function [  ] =
Plot_3PRR(a,B2,B3,H3,lambda1,lambda2,lambda3,xp,yp,psi)
%Representa el mecanismo

%Var. entrada: L: longitud barras (m)
%              a: lado triángulo (m)
%              B2: posición x motor 2 (m)
%              B3: posición x motor 3 (m)
%              H3: posición y motor 3 (m)
%              lambda1, lambda2, lambda3: posición de los motores
%              xp,yp,psi: posición y orientación elemento triangular




%Vértices P2 y P3
xp2 = xp + a * cos(psi);
yp2 = yp + a * sin(psi);
xp3 = xp + (a/2) * (cos(psi) - sqrt(3) * sin(psi));
yp3 = yp + (a/2) * (sin(psi) + sqrt(3) * cos(psi));

%Matriz transformación
R = [cos(psi) -sin(psi)
     sin(psi)  cos(psi)];
%Escala
k = 0.5 * a;

xc1 = lambda1;
yc1 = 0;
xc2 = B2 - lambda2;
yc2 = 0;
xc3 = B3;
yc3 = H3 - lambda3;

xd1 = 0;
yd1 = 0;
xd2 = B2;
yd2 = 0;
xd3 = B3;
yd3 = H3;


hold on;
```

```
plot([xp xp2 xp3 xp],[yp yp2 yp3 yp],'-r','LineWidth',1.5);
%Triángulo
plot([xd1 xc1 xp],[yd1 yc1 yp],'-k','LineWidth',1);  %Rama 1
plot([xc1 xp],[yc1 yp],'.k','MarkerSize',10);
plot([xd2 xc2 xp2],[yd2 yc2 yp2],'-k','LineWidth',1);  %Rama 2
plot([xc2 xp2],[yc2 yp2],'.k','MarkerSize',10);
plot([xd3 xc3 xp3],[yd3 yc3 yp3],'-k','LineWidth',1);  %Rama 3
plot([xc3 xp3],[yc3 yp3],'.k','MarkerSize',10);
quiver([xp xp],[yp yp],[k*R(1,1) k*R(1,2)],[k*R(2,1) k*R(2,2)],'g');
%Sist ref en el mecanismo
axis equal;
xlabel('X');
ylabel('Y');
hold off;


end
```

3. CÓDIGO DE LOS PROGRAMAS DEL MECANISMO 2RFR

3.1. INTEGRACIÓN NUMÉRICA. PROBLEMA INVERSO

```
% Initial design parameters of the mechanism
clear; close all;
load('FK_MultipleSolutions');

% Defining initial position
NSol = 2 ;

psi1 = psi1_sol(NSol);
psi2 = psi2_sol(NSol);
kr1  = kr1_sol(NSol);
kr2  = kr2_sol(NSol);

IN.B1.psi = psi1 ;
IN.B1.kr  = kr1 ;
IN.B2.psi = psi2 ;
IN.B2.kr  = kr2 ;


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% ROD PROPERTIES
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
diam = 1.5e-3 ;      % Diameter of the section [m]
rad = 0.5*diam ;
I = 0.25*pi*rad^4 ; % Intertial of the cross-section [m^4]

L  = 1 ;             % Length of the rod [m]

E = 210e9 ;          % Young's modulus [Pa]
EI = E*I;

N = 41 ;             % Number of discretitaiton points of the rod

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%
% Calculating n0, m0 and end effector position [x; y]

% Rod 1
% [ x1_loc, y1_loc, R1 ] = EmpArt_xyR_local( IN.B1 );
[ x_1, y_1, F1_x, F1_y ] = EmpArt_xyR( IN.B1 );
R1 = sqrt(F1_x^2 + F1_y^2);

% n0_1 = [-R1*cos(IN.B1.theta+IN.B1.psi); -
R1*sin(IN.B1.theta+IN.B1.psi)];

kmin1 = abs(cos(0.5*IN.B1.psi)) ;
k1    = sign(IN.B1.kr).*kmin1 + (1-kmin1).*IN.B1.kr ;
aux  = 1./k1.*cos(0.5*IN.B1.psi);
phi1 = asin(aux);
m0_1 = 2.*k1.*sqrt(IN.B1.EI.*R1).*cos(phi1);

% Rod 2
% [ x2_loc, y2_loc, R2 ] = EmpArt_xyR_local( IN.B2 );
[ x_2, y_2, F2_x, F2_y ] = EmpArt_xyR( IN.B2 );
R2 = sqrt(F2_x^2 + F2_y^2);
```

```
% n0_2 = [-R2*cos(IN.B2.theta+IN.B2.psi); -
R2*sin(IN.B2.theta+IN.B2.psi)];

kmin2 = abs(cos(0.5*IN.B2.psi)) ;
k2    = sign(IN.B2.kr).*kmin2 + (1-kmin2).*IN.B2.kr ;
aux   = 1./k2.*cos(0.5*IN.B2.psi);
phi1 = asin(aux);
m0_2 = 2.*k2.*sqrt(IN.B2.EI.*R2).*cos(phi1);

% End effector position
% R = [cos(IN.B1.theta) -sin(IN.B1.theta);
%      sin(IN.B1.theta)  cos(IN.B1.theta)] ;
% endpos = R*[x1_loc; y1_loc];
% x = endpos(1);
% y = endpos(2);


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%


%%%%%%%%%%%%%%%%%% GEOMETRIC DEFINITON OF THE MECHANISM
%%%%%%%%%%%%%%%%%%
% Angle of each actuator
% alpha1 = 15/180*pi ;           %[rad]    GV
% alpha2 = 15/180*pi ;           %[rad]    GV
alpha1 = IN.B1.theta;
alpha2 = IN.B2.theta;

% Angle of the rod at its clamped end
theta0_1 = alpha1 ;
theta0_2 = alpha2 ;

% Reference point through which each of the guides passes
% OA1 = [ -0.3; 0 ] ;      % [m]
% OA2 = [ 0.3; 0 ] ;        % [m]
OA1 = [IN.B1.x0; IN.B1.y0];
OA2 = [IN.B2.x0; IN.B2.y0];


% Reference point of the end effector
% OP = [0.5; 0.3] ;    % [m]
OP = [x_1; y_1];




%%%%%%%%%%%% CREATION OF THE STRUCTURE FOR THE SHOOTING METHOD
%%%%%%%%%%%%
% "GV" means here "guess value", a value that is not known a priori

% RODS.IN.B1.alpha  = alpha1 ;
RODS.IN.B1.L      = L ;
RODS.IN.B1.OA     = OA1 ;
RODS.IN.B1.N      = N ;
RODS.IN.B1.EI     = E*I ;
RODS.IN.B1.theta0 = theta0_1 ;
RODS.IN.B1.n0     = [F1_x; F1_y] ;     % GV
```

```
RODS.IN.B1.m0      = m0_1 ;      % GV
RODS.IN.B1.lambda = 0 ;

% RODS.IN.B2.alpha  = alpha2 ;
RODS.IN.B2.L       = L ;
RODS.IN.B2.OA      = OA2 ;
RODS.IN.B2.N       = N ;
RODS.IN.B2.EI      = E*I ;
RODS.IN.B2.theta0 = theta0_2 ;
RODS.IN.B2.n0      = [F2_x; F2_y] ;      % GV
RODS.IN.B2.m0      = m0_2 ;      % GV
RODS.IN.B2.lambda = 0 ;

RODS.IN.OP = OP ;


% Solving the Inverse Kinematic Problem in order to get the solution
for
% the initial position ;
RODS = IK_ShootingMethod( RODS, [0;0] ) ;

% % Update de reference point from which lambda is measured
% RODS.IN.B1.OA = RODS.IN.B1.OA + RODS.IN.B1.lambda*[
cos(RODS.IN.B1.alpha); sin(RODS.IN.B1.alpha) ] ;
% RODS.IN.B2.OA = RODS.IN.B2.OA + RODS.IN.B2.lambda*[
cos(RODS.IN.B2.alpha); sin(RODS.IN.B2.alpha) ] ;
% RODS.IN.B1.lambda = 0 ;
% RODS.IN.B2.lambda = 0 ;

RODS = IK_ShootingMethod( RODS, [0;0] ) ;


% Save de structure data
save('RODS', 'RODS') ;


% Plotting the mechanism
figure;
axis equal; grid; hold on;
GraficRepresentation( RODS ) ;

if RODS.OUT.sol == 0
    disp('Aborted calculation\n') ;
end

% Algorithm that solves de Inverse Kinematic position problem through
the
% the use of the Shootin Method
clear; close all;
addpath('EllipticIntegrals_functions');

% Creation (1) or not(0) of a .gif file
graf_gif = 0 ;


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% INPUT DATA
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
tol = 1e-4;
```

```
% Loading data of the mechanism in the initial position
load('RODS');

% Data for the Inverse Kinematic position problem
% Option 1: Reading data from xls file
% Option 2: Circular path on (x, y)
% Option 3: Rectangular path on (x, y)
% Option 4: Straigth path on (x, y)
opt = 2 ;

if opt == 1
    data = xlsread('IK_inputs.xlsx') ;

    incr_xp = data(:,1) ;       % Increment of x coordinate of the end
effector
                                % respect to value that xp takes in
the initial
                                % position
    incr_yp = data(:,2) ;       % Increment of y coordinate of the end
effector
                                % respect to value that yp takes in
the initial
                                % position
    Fext    = data(:,3:4) ;     % External forces. The first column is
the
                                % x component of the force and the
second
                                % colum, the y component
elseif opt == 2
    D = 0.2 ;      % Diameter of external circle
    Ncir = 5 ;     % Number of concentric circles
    Ndiv = 8 ;     % Number of divisions
    [incr_xp, incr_yp] = Workspace_Circles (D, Ncir, Ndiv);

    Fext = zeros (length(incr_xp),2) ;

elseif opt == 3
    b = 0.2 ;     % Base of external rectangle
    h = 0.2 ;     % Heigth of external rectangle
    N = 4 ;       % Number of rectangles
    Ndiv = 4 ;    % Number of divisions for each side
    [incr_xp, incr_yp] = Workspace_Rectangles (b, h, N, Ndiv) ;

    Fext = zeros (length(incr_xp),2) ;

elseif opt == 4
    L = 0.5 ;              % Path length
    phi = 180/180*pi ;      % Orientation angle [rad]
    Ndiv = 100 ;            % Number of divisions
    [incr_xp, incr_yp] = StraigthPath(L, phi, Ndiv) ;

    Fext = zeros (length(incr_xp),2) ;

end

% Inital position
OP_ref    = RODS.IN.OP ;
```

```
% Number of positions to be calculated
Nii = length(incr_xp) ;

% Creating vectors of points where solution is reached
x_sol = [] ;
y_sol = [] ;


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% OUTPUT DATA
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
RESULTS.THETA1  = zeros(Nii, 1) ;
RESULTS.THETA2  = zeros(Nii, 1) ;
RESULTS.MOMENT1 = zeros(Nii, 1) ;
RESULTS.MOMENT2 = zeros(Nii, 1) ;

RESULTS.FK_J11 = zeros(Nii, 1) ;
RESULTS.FK_J12 = zeros(Nii, 1) ;
RESULTS.FK_J21 = zeros(Nii, 1) ;
RESULTS.FK_J22 = zeros(Nii, 1) ;
RESULTS.detJFK = zeros(Nii, 1) ;

RESULTS.IK_J11 = zeros(Nii, 1) ;
RESULTS.IK_J12 = zeros(Nii, 1) ;
RESULTS.IK_J21 = zeros(Nii, 1) ;
RESULTS.IK_J22 = zeros(Nii, 1) ;
RESULTS.detJIK = zeros(Nii, 1) ;

RESULTS.K_11 = zeros(Nii, 1) ;
RESULTS.K_12 = zeros(Nii, 1) ;
RESULTS.K_21 = zeros(Nii, 1) ;
RESULTS.K_22 = zeros(Nii, 1) ;
RESULTS.detK = zeros(Nii, 1) ;


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% CALCULATION
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
h = figure(1);
if graf_gif == 1
    axis tight manual
    filename = 'IK_testAnimated.gif';
end


for ii = 1:Nii

    % Saving last RODS structure in case the shooting method does not
    % converge
    RODS0 = RODS;

    % Update the new end effector position
    RODS.IN.OP     = OP_ref     + [incr_xp(ii); incr_yp(ii)] ;

    % Shooting method
    RODS = IK_ShootingMethod( RODS, Fext(ii,:)' ) ;

    % Forward Kinematic Jacobian matrix
    FK_Jacob = FK_Jacobian( RODS, Fext(ii,:)' ) ;
```

```
    % Inverse Kinematic Jacobian matrix
    IK_Jacob = IK_Jacobian( RODS, Fext(ii,:)' ) ;

    % Stiffness matrix
    K = StiffnessMatrix_SM( RODS, Fext(ii,:)' ) ;

    if RODS.OUT.sol == 0
        %         fprintf('Aborted calculation\n') ;
        %         break ;

        % Solution is not valid. Invalid data are saved as NaN
        RESULTS.THETA1(ii)  = NaN ;
        RESULTS.THETA2(ii)  = NaN ;
        RESULTS.MOMENT1(ii) = NaN ;
        RESULTS.MOMENT2(ii) = NaN ;

        RESULTS.FK_J11(ii) = NaN ;
        RESULTS.FK_J12(ii) = NaN ;
        RESULTS.FK_J21(ii) = NaN ;
        RESULTS.FK_J22(ii) = NaN ;
        RESULTS.detJFK(ii) = NaN ;

        RESULTS.IK_J11(ii) = NaN ;
        RESULTS.IK_J12(ii) = NaN ;
        RESULTS.IK_J21(ii) = NaN ;
        RESULTS.IK_J22(ii) = NaN ;
        RESULTS.detJIK(ii) = NaN ;

        RESULTS.K_11(ii) = NaN ;
        RESULTS.K_12(ii) = NaN ;
        RESULTS.K_21(ii) = NaN ;
        RESULTS.K_22(ii) = NaN ;
        RESULTS.detK(ii) = NaN ;


        % If shooting method does not converge RODS is not valid so
the
        % previous RODS structure has to be used in next iteration
        RODS = RODS0;

    else
        % Position coordinates
        x_sol = [x_sol OP_ref(1)+incr_xp(ii) ] ;
        y_sol = [y_sol OP_ref(2)+incr_yp(ii) ] ;

        % Storing results
        RESULTS.THETA1(ii)  = RODS.IN.B1.theta0 ;
        RESULTS.THETA2(ii)  = RODS.IN.B2.theta0 ;
        RESULTS.MOMENT1(ii) = RODS.OUT.B1.Moment ;
        RESULTS.MOMENT2(ii) = RODS.OUT.B2.Moment ;

        RESULTS.FK_J11(ii) = FK_Jacob(1,1) ;
        RESULTS.FK_J12(ii) = FK_Jacob(1,2) ;
        RESULTS.FK_J21(ii) = FK_Jacob(2,1) ;
        RESULTS.FK_J22(ii) = FK_Jacob(2,2) ;
        RESULTS.detJFK(ii) = det(FK_Jacob) ;

        RESULTS.IK_J11(ii) = IK_Jacob(1,1) ;
        RESULTS.IK_J12(ii) = IK_Jacob(1,2) ;
```

```
        RESULTS.IK_J21(ii) = IK_Jacob(2,1) ;
        RESULTS.IK_J22(ii) = IK_Jacob(2,2) ;
        RESULTS.detJIK(ii) = det(IK_Jacob) ;

        RESULTS.K_11(ii) = K(1,1) ;
        RESULTS.K_12(ii) = K(1,2) ;
        RESULTS.K_21(ii) = K(2,1) ;
        RESULTS.K_22(ii) = K(2,2) ;
        RESULTS.detK(ii) = det(K) ;

        % Drawing the mechanism
        clf;
        axis equal; grid; hold on ;
        GraficRepresentation( RODS ) ;
        plot(OP_ref(1)+incr_xp(1:ii), OP_ref(2)+incr_yp(1:ii), '-m') ;
        plot(x_sol, y_sol, '.m') ;
        pause(1e-3) ;


        % Creation of a .gif file
        if graf_gif == 1

            drawnow
            % Capture the plot as an image
            frame = getframe(h);
            im = frame2im(frame);
            [imind,cm] = rgb2ind(im,256);

            % Write to the GIF File
            if ii == 1
                imwrite(imind,cm,filename,'gif', 'Loopcount',inf);
            else
                imwrite(imind,cm,filename,'gif','WriteMode','append');
            end

        end


        clc;
        fprintf('%f \n%%',(ii/Nii*100));

    end

end


if opt == 1

    matrix = [RESULTS.THETA1 RESULTS.THETA2 ] ;
    xlRange = ['F2:G' num2str(Nii+1)] ;
    xlswrite('IK_inputs.xlsx', matrix, xlRange) ;

    matrix = [ RESULTS.MOMENT1 RESULTS.MOMENT2 ] ;
    xlRange = ['I2:J' num2str(Nii+1)] ;
    xlswrite('IK_inputs.xlsx', matrix, xlRange) ;

    matrix = [ RESULTS.FK_J11 RESULTS.FK_J12
               RESULTS.FK_J21 RESULTS.FK_J22 ] ;
```

```
    xlRange = ['L2:O' num2str(Nii+1)] ;
    xlswrite('IK_inputs.xlsx', matrix, xlRange) ;

    xlRange = ['P2:P' num2str(Nii+1)] ;
    xlswrite('IK_inputs.xlsx', RESULTS.detJFK, xlRange) ;

    matrix = [ RESULTS.IK_J11 RESULTS.IK_J12 ...
               RESULTS.IK_J21 RESULTS.IK_J22 ] ;
    xlRange = ['R2:U' num2str(Nii+1)] ;
    xlswrite('IK_inputs.xlsx', matrix, xlRange) ;

    xlRange = ['V2:V' num2str(Nii+1)] ;
    xlswrite('IK_inputs.xlsx', RESULTS.detJIK, xlRange) ;

    matrix = [ RESULTS.K_11 RESULTS.K_12 ...
               RESULTS.K_21 RESULTS.K_22 ] ;
    xlRange = ['X2:AA' num2str(Nii+1)] ;
    xlswrite('IK_inputs.xlsx', matrix, xlRange) ;

    xlRange = ['AB2:AB' num2str(Nii+1)] ;
    xlswrite('IK_inputs.xlsx', RESULTS.detK, xlRange) ;

end


%%%%%%%%%%%%%%%%%%%%%%%%%%% PLOTTING RESULTS
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


Coord_x = OP_ref(1) + incr_xp;
Coord_y = OP_ref(2) + incr_yp;

[X, Y] = meshgrid(Coord_x, Coord_y);



% Theta1, Theta2 values as functions of (x,y)
Theta_1 = RESULTS.THETA1;
Theta_2 = RESULTS.THETA2;


fig1 = figure;
figure(fig1);

d1 = plot3(Coord_x, Coord_y, Theta_1, '.r', 'MarkerSize', 7);
hold on; grid;

d2 = plot3(Coord_x, Coord_y, Theta_2, '.b', 'MarkerSize',7);

if opt ~= 4
    THETA1 = griddata(Coord_x, Coord_y, Theta_1, X, Y) ;
    THETA2 = griddata(Coord_x, Coord_y, Theta_2, X, Y) ;

    mesh(X,Y,THETA1, 'FaceColor', 'r', 'EdgeColor', 'r', 'FaceAlpha',
0.2, 'EdgeAlpha', 0.2) ;
    mesh(X,Y,THETA2, 'FaceColor', 'b', 'EdgeColor', 'b', 'FaceAlpha',
0.2, 'EdgeAlpha', 0.2) ;
```

```
end

legend([d1, d2], '\theta_1', '\theta_2');
xlabel('X Coordinate');
ylabel('Y Coordinate');
zlabel(['{\theta}_' num2str(1) ', {\theta}_' num2str(2)]);


% |J_IK| as a function of (x,y)
det_JIK = RESULTS.detJIK ;

fig2 = figure;
figure(fig2);

plot3(Coord_x, Coord_y, det_JIK, '.r', 'MarkerSize', 7);
hold on; grid;

if opt ~= 4
    DJIK = griddata(Coord_x, Coord_y, det_JIK, X, Y) ;

    mesh(X,Y,DJIK, 'FaceColor', 'r', 'EdgeColor', 'r', 'FaceAlpha',
0.3, 'EdgeAlpha', 0.3) ;
    contour3(X,Y,DJIK, 'LevelList', 0, 'LineColor', 'black',
'LineWidth', 2);
end

xlabel('X Coordinate');
ylabel('Y Coordinate');
zlabel('|J_I_K|');

% Inverse problem singularities are marked
for jj = 1:length(det_JIK)
    if abs(det_JIK(jj)) <= tol
        plot3(Coord_x(jj), Coord_y(jj),
det_JIK(jj),'.g','MarkerSize',15);
    end

end


% |J_FK| as a function of (x,y)
det_JFK = RESULTS.detJFK ;

fig3 = figure;
figure(fig3);

plot3(Coord_x, Coord_y, det_JFK, '.g', 'MarkerSize', 7);
hold on; grid;

if opt ~= 4
    DJFK = griddata(Coord_x, Coord_y, det_JFK, X, Y) ;

    mesh(X,Y,DJFK, 'FaceColor', 'g', 'EdgeColor', 'g', 'FaceAlpha',
0.3, 'EdgeAlpha', 0.3) ;
    contour3(X,Y,DJFK, 'LevelList', 0, 'LineColor', 'black',
'LineWidth', 2);
end
```

```
xlabel('X Coordinate');
ylabel('Y Coordinate');
zlabel('|J_F_K|');

% Forward problem singularities are marked
for jj = 1:length(det_JFK)
    if abs(det_JFK(jj)) <= tol
        plot3(Coord_x(jj), Coord_y(jj),
det_JFK(jj),'.b','MarkerSize',15);
    end

end

% |K| as a function of (x,y)
det_K = RESULTS.detK ;

fig3 = figure;
figure(fig3);

plot3(Coord_x, Coord_y, det_K, '.b', 'MarkerSize', 7);
hold on; grid;

if opt ~= 4
    DK = griddata(Coord_x, Coord_y, det_K, X, Y) ;
    surf(X,Y,DK, 'FaceColor', 'b', 'EdgeColor', 'b', 'FaceAlpha', 0.3,
'EdgeAlpha', 0.3) ;
    contour3(X,Y,DK, 'LevelList', 0, 'LineColor', 'black',
'LineWidth', 2);
end

xlabel('X Coordinate');
ylabel('Y Coordinate');
zlabel('|K|');
```

3.2. INTEGRACIÓN NUMÉRICA. PROBLEMA DIRECTO

```
% Algorithm that solves de Forward Kinematic position problem through
the
% the use of the Shootin Method
clear; close all;
addpath('EllipticIntegrals_functions');

% Creation (1) or not(0) of a .gif file
graf_gif = 0 ;

% Data graphic representation as funtions of (x, y)(repr = 1) or
% (theta1, theta2)(repr = 2)
repr = 1 ;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% INPUT DATA
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
tol = 1e-4 ;

% Loading data of the mechanism in the initial position
load('RODS');

% Data for the Forward Kinematic position problem
% Option 1: Reading data from xls file
% Option 2: Circular path on (theta1, theta2)
% Option 3: Rectangular path (theta1, theta2)
% Option 4: Straigth path (theta1, theta2)
opt = 4 ;

if opt == 1
    data = xlsread('FK_inputs.xlsx') ;

    incr_theta1 = data(:,1) ;  % Increment of lambda1 respect to
initial
                               % position
    incr_theta2 = data(:,2) ;  % Increment of lambda2 respect to
initial
                               % position
    Fext     = data(:,3:4) ;   % External forces. The first column is
the
                               % x component of the force and the
second
                               % colum, the y component

elseif opt == 2
    D = 0.2 ;      % Diameter of external circle
    Ncir = 5 ;     % Number of concentric circles
    Ndiv = 8 ;     % Number of divisions
    [incr_theta1, incr_theta2] = Workspace_Circles (D, Ncir, Ndiv);

    Fext = zeros (length(incr_theta1),2) ;

elseif opt == 3
    b = 0.2 ;     % Base of external rectangle
    h = 0.2 ;     % Heigth of external rectangle
    N = 4 ;       % Number of rectangles
    Ndiv = 4 ;    % Number of divisions for each side
    [incr_theta1, incr_theta2] = Workspace_Rectangles (b, h, N, Ndiv)
;
```

```
    Fext = zeros (length(incr_theta1),2) ;

elseif opt == 4
    L = 1 ;             % Path length
    phi = 0/180*pi ;     % Orientation angle [rad]
    Ndiv = 100 ;           % Number of divisions
    [incr_theta1, incr_theta2] = StraigthPath(L, phi, Ndiv) ;

    Fext = zeros (length(incr_theta1),2) ;

end



% Number of positions to be calculated
Nii = length(incr_theta1) ;

% Inital position
Theta1_ref = RODS.IN.B1.theta0 ;
Theta2_ref = RODS.IN.B2.theta0 ;


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% OUTPUT DATA
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
RESULTS.XP      = zeros(Nii, 1) ;
RESULTS.YP      = zeros(Nii, 1) ;
RESULTS.MOMENT1 = zeros(Nii, 1) ;
RESULTS.MOMENT2 = zeros(Nii, 1) ;

RESULTS.FK_J11 = zeros(Nii, 1) ;
RESULTS.FK_J12 = zeros(Nii, 1) ;
RESULTS.FK_J21 = zeros(Nii, 1) ;
RESULTS.FK_J22 = zeros(Nii, 1) ;
RESULTS.detJFK = zeros(Nii, 1) ;

RESULTS.IK_J11 = zeros(Nii, 1) ;
RESULTS.IK_J12 = zeros(Nii, 1) ;
RESULTS.IK_J21 = zeros(Nii, 1) ;
RESULTS.IK_J22 = zeros(Nii, 1) ;
RESULTS.detJIK = zeros(Nii, 1) ;

RESULTS.K_11 = zeros(Nii, 1) ;
RESULTS.K_12 = zeros(Nii, 1) ;
RESULTS.K_21 = zeros(Nii, 1) ;
RESULTS.K_22 = zeros(Nii, 1) ;
RESULTS.detK = zeros(Nii, 1) ;


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% CALCULATION
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
h = figure(1);
if graf_gif == 1
    axis tight manual
    filename = 'FK_testAnimated.gif';
end

for ii = 1:Nii
```

```
% Update the position of the clamped end of the rod
RODS.IN.B1.theta0 = Theta1_ref + incr_theta1(ii) ;
RODS.IN.B2.theta0 = Theta2_ref + incr_theta2(ii) ;

% Shooting method
RODS = FK_ShootingMethod( RODS, Fext(ii,:)' ) ;

% Forward Kinematic Jacobian matrix
FK_Jacob = FK_Jacobian( RODS, Fext(ii,:)' ) ;

% Inverse Kinematic Jacobian matrix
IK_Jacob = IK_Jacobian( RODS, Fext(ii,:)' ) ;

% Stiffness matrix
K = StiffnessMatrix_SM( RODS, Fext(ii,:)' ) ;

if RODS.OUT.sol == 0
    %          fprintf('Aborted calculation\n') ;
    %          break ;

    % Solution is not valid. Invalid data are saved as NaN
    RESULTS.XP(ii)       = NaN ;
    RESULTS.YP(ii)       = NaN ;
    RESULTS.MOMENT1(ii)  = NaN ;
    RESULTS.MOMENT2(ii)  = NaN ;

    RESULTS.FK_J11(ii)  = NaN ;
    RESULTS.FK_J12(ii)  = NaN ;
    RESULTS.FK_J21(ii)  = NaN ;
    RESULTS.FK_J22(ii)  = NaN ;
    RESULTS.detJFK(ii)  = NaN ;

    RESULTS.IK_J11(ii)  = NaN ;
    RESULTS.IK_J12(ii)  = NaN ;
    RESULTS.IK_J21(ii)  = NaN ;
    RESULTS.IK_J22(ii)  = NaN ;
    RESULTS.detJIK(ii)  = NaN ;

    RESULTS.K_11(ii)  = NaN ;
    RESULTS.K_12(ii)  = NaN ;
    RESULTS.K_21(ii)  = NaN ;
    RESULTS.K_22(ii)  = NaN ;
    RESULTS.detK(ii)  = NaN ;

else

    % Storing results
    RESULTS.XP(ii)       = RODS.IN.OP(1) ;
    RESULTS.YP(ii)       = RODS.IN.OP(2) ;
    RESULTS.MOMENT1(ii)  = RODS.OUT.B1.Moment ;
    RESULTS.MOMENT2(ii)  = RODS.OUT.B2.Moment ;

    RESULTS.FK_J11(ii)  = FK_Jacob(1,1) ;
    RESULTS.FK_J12(ii)  = FK_Jacob(1,2) ;
    RESULTS.FK_J21(ii)  = FK_Jacob(2,1) ;
    RESULTS.FK_J22(ii)  = FK_Jacob(2,2) ;
    RESULTS.detJFK(ii)  = det(FK_Jacob) ;

    RESULTS.IK_J11(ii)  = IK_Jacob(1,1) ;
```

```
        RESULTS.IK_J12(ii) = IK_Jacob(1,2) ;
        RESULTS.IK_J21(ii) = IK_Jacob(2,1) ;
        RESULTS.IK_J22(ii) = IK_Jacob(2,2) ;
        RESULTS.detJIK(ii) = det(IK_Jacob) ;

        RESULTS.K_11(ii) = K(1,1) ;
        RESULTS.K_12(ii) = K(1,2) ;
        RESULTS.K_21(ii) = K(2,1) ;
        RESULTS.K_22(ii) = K(2,2) ;
        RESULTS.detK(ii) = det(K) ;

        % Drawing the mechanism
        clf;
        axis equal; grid; hold on ;
        GraficRepresentation( RODS ) ;
        plot(RESULTS.XP(1:ii), RESULTS.YP(1:ii), '.-m') ;
        pause(1e-3) ;


        % Creation of a .gif file
        if graf_gif == 1

            drawnow
            % Capture the plot as an image
            frame = getframe(h);
            im = frame2im(frame);
            [imind,cm] = rgb2ind(im,256);

            % Write to the GIF File
            if ii == 1
                imwrite(imind,cm,filename,'gif', 'Loopcount',inf);
            else
                imwrite(imind,cm,filename,'gif','WriteMode','append');
            end

        end


        clc;
        fprintf('%f \n%%',(ii/Nii*100));

    end

end


if opt == 1

    matrix = [RESULTS.XP RESULTS.YP ] ;
    xlRange = ['F2:G' num2str(Nii+1)] ;
    xlswrite('FK_inputs.xlsx', matrix, xlRange) ;

    matrix = [ RESULTS.MOMENT1 RESULTS.MOMENT2 ] ;
    xlRange = ['I2:J' num2str(Nii+1)] ;
    xlswrite('FK_inputs.xlsx', matrix, xlRange) ;

    matrix = [ RESULTS.FK_J11 RESULTS.FK_J12 ...
        RESULTS.FK_J21 RESULTS.FK_J22 ] ;
```

```
    xlRange = ['L2:O' num2str(Nii+1)] ;
    xlswrite('FK_inputs.xlsx', matrix, xlRange) ;

    xlRange = ['P2:P' num2str(Nii+1)] ;
    xlswrite('FK_inputs.xlsx', RESULTS.detJFK, xlRange) ;

    matrix = [ RESULTS.IK_J11 RESULTS.IK_J12 ...
        RESULTS.IK_J21 RESULTS.IK_J22 ] ;
    xlRange = ['R2:U' num2str(Nii+1)] ;
    xlswrite('FK_inputs.xlsx', matrix, xlRange) ;

    xlRange = ['V2:V' num2str(Nii+1)] ;
    xlswrite('FK_inputs.xlsx', RESULTS.detJIK, xlRange) ;

    matrix = [ RESULTS.K_11 RESULTS.K_12 ...
               RESULTS.K_21 RESULTS.K_22 ] ;
    xlRange = ['X2:AA' num2str(Nii+1)] ;
    xlswrite('FK_inputs.xlsx', matrix, xlRange) ;

    xlRange = ['AB2:AB' num2str(Nii+1)] ;
    xlswrite('FK_inputs.xlsx', RESULTS.detK, xlRange) ;

end


%%%%%%%%%%%%%%%%%%%%%%%%%%% PLOTTING RESULTS
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


Coord_x = RESULTS.XP ;
Coord_y = RESULTS.YP ;

Theta_1 = Theta1_ref + incr_theta1 ;
Theta_2 = Theta2_ref + incr_theta2 ;


% Plotting results as functions of (x,y)
if repr == 1
    [X, Y] = meshgrid(Coord_x, Coord_y);

    % Theta1, Theta2 values as functions of (x,y)

    fig1 = figure;
    figure(fig1);

    d1 = plot3(Coord_x, Coord_y, Theta_1, '.r', 'MarkerSize', 7);
    hold on; grid;

    d2 = plot3(Coord_x, Coord_y, Theta_2, '.b', 'MarkerSize',7);

    THETA1 = griddata(Coord_x, Coord_y, Theta_1, X, Y) ;
    THETA2 = griddata(Coord_x, Coord_y, Theta_2, X, Y) ;

    mesh(X,Y,THETA1, 'FaceColor', 'r', 'EdgeColor', 'r', 'FaceAlpha',
0.2, 'EdgeAlpha', 0.2) ;
    mesh(X,Y,THETA2, 'FaceColor', 'b', 'EdgeColor', 'b', 'FaceAlpha',
0.2, 'EdgeAlpha', 0.2) ;
```

```
    legend([d1, d2], 'Theta 1', 'Theta 2');
    xlabel('X Coordinate');
    ylabel('Y Coordinate');
    zlabel(['{\theta}_' num2str(1) ', {\theta}_' num2str(2)]);


    % |J_IK| as a function of (x,y)
    det_JIK = RESULTS.detJIK ;

    fig2 = figure;
    figure(fig2);

    plot3(Coord_x, Coord_y, det_JIK, '.r', 'MarkerSize', 7);
    hold on; grid;

    DJIK = griddata(Coord_x, Coord_y, det_JIK, X, Y) ;
    mesh(X,Y,DJIK, 'FaceColor', 'r', 'EdgeColor', 'r', 'FaceAlpha',
0.3, 'EdgeAlpha', 0.3) ;
    contour3(X,Y,DJIK, 'LevelList', 0, 'LineColor', 'black',
'LineWidth', 2);

    xlabel('X Coordinate');
    ylabel('Y Coordinate');
    zlabel('|J_I_K|');

    % Inverse problem singularities are marked
    for jj = 1:length(det_JIK)
        if abs(det_JIK(jj)) <= tol
            plot3(Coord_x(jj), Coord_y(jj),
det_JIK(jj),'.g','MarkerSize',15);
        end

    end


    % |J_FK| as a function of (x,y)
    det_JFK = RESULTS.detJFK ;

    fig3 = figure;
    figure(fig3);

    plot3(Coord_x, Coord_y, det_JFK, '.g', 'MarkerSize', 7);
    hold on; grid;

    DJFK = griddata(Coord_x, Coord_y, det_JFK, X, Y) ;
    surf(X,Y,DJFK, 'FaceColor', 'g', 'EdgeColor', 'g', 'FaceAlpha',
0.3, 'EdgeAlpha', 0.3) ;
    contour3(X,Y,DJFK, 'LevelList', 0, 'LineColor', 'black',
'LineWidth', 2);

    xlabel('X Coordinate');
    ylabel('Y Coordinate');
    zlabel('|J_F_K|');

    % Forward problem singularities are marked
    for jj = 1:length(det_JFK)
        if abs(det_JFK(jj)) <= tol
```

```
            plot3(Coord_x(jj), Coord_y(jj),
det_JFK(jj),'.b','MarkerSize',15);
        end

    end

    % |K| as a function of (x,y)
    det_K = RESULTS.detK ;

    fig3 = figure;
    figure(fig3);

    plot3(Coord_x, Coord_y, det_K, '.b', 'MarkerSize', 7);
    hold on; grid;

    if opt ~= 4
        DK = griddata(Coord_x, Coord_y, det_K, X, Y) ;
        surf(X,Y,DK, 'FaceColor', 'b', 'EdgeColor', 'b', 'FaceAlpha',
0.3, 'EdgeAlpha', 0.3) ;
        contour3(X,Y,DK, 'LevelList', 0, 'LineColor', 'black',
'LineWidth', 2);
    end

    xlabel('X Coordinate');
    ylabel('Y Coordinate');
    zlabel('|K|');




% Plotting results as functions of (theta1,theta2)
elseif repr == 2
    [THETA1, THETA2] = meshgrid(Theta_1, Theta_2);

    % x, y as functions of (theta1, theta2)

    fig1 = figure;
    figure(fig1);

    plot3(Theta_1, Theta_2, Coord_x, '.r', 'MarkerSize', 7);
    hold on; grid;

    plot3(Theta_1, Theta_2, Coord_y, '.b', 'MarkerSize',7);

    if opt ~= 4
        X = griddata(Theta_1, Theta_2, Coord_x, THETA1, THETA2) ;
        Y = griddata(Theta_1, Theta_2, Coord_y, THETA1, THETA2) ;

        mesh(THETA1, THETA2, X, 'FaceColor', 'r', 'EdgeColor', 'r',
'FaceAlpha', 0.2, 'EdgeAlpha', 0.2) ;
        mesh(THETA1, THETA2, Y, 'FaceColor', 'b', 'EdgeColor', 'b',
'FaceAlpha', 0.2, 'EdgeAlpha', 0.2) ;
    end

    legend('x_P', 'y_P');
    xlabel('{\theta}_1');
    ylabel('{\theta}_2');
    zlabel(['X, Y']);
```

```
    % |J_IK| as a function of (x,y)
    det_JIK = RESULTS.detJIK ;

    fig2 = figure;
    figure(fig2);

    plot3(Theta_1, Theta_2, det_JIK, '.r', 'MarkerSize', 7);
    hold on; grid;

    if opt ~= 4
        DJIK = griddata(Theta_1, Theta_2, det_JIK, THETA1, THETA2) ;
        mesh(THETA1, THETA2, DJIK, 'FaceColor', 'r', 'EdgeColor', 'r',
'FaceAlpha', 0.3, 'EdgeAlpha', 0.3) ;
        contour3(THETA1, THETA2, DJIK, 'LevelList', 0, 'LineColor',
'black', 'LineWidth', 2);
    end

    xlabel('{\theta}_1');
    ylabel('{\theta}_2');
    zlabel('|J_I_K|');

    % Inverse problem singularities are marked
    for jj = 1:length(det_JIK)
        if abs(det_JIK(jj)) <= tol
            plot3(Theta_1(jj), Theta_2(jj),
det_JIK(jj),'.g','MarkerSize',15);
        end
    end

    % |J_FK| as a function of (x,y)
    det_JFK = RESULTS.detJFK ;

    fig3 = figure;
    figure(fig3);

    plot3(Theta_1, Theta_2, det_JFK, '.g', 'MarkerSize', 7);
    hold on; grid;

    if opt ~= 4
        DJFK = griddata(Theta_1, Theta_2, det_JFK, THETA1, THETA2) ;
        surf(THETA1, THETA2, DJFK, 'FaceColor', 'g', 'EdgeColor', 'g',
'FaceAlpha', 0.3, 'EdgeAlpha', 0.3) ;
        contour3(THETA1, THETA2, DJFK, 'LevelList', 0, 'LineColor',
'black', 'LineWidth', 2);
    end

    xlabel('{\theta}_1');
    ylabel('{\theta}_2');
    zlabel('|J_F_K|');

    % Forward problem singularities are marked
    for jj = 1:length(det_JFK)
        if abs(det_JFK(jj)) <= tol
            plot3(Theta_1(jj), Theta_2(jj),
det_JFK(jj),'.b','MarkerSize',15);
        end
```

```
    end

    % |K| as a function of (x,y)
    det_K = RESULTS.detK ;

    fig3 = figure;
    figure(fig3);

    plot3(Theta_1, Theta_2, det_K, '.b', 'MarkerSize', 7);
    hold on; grid;

    if opt ~= 4
        DK = griddata(Theta_1, Theta_2, det_K, THETA1, THETA2) ;
        surf(THETA1,THETA2,DK, 'FaceColor', 'b', 'EdgeColor', 'b',
'FaceAlpha', 0.3, 'EdgeAlpha', 0.3) ;
        contour3(THETA1,THETA2,DK, 'LevelList', 0, 'LineColor',
'black', 'LineWidth', 2);
    end

    xlabel('X Coordinate');
    ylabel('Y Coordinate');
    zlabel('|K|');

end
```

3.3. INTEGRACIÓN NUMÉRICA. FUNCIONES ASOCIADAS

```
function [ Jacob ] = FK_Jacobian( RODS, Fext )
% Calculation of the Jacobian for the FK position problem
Jacob = zeros(2) ;

RODS_ref = RODS ;

% Increment of value of each lambda to calculate nummerically the
Jacobian
epsilon = 1e-6 ;

% Increment of lambda 1
RODS.IN.B1.theta0 = RODS.IN.B1.theta0 + epsilon ;
RODS = FK_ShootingMethod( RODS, Fext ) ;
if RODS.OUT.sol == 1
    Jacob(:,1) = [ RODS.IN.OP ] - ...
                 [ RODS_ref.IN.OP ] ;
    RODS.IN.B1.theta0 = RODS.IN.B1.theta0 - epsilon ;
else
    Jacob(:,1) = NaN ;
end


% Increment of lambda 2
RODS.IN.B2.theta0 = RODS.IN.B2.theta0 + epsilon ;
RODS = FK_ShootingMethod( RODS, Fext ) ;
if RODS.OUT.sol == 1
    Jacob(:,2) = [ RODS.IN.OP ] - ...
                 [ RODS_ref.IN.OP ] ;
    RODS.IN.B2.theta0 = RODS.IN.B2.theta0 - epsilon ;
else
    Jacob(:,2) = NaN ;
end


Jacob = Jacob/epsilon ;

end

function [ RODS ] = FK_ShootingMethod( RODS, Fext )
% Shooting method to solve the Forward Kinematic position problem

% Absolute and relative tolerance of the element of the residue vector
tol = 1e-8 ;


% Runge-Kuta method of order 4 for each rod
[ RODS.OUT.B1 ] = RK4_rod( RODS.IN.B1 ) ;
[ RODS.OUT.B2 ] = RK4_rod( RODS.IN.B2 ) ;

% Update de new position at pinned end (known value)
RODS.IN.B1.pL = RODS.IN.OP ;
RODS.IN.B2.pL = RODS.IN.OP ;

% Initial residue vector
resid = Residue( RODS, Fext ) ;

% Residue vector of the former iteration. Before the iteration (while
```

```
% loop), this residue vector is null
resid0 = zeros(size(resid)) ;


while max(abs(resid)) > tol && max(abs(resid-resid0)) > tol

    resid0 = resid ;

    % Jacobian of the residue vector respect the variables that acts
as
    % guess values
    J = Jacobian( RODS, Fext ) ;

    % Increment of the guess values
    delta = -J\resid0 ;

    % Updating guess values
    RODS.IN.B1.m0 = RODS.IN.B1.m0 + delta(1) ;
    RODS.IN.B1.n0 = RODS.IN.B1.n0 + delta(2:3) ;
    RODS.IN.B2.m0 = RODS.IN.B2.m0 + delta(4) ;
    RODS.IN.B2.n0 = RODS.IN.B2.n0 + delta(5:6) ;
    RODS.IN.OP    = RODS.IN.OP    + delta(7:8) ;

    % Runge-Kuta method of order 4 for each rod with the updated guess
    % values
    [ RODS.OUT.B1 ] = RK4_rod( RODS.IN.B1 ) ;
    [ RODS.OUT.B2 ] = RK4_rod( RODS.IN.B2 ) ;

    % Update the new guess position at pinned end (known value)
    RODS.IN.B1.pL = RODS.IN.OP ;
    RODS.IN.B2.pL = RODS.IN.OP ;

    % New resude vector
    resid = Residue( RODS, Fext ) ;

    % In case in which the new residue vector is bigger that the
previous
    % one, the incrementation of the guess values are forced to take
half
    % of value calculated. This operation is applied iteratively until
the
    % new residue vector is smaller that the previous one
    while max(abs(resid)) > max(abs(resid0)) &&
max(abs(delta))>tol*1e-6
        delta = 0.5*delta ;

        RODS.IN.B1.m0 = RODS.IN.B1.m0 - delta(1) ;
        RODS.IN.B1.n0 = RODS.IN.B1.n0 - delta(2:3) ;
        RODS.IN.B2.m0 = RODS.IN.B2.m0 - delta(4) ;
        RODS.IN.B2.n0 = RODS.IN.B2.n0 - delta(5:6) ;
        RODS.IN.OP    = RODS.IN.OP    - delta(7:8) ;

        [ RODS.OUT.B1 ] = RK4_rod( RODS.IN.B1 ) ;
        [ RODS.OUT.B2 ] = RK4_rod( RODS.IN.B2 ) ;

        % Update the new guess position at pinned end (known value)
        RODS.IN.B1.pL = RODS.IN.OP ;
        RODS.IN.B2.pL = RODS.IN.OP ;
```

```
        resid = Residue( RODS, Fext ) ;
    end

end


% If the solution has not been reached or the reside is NaN, the
variable
% "sol" take the value 0
if max(abs(resid)) > tol || not(sum(abs(resid))>=0)
    RODS.OUT.sol = 0 ;
else
    RODS.OUT.sol = 1 ;

    % Froce applied at clamped end projected in the direction of each
    % linear guide
    RODS.OUT.B1.Moment = RODS.IN.B1.m0 ;
    RODS.OUT.B2.Moment = RODS.IN.B2.m0 ;

end





function [ Jacob ] = Jacobian( RODS, Fext )
% Functions that calculates numerically the Jacobian of the residue
vector
% respect to the variables of the problem (guess values)
Jacob = zeros(8) ;


% Increment of the variables (guess values) to calculate numerically
the
% Jacobian
epsilon = 1e-11 ;


% Variables for rod 1
RODS.IN.B1.m0 = RODS.IN.B1.m0 + epsilon ;
OUT_aux = RK4_rod( RODS.IN.B1 ) ;
Jacob(1:3,1) = [ OUT_aux.m(end) ;
                  OUT_aux.p(:,end) ] - ...
                [ RODS.OUT.B1.m(end) ;
                  RODS.OUT.B1.p(:,end) ] ;
Jacob(1:3,1) = Jacob(1:3,1)/epsilon ;
RODS.IN.B1.m0 = RODS.IN.B1.m0 - epsilon ;

RODS.IN.B1.n0(1) = RODS.IN.B1.n0(1) + epsilon ;
OUT_aux = RK4_rod( RODS.IN.B1 ) ;
Jacob(1:3,2) = [ OUT_aux.m(end) ;
                  OUT_aux.p(:,end) ] - ...
                [ RODS.OUT.B1.m(end) ;
                  RODS.OUT.B1.p(:,end) ] ;
Jacob(1:3,2) = Jacob(1:3,2)/epsilon ;
RODS.IN.B1.n0(1) = RODS.IN.B1.n0(1) - epsilon ;

RODS.IN.B1.n0(2) = RODS.IN.B1.n0(2) + epsilon ;
OUT_aux = RK4_rod( RODS.IN.B1 ) ;
```

```
Jacob(1:3,3) = [ OUT_aux.m(end) ;
                 OUT_aux.p(:,end) ] - ...
               [ RODS.OUT.B1.m(end) ;
                 RODS.OUT.B1.p(:,end) ] ;
Jacob(1:3,3) = Jacob(1:3,3)/epsilon ;
RODS.IN.B1.n0(2) = RODS.IN.B1.n0(2) - epsilon ;


% Variables for rod 2
RODS.IN.B2.m0 = RODS.IN.B2.m0 + epsilon ;
OUT_aux = RK4_rod( RODS.IN.B2 ) ;
Jacob(4:6,4) = [ OUT_aux.m(end) ;
                 OUT_aux.p(:,end) ] - ...
               [ RODS.OUT.B2.m(end) ;
                 RODS.OUT.B2.p(:,end) ] ;
Jacob(4:6,4) = Jacob(4:6,4)/epsilon ;
RODS.IN.B2.m0 = RODS.IN.B2.m0 - epsilon ;

RODS.IN.B2.n0(1) = RODS.IN.B2.n0(1) + epsilon ;
OUT_aux = RK4_rod( RODS.IN.B2 ) ;
Jacob(4:6,5) = [ OUT_aux.m(end) ;
                 OUT_aux.p(:,end) ] - ...
               [ RODS.OUT.B2.m(end) ;
                 RODS.OUT.B2.p(:,end) ] ;
Jacob(4:6,5) = Jacob(4:6,5)/epsilon ;
RODS.IN.B2.n0(1) = RODS.IN.B2.n0(1) - epsilon ;

RODS.IN.B2.n0(2) = RODS.IN.B2.n0(2) + epsilon ;
OUT_aux = RK4_rod( RODS.IN.B2 ) ;
Jacob(4:6,6) = [ OUT_aux.m(end) ;
                 OUT_aux.p(:,end) ] - ...
               [ RODS.OUT.B2.m(end) ;
                 RODS.OUT.B2.p(:,end) ] ;
Jacob(4:6,6) = Jacob(4:6,6)/epsilon ;
RODS.IN.B2.n0(2) = RODS.IN.B2.n0(2) - epsilon ;


% Variables of the position and orientation of the end effecto
plattform
Jacob([2 5],7) = -1 ;
Jacob([3 6],8) = -1 ;

% Unit elements
Jacob(7,[2 5]) = 1 ;
Jacob(8,[3 6]) = 1 ;




function [ resid ] = Residue( RODS, Fext )
% Function of the residue vector of the whole mechanism

resid = [ RODS.OUT.B1.m(end) ;
          RODS.OUT.B1.p(:,end) - RODS.IN.B1.pL ;
          RODS.OUT.B2.m(end) ;
          RODS.OUT.B2.p(:,end) - RODS.IN.B2.pL ;
          RODS.IN.B1.n0 + RODS.IN.B2.n0 - Fext] ;
```

```
function [ OUT ] = RK4_rod ( IN )
% Functions that solves de initial value problem using the method of
% Runge-Kutta of order four.
%   IN:  Structure in which are geometric and mechanical parameters
%       IN.L        Length of the rod [m]
%       IN.N        Number of nodes of the rod
%       IN.EI       Bending stiffness of the rod [Pa]
%       IN.d        Distance of the rigid part located in pinned end
of the
%                   rod [m]
%       IN.alpha    Angle of the guide [rad]
%       In.theta0   Angle of the rod at clamped end [rad]
%       IN.lambda   Value of lambda [m]
%       IN.m0       Bending at the clamped end [N.m]
%       IN.n0       Internal force at the clamped end (2x1 vector) [N]

%   OUT:    Structure in which are saved the solution
%       OUT.p       array (2xIN.N) with the x,y component of the
%                   centroid position of the rod
%       OUT.m       array (1xIN.N) with the bending moment in each
node
%                   of the rod
%       OUT.theta   array (1xIN.N) with the angle of the tangent of
%                   the rod
%       OUT.Ener    Elastic deformation enegy of the rod using the
Simpson
%                   rule


% Increment of length
ds = IN.L/(IN.N-1) ;

%%%%%%%%%%%%%%%%%%%%%% Variables to save the solution
%%%%%%%%%%%%%%%%%%%%%
OUT.p     = zeros(2,IN.N) ; % Centroid position vector function (x,y)
OUT.theta = zeros(1,IN.N) ;   % Angle function of the tangent of the
rod
OUT.m     = zeros(1,IN.N) ; % Bending moment function

% Variables at the first point (clamped end)
OUT.p(:,1)    = IN.OA + IN.lambda*[cos(IN.theta0); sin(IN.theta0)];
OUT.theta(1) = IN.theta0 ;
OUT.m(1)     = IN.m0 ;

% Vector with the values of the dependent variables at the clamped end
var = [ OUT.p(:,1) ;
        OUT.theta(1) ;
        OUT.m(1) ] ;

% "For" loop to integrate the system of differential equations through
the
% rod
for ii = 2:IN.N

    % Runge-Kutta method of order four
    k1 = ds *  Right_function( IN.EI, IN.n0, var ) ;
    k2 = ds *  Right_function( IN.EI, IN.n0, var + 0.5*k1 ) ;
```

```
    k3 = ds *  Right_function( IN.EI, IN.n0, var + 0.5*k2 ) ;
    k4 = ds *  Right_function( IN.EI, IN.n0, var + k3 ) ;

    var = var + (k1 + 2*k2 + 2*k3 + k4)/6 ;

    OUT.p(:,ii)    = var(1:2) ;
    OUT.theta(ii) = var(3) ;
    OUT.m(ii)      = var(4) ;

end


% Vector product of pinned end position and force applied on it
OUT.pxn = OUT.p(1,end)*IN.n0(2)-OUT.p(2,end)*IN.n0(1) ;


% Elastic energy calculated throught the the Simpson's rule
m2 = OUT.m(1:end-1).^2 ;         % Square of the internal moemnt
funciton

OUT.Ener = ds/3 * ( m2(1) + m2(end) + 4*sum(m2(2:2:(IN.N-1))) + ...
           2*sum(m2(3:2:(IN.N-2))) ) ;
OUT.Ener = OUT.Ener/IN.EI ;




function [ func ] = Right_function( EI, n, var )
% Value of the right part of the system of differential equations
% var(1)    px
% var(2)    py
% var(3)    theta
% var(4)    moment

func = [ cos(var(3)) ;
         sin(var(3)) ;
         var(4)/EI ;
         n(1)*sin(var(3)) - n(2)*cos(var(3)) ] ;


function [  ] = GraficRepresentation( RODS )
% Plotting each rod and end effector plattform of the mechanism

% Rod 1
plot(RODS.OUT.B1.p(1,1:end-1),RODS.OUT.B1.p(2,1:end-1), '.-', 'color',
[0 0.6275 0]) ;
plot(RODS.OUT.B1.p(1,end-1:end),RODS.OUT.B1.p(2,end-1:end), '-',
'color', [0 0.6275 0]) ;

% Rod 2
plot(RODS.OUT.B2.p(1,1:end-1),RODS.OUT.B2.p(2,1:end-1), '.-', 'color',
[0 0 0.6275]) ;
plot(RODS.OUT.B2.p(1,end-1:end),RODS.OUT.B2.p(2,end-1:end),'-',
'color', [0 0 0.6275]) ;

% % Rod 3
% plot(RODS.OUT.B3.p(1,1:end-1),RODS.OUT.B3.p(2,1:end-1),'.-',
'color', [0.6275 0 0]) ;
```

```
% plot(RODS.OUT.B3.p(1,end-1:end),RODS.OUT.B3.p(2,end-1:end),'-',
'color', [0.6275 0 0]) ;

% % End effector plattform
% x_vert = [ RODS.IN.B1.pL(1) RODS.IN.B2.pL(1) RODS.IN.B3.pL(1)
RODS.IN.B1.pL(1) ] ;
% y_vert = [ RODS.IN.B1.pL(2) RODS.IN.B2.pL(2) RODS.IN.B3.pL(2)
RODS.IN.B1.pL(2) ] ;
% plot(x_vert, y_vert, 'k', 'linewidth', 2) ;

% End effector
plot(RODS.IN.B1.pL(1), RODS.IN.B1.pL(2), '.k', 'MarkerSize', 15);

% % End effector orientation
% const = norm(RODS.IN.B1.r) ;
% quiver( RODS.IN.OP(1), RODS.IN.OP(2), const*cos(RODS.IN.theta),
const*sin(RODS.IN.theta), 'r' ) ;
% quiver( RODS.IN.OP(1), RODS.IN.OP(2), -const*sin(RODS.IN.theta),
const*cos(RODS.IN.theta), 'g' ) ;

% % Value of lambda
% x1_guide = [ 0 RODS.IN.B1.lambda*cos(RODS.IN.B1.alpha) ] +
RODS.IN.B1.OA(1) ;
% y1_guide = [ 0 RODS.IN.B1.lambda*sin(RODS.IN.B1.alpha) ] +
RODS.IN.B1.OA(2) ;
% plot(x1_guide, y1_guide, '-', 'color', [0 0.6275 0], 'linewidth', 4)
;
%
% x2_guide = [ 0 RODS.IN.B2.lambda*cos(RODS.IN.B2.alpha) ] +
RODS.IN.B2.OA(1) ;
% y2_guide = [ 0 RODS.IN.B2.lambda*sin(RODS.IN.B2.alpha) ] +
RODS.IN.B2.OA(2) ;
% plot(x2_guide, y2_guide, '-', 'color', [0 0 0.6275], 'linewidth', 4)
;
%
% x3_guide = [ 0 RODS.IN.B3.lambda*cos(RODS.IN.B3.alpha) ] +
RODS.IN.B3.OA(1) ;
% y3_guide = [ 0 RODS.IN.B3.lambda*sin(RODS.IN.B3.alpha) ] +
RODS.IN.B3.OA(2) ;
% plot(x3_guide, y3_guide, '-', 'color', [0.6275 0 0], 'linewidth', 4)
;

end

function [ Jacob ] = IK_Jacobian( RODS, Fext )
% Calculation of the Jacobian for the FK position problem
Jacob = zeros(2) ;

RODS_ref = RODS ;

% Increment of value of each lambda to calculate nummerically the
Jacobian
epsilon = 1e-6 ;

% Increment of OP, x component
RODS.IN.OP(1) = RODS.IN.OP(1) + epsilon ;
RODS = IK_ShootingMethod( RODS, Fext ) ;
if RODS.OUT.sol == 1
    Jacob(:,1) = [ RODS.IN.B1.theta0 ;
                   RODS.IN.B2.theta0 ] - ...
```

```
                  [ RODS_ref.IN.B1.theta0 ;
                    RODS_ref.IN.B2.theta0 ] ;
    RODS.IN.OP(1) = RODS.IN.OP(1) - epsilon ;
else
    Jacob(:,1) = NaN ;
end

% Increment of OP, y component
RODS.IN.OP(2) = RODS.IN.OP(2) + epsilon ;
RODS = IK_ShootingMethod( RODS, Fext ) ;
if RODS.OUT.sol == 1
    Jacob(:,2) = [ RODS.IN.B1.theta0 ;
                   RODS.IN.B2.theta0 ] - ...
                 [ RODS_ref.IN.B1.theta0 ;
                   RODS_ref.IN.B2.theta0 ] ;
    RODS.IN.OP(2) = RODS.IN.OP(2) - epsilon ;
else
    Jacob(:,2) = NaN ;
end


Jacob = Jacob/epsilon ;

end


function [ RODS ] = IK_ShootingMethod( RODS, Fext )
% Shooting method to solve the Inverse Kinematic position problem

% Absolute and relative tolerance of the element of the residue vector
tol = 1e-8 ;


% Update de new position at pinned end (known value)
RODS.IN.B1.pL = RODS.IN.OP ;
RODS.IN.B2.pL = RODS.IN.OP ;

% Runge-Kuta method of order 4 for each rod
[ RODS.OUT.B1 ] = RK4_rod( RODS.IN.B1 ) ;
[ RODS.OUT.B2 ] = RK4_rod( RODS.IN.B2 ) ;

% Initial residue vector
resid = Residue( RODS, Fext ) ;


% Residue vector of the former iteration. Before the iteration (while
% loop), this residue vector is null
resid0 = zeros(size(resid)) ;


while max(abs(resid)) > tol && max(abs(resid-resid0)) > tol

    resid0 = resid ;

    % Jacobian of the residue vector respect the variables that acts
as
    % guess values
    J = Jacobian( RODS ) ;
```

```
    % Increment of the guess values
    delta = -J\resid0 ;

    % Updating guess values
    RODS.IN.B1.m0     = RODS.IN.B1.m0     + delta(1) ;
    RODS.IN.B1.n0     = RODS.IN.B1.n0     + delta(2:3) ;
    RODS.IN.B1.theta0 = RODS.IN.B1.theta0 + delta(4) ;
    RODS.IN.B2.m0     = RODS.IN.B2.m0     + delta(5) ;
    RODS.IN.B2.n0     = RODS.IN.B2.n0     + delta(6:7) ;
    RODS.IN.B2.theta0 = RODS.IN.B2.theta0 + delta(8) ;

    % Runge-Kuta method of order 4 for each rod with the updated guess
    % values
    [ RODS.OUT.B1 ] = RK4_rod( RODS.IN.B1 ) ;
    [ RODS.OUT.B2 ] = RK4_rod( RODS.IN.B2 ) ;


    % New resude vector
    resid = Residue( RODS, Fext ) ;

    % In case in which the new residue vector is bigger that the
previous
    % one, the incrementation of the guess values are forced to take
half
    % of value calculated. This operation is applied iteratively until
the
    % new residue vector is smaller that the previous one
    while max(abs(resid)) > max(abs(resid0)) &&
max(abs(delta))>tol*1e-6
        delta = 0.5*delta ;

        RODS.IN.B1.m0     = RODS.IN.B1.m0     - delta(1) ;
        RODS.IN.B1.n0     = RODS.IN.B1.n0     - delta(2:3) ;
        RODS.IN.B1.theta0 = RODS.IN.B1.theta0 - delta(4) ;
        RODS.IN.B2.m0     = RODS.IN.B2.m0     - delta(5) ;
        RODS.IN.B2.n0     = RODS.IN.B2.n0     - delta(6:7) ;
        RODS.IN.B2.theta0 = RODS.IN.B2.theta0 - delta(8) ;

        [ RODS.OUT.B1 ] = RK4_rod( RODS.IN.B1 ) ;
        [ RODS.OUT.B2 ] = RK4_rod( RODS.IN.B2 ) ;
        resid = Residue( RODS, Fext ) ;

    end

end


% If the solution has not been reached or the reside is NaN, the
variable
% "sol" take the value 0
if max(abs(resid)) > tol || not(sum(abs(resid))>=0)
    RODS.OUT.sol = 0 ;
else
    RODS.OUT.sol = 1 ;

    % Moment applied at clamped end
    RODS.OUT.B1.Moment = RODS.IN.B1.m0 ;
    RODS.OUT.B2.Moment = RODS.IN.B2.m0 ;
```

```
%     % Theta0 between 0 and 2*pi
%     while RODS.IN.B1.theta0 >= 2*pi
%         RODS.IN.B1.theta0 = RODS.IN.B1.theta0 - 2*pi;
%     end
%     while RODS.IN.B1.theta0 < 0
%         RODS.IN.B1.theta0 = RODS.IN.B1.theta0 + 2*pi;
%     end
%
%     while RODS.IN.B2.theta0 >= 2*pi
%         RODS.IN.B2.theta0 = RODS.IN.B2.theta0 - 2*pi;
%     end
%     while RODS.IN.B2.theta0 < 0
%         RODS.IN.B2.theta0 = RODS.IN.B2.theta0 + 2*pi;
%     end



end




function [ Jacob ] = Jacobian( RODS )
% Functions that calculates numerically the Jacobian of the residue
vector
% respect to the variables of the problem (guess values)
Jacob = zeros(8) ;


% Increment of the variables (guess values) to calculate numerically
the
% Jacobian
epsilon = 1e-11 ;


% Variables for rod 1
RODS.IN.B1.m0 = RODS.IN.B1.m0 + epsilon ;
OUT_aux = RK4_rod( RODS.IN.B1 ) ;
Jacob([1:3],1) = [ OUT_aux.m(end) ;
                  OUT_aux.p(:,end) ] - ...
                [ RODS.OUT.B1.m(end) ;
                  RODS.OUT.B1.p(:,end) ] ;
Jacob([1:3] ,1) = Jacob([1:3],1)/epsilon ;
RODS.IN.B1.m0 = RODS.IN.B1.m0 - epsilon ;

RODS.IN.B1.n0(1) = RODS.IN.B1.n0(1) + epsilon ;
OUT_aux = RK4_rod( RODS.IN.B1 ) ;
Jacob([1:3],2) = [ OUT_aux.m(end) ;
                  OUT_aux.p(:,end) ] - ...
                [ RODS.OUT.B1.m(end) ;
                  RODS.OUT.B1.p(:,end) ] ;
Jacob([1:3],2) = Jacob([1:3],2)/epsilon ;
RODS.IN.B1.n0(1) = RODS.IN.B1.n0(1) - epsilon ;

RODS.IN.B1.n0(2) = RODS.IN.B1.n0(2) + epsilon ;
OUT_aux = RK4_rod( RODS.IN.B1 ) ;
Jacob(1:3,3) = [ OUT_aux.m(end) ;
                  OUT_aux.p(:,end) ] - ...
```

```
               [ RODS.OUT.B1.m(end) ;
                 RODS.OUT.B1.p(:,end) ] ;
Jacob(1:3,3) = Jacob(1:3,3)/epsilon ;
RODS.IN.B1.n0(2) = RODS.IN.B1.n0(2) - epsilon ;

RODS.IN.B1.theta0 = RODS.IN.B1.theta0 + epsilon ;
OUT_aux = RK4_rod( RODS.IN.B1 ) ;
Jacob(1:3,4) = [ OUT_aux.m(end) ;
                 OUT_aux.p(:,end) ] - ...
               [ RODS.OUT.B1.m(end) ;
                 RODS.OUT.B1.p(:,end) ] ;
Jacob(1:3,4) = Jacob(1:3,4)/epsilon ;
RODS.IN.B1.theta0 = RODS.IN.B1.theta0 - epsilon ;

% Variables for rod 2
RODS.IN.B2.m0 = RODS.IN.B2.m0 + epsilon ;
OUT_aux = RK4_rod( RODS.IN.B2 ) ;
Jacob(4:6,5) = [ OUT_aux.m(end) ;
                 OUT_aux.p(:,end) ] - ...
               [ RODS.OUT.B2.m(end) ;
                 RODS.OUT.B2.p(:,end) ] ;
Jacob(4:6,5) = Jacob(4:6,5)/epsilon ;
RODS.IN.B2.m0 = RODS.IN.B2.m0 - epsilon ;

RODS.IN.B2.n0(1) = RODS.IN.B2.n0(1) + epsilon ;
OUT_aux = RK4_rod( RODS.IN.B2 ) ;
Jacob(4:6,6) = [ OUT_aux.m(end) ;
                 OUT_aux.p(:,end) ] - ...
               [ RODS.OUT.B2.m(end) ;
                 RODS.OUT.B2.p(:,end) ] ;
Jacob(4:6,6) = Jacob(4:6,6)/epsilon ;
RODS.IN.B2.n0(1) = RODS.IN.B2.n0(1) - epsilon ;

RODS.IN.B2.n0(2) = RODS.IN.B2.n0(2) + epsilon ;
OUT_aux = RK4_rod( RODS.IN.B2 ) ;
Jacob(4:6,7) = [ OUT_aux.m(end) ;
                 OUT_aux.p(:,end) ] - ...
               [ RODS.OUT.B2.m(end) ;
                 RODS.OUT.B2.p(:,end) ] ;
Jacob(4:6,7) = Jacob(4:6,7)/epsilon ;
RODS.IN.B2.n0(2) = RODS.IN.B2.n0(2) - epsilon ;

RODS.IN.B2.theta0 = RODS.IN.B2.theta0 + epsilon ;
OUT_aux = RK4_rod( RODS.IN.B2 ) ;
Jacob(4:6,8) = [ OUT_aux.m(end) ;
                 OUT_aux.p(:,end) ] - ...
               [ RODS.OUT.B2.m(end) ;
                 RODS.OUT.B2.p(:,end) ] ;
Jacob(4:6,8) = Jacob(4:6,8)/epsilon ;
RODS.IN.B2.theta0 = RODS.IN.B2.theta0 - epsilon ;


% Unit elements
Jacob(7,[2 6]) = 1 ;
Jacob(8,[3 7]) = 1 ;
```

```
function [ resid ] = Residue( RODS, Fext )
% Function of the residue vector of the whole mechanism

resid = [ RODS.OUT.B1.m(end) ;
          RODS.OUT.B1.p(:,end) - RODS.IN.B1.pL ;
          RODS.OUT.B2.m(end) ;
          RODS.OUT.B2.p(:,end) - RODS.IN.B2.pL ;
          RODS.IN.B1.n0 + RODS.IN.B2.n0 - Fext ] ;




function [ OUT ] = RK4_rod ( IN )
% Functions that solves de initial value problem using the method of
% Runge-Kutta of order four.
%   IN:  Structure in which are geometric and mechanical parameters
%       IN.L        Length of the rod [m]
%       IN.N        Number of nodes of the rod
%       IN.EI       Bending stiffness of the rod [Pa]
%       IN.d        Distance of the rigid part located in pinned end
of the
%                   rod [m]
%       IN.alpha    Angle of the guide [rad]
%       In.theta0   Angle of the rod at clamped end [rad]
%       IN.lambda   Value of lambda [m]
%       IN.m0       Bending at the clamped end [N.m]
%       IN.n0       Internal force at the clamped end (2x1 vector) [N]

%   OUT:    Structure in which are saved the solution
%       OUT.p       array (2xIN.N) with the x,y component of the
%                   centroid position of the rod
%       OUT.m       array (1xIN.N) with the bending moment in each
node
%                   of the rod
%       OUT.theta   array (1xIN.N) with the angle of the tangent of
%                   the rod
%       OUT.Ener    Elastic deformation enegy of the rod using the
Simpson
%                   rule


% Increment of length
ds = IN.L/(IN.N-1) ;

%%%%%%%%%%%%%%%%%%%%% Variables to save the solution
%%%%%%%%%%%%%%%%%%%%%
OUT.p     = zeros(2,IN.N) ; % Centroid position vector function (x,y)
OUT.theta = zeros(1,IN.N) ;   % Angle function of the tangent of the
rod
OUT.m     = zeros(1,IN.N) ; % Bending moment function

% Variables at the first point (clamped end)
OUT.p(:,1)    = IN.OA + IN.lambda*[cos(IN.theta0); sin(IN.theta0)];
OUT.theta(1) = IN.theta0 ;
OUT.m(1)     = IN.m0 ;

% Vector with the values of the dependent variables at the clamped end
var = [ OUT.p(:,1) ;
        OUT.theta(1) ;
```

```
        OUT.m(1) ] ;

% "For" loop to integrate the system of differential equations through
the
% rod
for ii = 2:IN.N

    % Runge-Kutta method of order four
    k1 = ds *  Right_function( IN.EI, IN.n0, var ) ;
    k2 = ds *  Right_function( IN.EI, IN.n0, var + 0.5*k1 ) ;
    k3 = ds *  Right_function( IN.EI, IN.n0, var + 0.5*k2 ) ;
    k4 = ds *  Right_function( IN.EI, IN.n0, var + k3 ) ;

    var = var + (k1 + 2*k2 + 2*k3 + k4)/6 ;

    OUT.p(:,ii)    = var(1:2) ;
    OUT.theta(ii) = var(3) ;
    OUT.m(ii)      = var(4) ;

end



% Vector product of pinned end position and force applied on it
OUT.pxn = OUT.p(1,end)*IN.n0(2)-OUT.p(2,end)*IN.n0(1) ;


% Elastic energy calculated throught the the Simpson's rule
m2 = OUT.m(1:end-1).^2 ;         % Square of the internal moemnt
funciton

OUT.Ener = ds/3 * ( m2(1) + m2(end) + 4*sum(m2(2:2:(IN.N-1))) + ...
           2*sum(m2(3:2:(IN.N-2))) ) ;
OUT.Ener = OUT.Ener/IN.EI ;




function [ func ] = Right_function( EI, n, var )
% Value of the right part of the system of differential equations
% var(1)     px
% var(2)     py
% var(3)     theta
% var(4)     moment

func = [ cos(var(3)) ;
         sin(var(3)) ;
         var(4)/EI ;
         n(1)*sin(var(3)) - n(2)*cos(var(3)) ] ;


function [x0,y0,iout,jout] = intersections(x1,y1,x2,y2,robust)
%INTERSECTIONS Intersections of curves.
%   Computes the (x,y) locations where two curves intersect.  The
curves
%   can be broken with NaNs or have vertical segments.
%
% Example:
```

```
%   [X0,Y0] = intersections(X1,Y1,X2,Y2,ROBUST);
%
% where X1 and Y1 are equal-length vectors of at least two points and
% represent curve 1.  Similarly, X2 and Y2 represent curve 2.
% X0 and Y0 are column vectors containing the points at which the two
% curves intersect.
%
% ROBUST (optional) set to 1 or true means to use a slight variation
of the
% algorithm that might return duplicates of some intersection points,
and
% then remove those duplicates.  The default is true, but since the
% algorithm is slightly slower you can set it to false if you know
that
% your curves don't intersect at any segment boundaries.  Also, the
robust
% version properly handles parallel and overlapping segments.
%
% The algorithm can return two additional vectors that indicate which
% segment pairs contain intersections and where they are:
%
%   [X0,Y0,I,J] = intersections(X1,Y1,X2,Y2,ROBUST);
%
% For each element of the vector I, I(k) = (segment number of (X1,Y1))
+
% (how far along this segment the intersection is).  For example, if
I(k) =
% 45.25 then the intersection lies a quarter of the way between the
line
% segment connecting (X1(45),Y1(45)) and (X1(46),Y1(46)).  Similarly
for
% the vector J and the segments in (X2,Y2).
%
% You can also get intersections of a curve with itself.  Simply pass
in
% only one curve, i.e.,
%
%   [X0,Y0] = intersections(X1,Y1,ROBUST);
%
% where, as before, ROBUST is optional.

% Version: 2.0, 25 May 2017
% Author:  Douglas M. Schwarz
% Email:   dmschwarz=ieee*org, dmschwarz=urgrad*rochester*edu
% Real_email = regexprep(Email,{'=','*'},{'@','.'})



% Theory of operation:
%
% Given two line segments, L1 and L2,
%
%   L1 endpoints:  (x1(1),y1(1)) and (x1(2),y1(2))
%   L2 endpoints:  (x2(1),y2(1)) and (x2(2),y2(2))
%
% we can write four equations with four unknowns and then solve them.
The
% four unknowns are t1, t2, x0 and y0, where (x0,y0) is the
intersection of
% L1 and L2, t1 is the distance from the starting point of L1 to the
% intersection relative to the length of L1 and t2 is the distance
from the
```

```
% starting point of L2 to the intersection relative to the length of
L2.
%
% So, the four equations are
%
%    (x1(2) - x1(1))*t1 = x0 - x1(1)
%    (x2(2) - x2(1))*t2 = x0 - x2(1)
%    (y1(2) - y1(1))*t1 = y0 - y1(1)
%    (y2(2) - y2(1))*t2 = y0 - y2(1)
%
% Rearranging and writing in matrix form,
%
%  [x1(2)-x1(1)       0       -1   0;      [t1;      [-x1(1);
%        0       x2(2)-x2(1)  -1   0;   *   t2;   =   -x2(1);
%   y1(2)-y1(1)       0        0  -1;       x0;       -y1(1);
%        0       y2(2)-y2(1)   0  -1]       y0]       -y2(1)]
%
% Let's call that A*T = B.  We can solve for T with T = A\B.
%
% Once we have our solution we just have to look at t1 and t2 to
determine
% whether L1 and L2 intersect.  If 0 <= t1 < 1 and 0 <= t2 < 1 then
the two
% line segments cross and we can include (x0,y0) in the output.
%
% In principle, we have to perform this computation on every pair of
line
% segments in the input data.  This can be quite a large number of
pairs so
% we will reduce it by doing a simple preliminary check to eliminate
line
% segment pairs that could not possibly cross.  The check is to look
at the
% smallest enclosing rectangles (with sides parallel to the axes) for
each
% line segment pair and see if they overlap.  If they do then we have
to
% compute t1 and t2 (via the A\B computation) to see if the line
segments
% cross, but if they don't then the line segments cannot cross.  In a
% typical application, this technique will eliminate most of the
potential
% line segment pairs.


% Input checks.
if verLessThan('matlab','7.13')
    error(nargchk(2,5,nargin)) %#ok<NCHKN>
else
    narginchk(2,5)
end

% Adjustments based on number of arguments.
switch nargin
    case 2
        robust = true;
        x2 = x1;
        y2 = y1;
        self_intersect = true;
    case 3
        robust = x2;
```

```
        x2 = x1;
        y2 = y1;
        self_intersect = true;
    case 4
        robust = true;
        self_intersect = false;
    case 5
        self_intersect = false;
end

% x1 and y1 must be vectors with same number of points (at least 2).
if sum(size(x1) > 1) ~= 1 || sum(size(y1) > 1) ~= 1 || ...
        length(x1) ~= length(y1)
    error('X1 and Y1 must be equal-length vectors of at least 2
points.')
end
% x2 and y2 must be vectors with same number of points (at least 2).
if sum(size(x2) > 1) ~= 1 || sum(size(y2) > 1) ~= 1 || ...
        length(x2) ~= length(y2)
    error('X2 and Y2 must be equal-length vectors of at least 2
points.')
end


% Force all inputs to be column vectors.
x1 = x1(:);
y1 = y1(:);
x2 = x2(:);
y2 = y2(:);

% Compute number of line segments in each curve and some differences
we'll
% need later.
n1 = length(x1) - 1;
n2 = length(x2) - 1;
xy1 = [x1 y1];
xy2 = [x2 y2];
dxy1 = diff(xy1);
dxy2 = diff(xy2);


% Determine the combinations of i and j where the rectangle enclosing
the
% i'th line segment of curve 1 overlaps with the rectangle enclosing
the
% j'th line segment of curve 2.

% Original method that works in old MATLAB versions, but is slower
than
% using binary singleton expansion (explicit or implicit).
% [i,j] = find( ...
%   repmat(mvmin(x1),1,n2) <= repmat(mvmax(x2).',n1,1) & ...
%   repmat(mvmax(x1),1,n2) >= repmat(mvmin(x2).',n1,1) & ...
%   repmat(mvmin(y1),1,n2) <= repmat(mvmax(y2).',n1,1) & ...
%   repmat(mvmax(y1),1,n2) >= repmat(mvmin(y2).',n1,1));

% Select an algorithm based on MATLAB version and number of line
% segments in each curve.  We want to avoid forming large matrices for
% large numbers of line segments.  If the matrices are not too large,
% choose the best method available for the MATLAB version.
```

```
if n1 > 1000 || n2 > 1000 || verLessThan('matlab','7.4')
    % Determine which curve has the most line segments.
    if n1 >= n2
        % Curve 1 has more segments, loop over segments of curve 2.
        ijc = cell(1,n2);
        min_x1 = mvmin(x1);
        max_x1 = mvmax(x1);
        min_y1 = mvmin(y1);
        max_y1 = mvmax(y1);
        for k = 1:n2
            k1 = k + 1;
            ijc{k} = find( ...
                min_x1 <= max(x2(k),x2(k1)) & max_x1 >=
min(x2(k),x2(k1)) & ...
                min_y1 <= max(y2(k),y2(k1)) & max_y1 >=
min(y2(k),y2(k1)));
            ijc{k}(:,2) = k;
        end
        ij = vertcat(ijc{:});
        i = ij(:,1);
        j = ij(:,2);
    else
        % Curve 2 has more segments, loop over segments of curve 1.
        ijc = cell(1,n1);
        min_x2 = mvmin(x2);
        max_x2 = mvmax(x2);
        min_y2 = mvmin(y2);
        max_y2 = mvmax(y2);
        for k = 1:n1
            k1 = k + 1;
            ijc{k}(:,2) = find( ...
                min_x2 <= max(x1(k),x1(k1)) & max_x2 >=
min(x1(k),x1(k1)) & ...
                min_y2 <= max(y1(k),y1(k1)) & max_y2 >=
min(y1(k),y1(k1)));
            ijc{k}(:,1) = k;
        end
        ij = vertcat(ijc{:});
        i = ij(:,1);
        j = ij(:,2);
    end

elseif verLessThan('matlab','9.1')
    % Use bsxfun.
    [i,j] = find( ...
        bsxfun(@le,mvmin(x1),mvmax(x2).') & ...
        bsxfun(@ge,mvmax(x1),mvmin(x2).') & ...
        bsxfun(@le,mvmin(y1),mvmax(y2).') & ...
        bsxfun(@ge,mvmax(y1),mvmin(y2).'));

else
    % Use implicit expansion.
    [i,j] = find( ...
        mvmin(x1) <= mvmax(x2).' & mvmax(x1) >= mvmin(x2).' & ...
        mvmin(y1) <= mvmax(y2).' & mvmax(y1) >= mvmin(y2).');

end
```

```
% Find segments pairs which have at least one vertex = NaN and remove
them.
% This line is a fast way of finding such segment pairs.  We take
% advantage of the fact that NaNs propagate through calculations, in
% particular subtraction (in the calculation of dxy1 and dxy2, which
we
% need anyway) and addition.
% At the same time we can remove redundant combinations of i and j in
the
% case of finding intersections of a line with itself.
if self_intersect
    remove = isnan(sum(dxy1(i,:) + dxy2(j,:),2)) | j <= i + 1;
else
    remove = isnan(sum(dxy1(i,:) + dxy2(j,:),2));
end
i(remove) = [];
j(remove) = [];

% Initialize matrices.  We'll put the T's and B's in matrices and use
them
% one column at a time.  AA is a 3-D extension of A where we'll use
one
% plane at a time.
n = length(i);
T = zeros(4,n);
AA = zeros(4,4,n);
AA([1 2],3,:) = -1;
AA([3 4],4,:) = -1;
AA([1 3],1,:) = dxy1(i,:).';
AA([2 4],2,:) = dxy2(j,:).';
B = -[x1(i) x2(j) y1(i) y2(j)].';

% Loop through possibilities.  Trap singularity warning and then use
% lastwarn to see if that plane of AA is near singular.  Process any
such
% segment pairs to determine if they are colinear (overlap) or merely
% parallel.  That test consists of checking to see if one of the
endpoints
% of the curve 2 segment lies on the curve 1 segment.  This is done by
% checking the cross product
%
%   (x1(2),y1(2)) - (x1(1),y1(1)) x (x2(2),y2(2)) - (x1(1),y1(1)).
%
% If this is close to zero then the segments overlap.

% If the robust option is false then we assume no two segment pairs
are
% parallel and just go ahead and do the computation.  If A is ever
singular
% a warning will appear.  This is faster and obviously you should use
it
% only when you know you will never have overlapping or parallel
segment
% pairs.

if robust
    overlap = false(n,1);
    warning_state = warning('off','MATLAB:singularMatrix');
    % Use try-catch to guarantee original warning state is restored.
    try
        lastwarn('')
```

```
        for k = 1:n
            T(:,k) = AA(:,:,k)\B(:,k);
            [unused,last_warn] = lastwarn; %#ok<ASGLU>
            lastwarn('')
            if strcmp(last_warn,'MATLAB:singularMatrix')
                % Force in_range(k) to be false.
                T(1,k) = NaN;
                % Determine if these segments overlap or are just
parallel.
                overlap(k) = rcond([dxy1(i(k),:);xy2(j(k),:) -
xy1(i(k),:)]) < eps;
            end
        end
        warning(warning_state)
    catch err
        warning(warning_state)
        rethrow(err)
    end
    % Find where t1 and t2 are between 0 and 1 and return the
corresponding
    % x0 and y0 values.
    in_range = (T(1,:) >= 0 & T(2,:) >= 0 & T(1,:) <= 1 & T(2,:) <=
1).';
    % For overlapping segment pairs the algorithm will return an
    % intersection point that is at the center of the overlapping
region.
    if any(overlap)
        ia = i(overlap);
        ja = j(overlap);
        % set x0 and y0 to middle of overlapping region.
        T(3,overlap) = (max(min(x1(ia),x1(ia+1)),min(x2(ja),x2(ja+1)))
+ ...
            min(max(x1(ia),x1(ia+1)),max(x2(ja),x2(ja+1)))).'/2;
        T(4,overlap) = (max(min(y1(ia),y1(ia+1)),min(y2(ja),y2(ja+1)))
+ ...
            min(max(y1(ia),y1(ia+1)),max(y2(ja),y2(ja+1)))).'/2;
        selected = in_range | overlap;
    else
        selected = in_range;
    end
    xy0 = T(3:4,selected).';

    % Remove duplicate intersection points.
    [xy0,index] = unique(xy0,'rows');
    x0 = xy0(:,1);
    y0 = xy0(:,2);

    % Compute how far along each line segment the intersections are.
    if nargout > 2
        sel_index = find(selected);
        sel = sel_index(index);
        iout = i(sel) + T(1,sel).';
        jout = j(sel) + T(2,sel).';
    end
else % non-robust option
    for k = 1:n
        [L,U] = lu(AA(:,:,k));
        T(:,k) = U\(L\B(:,k));
    end
```

```
    % Find where t1 and t2 are between 0 and 1 and return the
corresponding
    % x0 and y0 values.
    in_range = (T(1,:) >= 0 & T(2,:) >= 0 & T(1,:) < 1 & T(2,:) <
1).';
    x0 = T(3,in_range).';
    y0 = T(4,in_range).';

    % Compute how far along each line segment the intersections are.
    if nargout > 2
        iout = i(in_range) + T(1,in_range).';
        jout = j(in_range) + T(2,in_range).';
    end
end

% Plot the results (useful for debugging).
% plot(x1,y1,x2,y2,x0,y0,'ok');

function y = mvmin(x)
% Faster implementation of movmin(x,k) when k = 1.
y = min(x(1:end-1),x(2:end));

function y = mvmax(x)
% Faster implementation of movmax(x,k) when k = 1.
y = max(x(1:end-1),x(2:end));


function [ h, puntos ] = isocurve3(
X,Y,Z,V1,V2,isovalue1,isovalue2,varargin)
% h = isocurve3( X,Y,Z,V1,V2,isovalue1,isovalue2)
% Plot intersection curve between isosurfaces
% X,Y,Z,V1,isovalue1 and X,Y,Z,V2,isovalue2 are input data for
isosurface
% Additional arguments are passed to plot3
% h - axes handle

% Leif Persson, Mathematics department, Umeå University, March 2016

diagnostic_mode = false;

%% Compute first isosurface and faces crossing second isosurface
h=isosurface(X,Y,Z,V1,isovalue1);
v=h.vertices; % Vertex coordinates; v(i_v,:) are the x,y,z-coordinates
of vertex i_v
f2v=h.faces; % Faces of isosurface 1; f(i_f,:) are the vertex numbers
of face i_f
V21 = interp3(X,Y,Z,V2,v(:,1),v(:,2),v(:,3)); % Interpolated V2 values
on vertices of isosurface 1
s_f=V21(f2v)-isovalue2; % Second isosurface sign on faces of first
surface
is_cf=(s_f(:,1).*s_f(:,2)<=0)|(s_f(:,2).*s_f(:,3)<=0)|(s_f(:,3).*s_f(:
,1)<0); % boolean vector for crossing faces
f2v=f2v(is_cf,:); % Restrict to crossing faces
n_v=size(v,1); % Number of vertices
n_f=size(f2v,1); % Number of faces
s_f = s_f(is_cf,:);

%% Restrict to vertices of crossing faces, renumber vertices
old_i_v = unique(f2v);
v = v(old_i_v, :);
```

```
V21 = V21(old_i_v);
n_new_v = length(old_i_v);
v2new_v=zeros(size(1,n_v));
v2new_v(old_i_v) = 1:n_new_v; % Translation vector
for i_f=1:n_f,
    f2v(i_f,1)=v2new_v(f2v(i_f,1));
    f2v(i_f,2)=v2new_v(f2v(i_f,2));
    f2v(i_f,3)=v2new_v(f2v(i_f,3));
end
n_v = size(v,1);

%% Alternating edge to vertex mapping
e2v = [];
for i_f=1:n_f,
    if s_f(i_f,1)*s_f(i_f,2)<=0
        e2v = [ e2v; f2v(i_f,[1,2]) ];
    end
    if s_f(i_f,2)*s_f(i_f,3)<=0
        e2v = [ e2v; f2v(i_f,[2,3]) ];
    end
    if s_f(i_f,3)*s_f(i_f,1)<=0
        e2v = [ e2v; f2v(i_f,[3,1]) ];
    end
end
e2v = unique(sort(e2v,2),'rows');
n_e = size(e2v,1);

%% Face to alternating edge mapping
v2e = sparse([e2v(:,1);e2v(:,2)],[e2v(:,2);e2v(:,1)],[(1:n_e)';
(1:n_e)'],n_v,n_v);
f2e = zeros(n_f,2);
for i_f=1:n_f,
    k=1;
    i_e = v2e(f2v(i_f,1),f2v(i_f,2));
    if i_e > 0,
        f2e(i_f, k) = i_e; k=k+1;
    end
    i_e = v2e(f2v(i_f,2),f2v(i_f,3));
    if i_e > 0,
        f2e(i_f, k) = i_e; k=k+1;
    end
    i_e = v2e(f2v(i_f,3),f2v(i_f,1));
    if i_e > 0,
        f2e(i_f, k) = i_e; k=k+1;
    end
end

%% Alternating edge to face mapping
e2f = zeros(n_e,2);
for i_f=1:n_f,
    i_e = f2e(i_f,1);
    if i_e>0,
        if e2f(i_e,1)==0,
            e2f(i_e,1)=i_f;
        elseif e2f(i_e,2)==0
            e2f(i_e,2)=i_f;
        else
            error('Too many faces for edge');
        end
    end
    i_e=f2e(i_f,2);
```

```
    if i_e>0,
        if e2f(i_e,1)==0,
            e2f(i_e,1)=i_f;
        elseif e2f(i_e,2)==0
            e2f(i_e,2)=i_f;
        else
            error('Too many faces for edge');
        end
    end
end

%% Build alternating edge-to-edge mapping
e2e = zeros(n_e,2);
for i_e=1:n_e,
    j_f=e2f(i_e,:);
    j_f=j_f(j_f>0); % The faces neighboring edge i_e (may be one or
two)
    k_e=f2e(j_f,:); % The edges of those faces
    k_e=reshape(k_e,1,[]);
    k_e=k_e((k_e>0)&(k_e~=i_e)); % The neighboring edges of i_e
    e2e(i_e,1:length(k_e))=k_e;
end
if size(e2e,2)>2,
    error('Incompatible edge-to-edge mapping');
end

%% Compute crossings on alternating edges
tmp=abs(V21(e2v)-isovalue2);
c = zeros(n_e,3);
c(:,1) =
(v(e2v(:,1),1).*tmp(:,2)+v(e2v(:,2),1).*tmp(:,1))./(tmp(:,1)+tmp(:,2))
;
c(:,2) =
(v(e2v(:,1),2).*tmp(:,2)+v(e2v(:,2),2).*tmp(:,1))./(tmp(:,1)+tmp(:,2))
;
c(:,3) =
(v(e2v(:,1),3).*tmp(:,2)+v(e2v(:,2),3).*tmp(:,1))./(tmp(:,1)+tmp(:,2))
;

%% Build adjacency matrix
A = sparse(n_e,n_e);
for i_e=1:n_e,
    tmp = e2e(i_e,:);
    A(i_e, tmp(tmp>0))=1;
end
if any(diag(A)~=0) || any(any(A~=A')),
    error('Incompatible adjacency matrix');
end
%% Compute parts (connected components)
p = []; % Boolean matrix; each row represents nodes of one component
r = ones(1,n_e); % Boolean vector for remaining nodes
while any(r>0)
    x=zeros(1,n_e);
    i_e=find(r,1);
    x(i_e)=1; % Start with first remaining edge
    tmp=x*(A+speye(size(A)))'; % Add nodes
    tmp=(tmp>0);
    while any(tmp~=x)
        x=tmp;
        tmp=x*(A+speye(size(A)))';
        tmp=(tmp>0);
```

```
    end
    p=[p;x];
    r = ~any([p; p]>0);
end
p=(p>0);
n_p = size(p,1); % Number of parts (connected components)

%% Part edge numbers (unsorted)
p2e = zeros(size(p));
for i_p=1:n_p,
    tmp=1:n_e;
    tmp = tmp(p(i_p,:));
    p2e(i_p,1:length(tmp))=tmp;
end
n_ep = zeros(1,n_p);
%% Sort edges in each part
sorted_p2e = zeros(size(p,1),size(p,2)+1); % +1 needed for closed
curves
for i_p=1:n_p,
    ep=p2e(i_p,:);
    ep=ep(ep>0); % Edge numbers in part
    n_ep(i_p) = length(ep);
    is_open_curve = any(e2f(ep,1)<=0 | e2f(ep,2)<=0);
    if is_open_curve
%         fprintf('Part %d is open\n', i_p);
        i_ep = find(e2f(ep,1)<=0 | e2f(ep,2)<=0);
        ep_start=ep(i_ep(1));
        ep_end  =ep(i_ep(2));
    else                                  % Closed curve
%       fprintf('Part %d is closed\n', i_p);
        n_ep(i_p)=n_ep(i_p)+1;
        ep_start=ep(1);
        ep_end  =ep(1);
    end
    sorted_ep = zeros(1,n_ep(i_p)+1); % +1 needed if closed curve
    is_remaining_ep = true(1,n_e); % Boolean vector indicating
remaining edges
    sorted_ep(1) = ep_start;
    is_remaining_ep(ep_start)=false;
    for i_ep=2:n_ep(i_p)-1,
        j_f = e2f(sorted_ep(i_ep-1),:);
        j_f = j_f(j_f>0);
        k_e = f2e(j_f,:);
        k_e = reshape(k_e,1,[]);
        k_e = k_e.*double(is_remaining_ep(k_e));
        k_e = k_e(k_e>0);
        if numel(k_e)==0
            error('Sorting problem...');
        end
        sorted_ep(i_ep)=k_e(1);
        is_remaining_ep(k_e(1))=false;
    end
    i_ep = n_ep(i_p);
    sorted_ep(i_ep) = ep_end;
    sorted_p2e(i_p,1:numel(sorted_ep))=sorted_ep;
end
p2e=sorted_p2e;

%% Plot curves
puntos = [] ;
```

```
for i_p=1:n_p,

    cx = c(p2e(i_p,1:n_ep(i_p)),1) ;
    cy = c(p2e(i_p,1:n_ep(i_p)),2) ;
    cz = c(p2e(i_p,1:n_ep(i_p)),3) ;

    puntos = [ puntos;
               cx cy cz ;
               NaN NaN NaN ] ;


plot3(c(p2e(i_p,1:n_ep(i_p)),1),c(p2e(i_p,1:n_ep(i_p)),2),c(p2e(i_p,1:
n_ep(i_p)),3),varargin{:});
    hold on;
end
h = gca; % Return handle to current axis

%% Diagnostic plot for edge-face mappings
if diagnostic_mode,
    f_list=containers.Map('KeyType', 'double', 'ValueType', 'double');
    figure;
    hold on;
    view(3);
    i_f=1; f_list(i_f)=0; % Store i_f in face list
    plot_face(i_f, 'LineWidth', 2, 'Color', 'blue','LineStyle', ':');
    i_e=f2e(i_f,:); % Alternating edges of face i_f
    plot_edge(i_e(1), 'LineWidth', 2, 'Color', 'red');
    plot_edge(i_e(2), 'LineWidth', 2, 'Color', 'red');
    j_f=unique(e2f(i_e,:));
    j_f=j_f(j_f>0);
    while any(~isKey(f_list, num2cell(j_f)))
        for j=1:length(j_f),
            if ~isKey(f_list, j_f(j)) break; end
        end
        i_f=j_f(j); f_list(i_f)=0;
        plot_face(i_f, 'LineWidth', 2, 'Color', 'blue','LineStyle',
':');
        i_e=f2e(i_f,:); % Alternating edges of face i_f
        plot_edge(i_e(1), 'LineWidth', 2, 'Color', 'red');
        plot_edge(i_e(2), 'LineWidth', 2, 'Color', 'red');
        j_f=unique(e2f(i_e,:));
        j_f=j_f(j_f>0);
    end
    for i_p=1:n_p,

plot3(c(p(i_p,:)',1),c(p(i_p,:)',2),c(p(i_p,:)',3),'o','MarkerSize',
10, 'MarkerFaceColor', 'black');
    end
    %plot3(c(p(2,:)',1),c(p(2,:)',2),c(p(2,:)',3),'d','MarkerSize',
10, 'MarkerFaceColor', 'black');
end
%% Auxiliary functions for diagnostic plot

    function plot_face(i_f,varargin)
        ii_e=f2e(i_f,:); % Edges of face i_f
        ii_v=e2v(ii_e,:); % Vertex numbers of edges
        ii_v=unique(ii_v); % Unique vertex numbers
        v_f=v(ii_v,:);
        line(v_f([1,2],1),v_f([1,2],2),v_f([1,2],3),varargin{:});
        line(v_f([2,3],1),v_f([2,3],2),v_f([2,3],3),varargin{:});
```

```
        line(v_f([3,1],1),v_f([3,1],2),v_f([3,1],3),varargin{:});
    end

    function plot_edge(i_e,varargin)
        ii_v=e2v(i_e,:); % Vertex numbers of edge
        v_f=v(ii_v,:);   % Vertex coordinates
        line(v_f([1,2],1),v_f([1,2],2),v_f([1,2],3),varargin{:});
        text(0.5*sum(v_f([1,2],1)), 0.5*sum(v_f([1,2],2)),
0.5*sum(v_f([1,2],3)), num2str(i_e));
    end
end

function [ x, y, z ] = isoline_intersection( X, Y, Z, V1, V2, V12 )

iter = 0 ;

x = [] ;
y = [] ;
z = [] ;

% Isolines
if max(max(max(V1)))*min(min(min(V1))) < 0
% %       [ H1, puntos1 ] = isocurve3( X, Y, Z, V12, V1, 0, 0, ...
% %                                  'LineWidth', 2, 'Color',
'blue');
    [ H1, puntos1 ] = isocurve3( X, Y, Z, V1, V12, 0, 0, ...
                                 'LineWidth', 2, 'Color', 'blue');
    iter = iter + 1 ;
end
if max(max(max(V2)))*min(min(min(V2))) < 0
    [ H2, puntos2 ] = isocurve3( X, Y, Z, V12, V2, 0, 0, ...
                                 'LineWidth', 2, 'Color', 'red');

% %       [ H2, puntos2 ] = isocurve3( X, Y, Z, V2, V12, 0, 0, ...
% %                                  'LineWidth', 2, 'Color',
'red');

    iter = iter + 1 ;
end


% Intersection of isolines (solution points)
if iter == 2
    aux_x = puntos1   ;
    aux_y = puntos2   ;

    z = [] ;

    [x, y, iout, jout ] = intersections( aux_x(:,1), aux_x(:,2), ...
                                         aux_y(:,1), aux_y(:,2) ) ;

    if numel(x) > 0
        pos_x(1:3:3*numel(iout)) = floor(iout) ;
        pos_x(2:3:3*numel(iout)) = ceil(iout) ;
        pos_x(3:3:3*numel(iout)) = size(aux_x,1) ;

        pos_y(1:3:3*numel(iout)) = floor(jout) ;
        pos_y(2:3:3*numel(iout)) = ceil(jout) ;
        pos_y(3:3:3*numel(iout)) = size(aux_y,1) ;
```

```
        aux_x = aux_x(pos_x',:) ;
        aux_y = aux_y(pos_y',:) ;

% %         aux_1 = aux_x(2:3:3*numel(iout),3) - 
aux_x(1:3:3*numel(iout),3) ;
% %         aux_1 = aux_1./(aux_x(2:3:3*numel(iout),2) - 
aux_x(1:3:3*numel(iout),2)) ;
% %         aux_2 = aux_y(2:3:3*numel(iout),3) - 
aux_y(1:3:3*numel(iout),3) ;
% %         aux_2 = aux_2./(aux_y(2:3:3*numel(iout),2) - 
aux_y(1:3:3*numel(iout),2)) ;
% %         aux = abs( aux_1./aux_2 - 1 ) ;

%         aux_x(1:3:3*numel(iout),3) = aux_x(1:3:3*numel(iout),3) + 
1e-4 ;
%         aux_x(2:3:3*numel(iout),3) = aux_x(2:3:3*numel(iout),3) - 
1e-4 ;


        [x, z, i_xz, j_xz ] = intersections( aux_x(:,1), aux_x(:,3), 
...
                                           aux_y(:,1), aux_y(:,3) 
) ;
% %         [y, z_y, i_yz, j_yz ] = intersections( aux_x(:,2), 
aux_x(:,3), ...
% %                                               aux_y(:,2), 
aux_y(:,3) ) ;
% %
% %         matrix = i_xz*ones(1,numel(i_yz)) - ...
% %                  ones(numel(i_xz),1)*i_yz' ;
% %         pos = find(abs(matrix)<1) ;
% %
% %         if not(numel(z_x)==numel(z_y))
% %             pos1 = floor(i_xz) ;
% %             pos2 = floor(pos1/3)+1 ;
% %
% %             for kk = 1:numel(z_x)
% %                 if aux(pos2(kk)) < 1e-3
% %                     y(kk) = aux_x(pos1(kk),2)*(1-
i_xz(kk)+pos1(kk)) + ...
% %                         aux_x(pos1(kk)+1,2) ;
% %                     z(kk) = aux_x(pos1(kk),3)*(1-
i_xz(kk)+pos1(kk)) + ...
% %                         aux_x(pos1(kk)+1,3) ;
% %                 end
% %             end
% %
% %         end
% %
% %         [II,JJ] = ind2sub(size(matrix), pos) ;
% %
% %         x = x(II) ;
% %         y = y(JJ) ;
% %         z = z_y(JJ) ;

        pos0 = floor(i_xz) ;
        y = aux_x(pos0,2) + (aux_x(pos0+1,2) - aux_x(pos0,2)).*(i_xz-
pos0) ;
```

```
        plot3(x, y, z, 'og', 'MarkerSize', 12, 'MarkerFaceColor', 'g'
) ;
    end
end


end

function [ A, B ] = p2AB( k )

tol = 1e-6 ;

B = linspace(-100,0,1e3+1) ;

res = exp(B).*(B-k)+k ;

aux = res(1:end-1).*res(2:end) ;

ii = find(aux<0) ;

B1 = B(ii) ;          res1 = res(ii) ;
B2 = B(ii+1) ;        res2 = res(ii+1) ;

while abs(res2) > tol

    B3 = B2 - res2/(res2-res1)*(B2-B1) ;
    res3 = exp(B3).*(B3-k)+k ;

    B1 = B2 ;
    B2 = B3 ;

    res1 = res2 ;
    res2 = res3 ;

end

B = B2 ;
A = 1/(exp(B)-1) ;

% % A = (1-1e-6)*A ;

end
```

3.4. INTEGRALES ELÍPTICAS. PROBLEMA DIRECTO

```
%%%%%%%%%%%%%%%%%% Program to Obtain all solutions of the FK problem
%%%%%%
%%%%%%%%%%%%%%%%%% 2 DOF closed-loop mechanisms with 2 flexible rods
%%%%%%
%%%%%%%%%%%%%%%%%% 2 rods clamped-hinged

%%%%%%%%%%%%%%%%%% Initialization:
clc; clear; close all;
addpath('EllipticIntegrals_functions') ;
% addpath('C:\Users\Javier Pérez\Desktop\Proyecto
TFG\2R_FlexClampedPinned\EllipticIntegrals_functions');
%%%%%%%%%%%%%%%%%%% MECHANICAL AND GEOMETRIC PROPERTIES
%%%%%%%%%%%%%%%%%%%
% Rod length
L = 1 ;
% Cross-section properties
diam = 1.5e-3  ;
r = 0.5*diam ;
I = 0.25*pi*r^4 ;
% Young's modulus
E = 210e9 ;

% Load at end-effector
FP = [0.3; -0.1] ;


%%%%%%%%%%%%%%%%% POSITION AND ORIENTATION OF THE ACTUATORS
%%%%%%%%%%%%%%%%%
% Pose of local frame at clamped-end defined by inputs

% Positions of the actuators
x1_clamped = - 0.3 ;
y1_clamped = 0 ;

x2_clamped = 0.3 ;
y2_clamped = 0 ;

% Angles of the actuators
theta1 = 40/180*pi ;
theta2 = 60/180*pi ;

% Mode of deflexion to be considered in the solution
mode1 = 1 ;
mode2 = 1 ;

%%%%%%%%%%%%%%%%%%%%%%%%% CREATION OF THE STRUCTURE
%%%%%%%%%%%%%%%%%%%%%%%%%

IN.B1.L          = L ;
IN.B1.EI         = E*I ;
IN.B1.mode       = mode1 ;
IN.B1.x0         = x1_clamped ;
IN.B1.y0         = y1_clamped ;
IN.B1.lambda     = 0 ; % preguntar a Diego
IN.B1.theta      = theta1 ;
IN.B1.FP         = FP ;

IN.B2.L          = L ;
```

```
IN.B2.EI        = E*I ;
IN.B2.mode      = mode2 ;
IN.B2.x0        = x2_clamped ;
IN.B2.y0        = y2_clamped ;
IN.B2.lambda    = 0 ; % preguntar a Diego
IN.B2.theta     = theta2 ;
IN.B2.FP        = FP ;



%%%%%%%%%%%%%%%%%%%% DISCRETIZATION IN PSI1 KR1 AND KR2
%%%%%%%%%%%%%%%%%%%

nt= 21; % partes en KR positivo/negativo
npsi=30; % partes en psi1

% Discretization of the kr axis with an exponential relationship
p1 = 0.25 ; % pendiente de la exponencial en el extremo
[ A, B ] = p2AB( p1 ) ;

t = linspace(0,1,nt) ;
y = A*(exp(B*t)-1) ;
y(1) = 1e-6 ;
y(end) = 1-1e-6 ;

% Discretization of psi1 axis with equidistant values
psi1_range      = linspace( 0, 2*pi, npsi ) ;
psi1_range(1)   = psi1_range(1) + 1e-3 ;
psi1_range(end) = psi1_range(end) - 1e-3 ;







%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% CALULATION
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
tic ;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% apertura de la figura de residuos
figure;
grid; view(3) ; xlim([0 2*pi]) ;
xlabel('\psi_1 [rad]'); ylabel('k_r^1'); zlabel('k_r^2') ;
hold on ;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


[ PSI1, KR1, KR2 ]  = meshgrid( psi1_range, y, y ) ;

IN.B1.psi = PSI1 ;

% Creation of reference data (data in local coordinate system, each
rod)
IN.B1.kr = KR1 ;
```

```
[ X1_loc_p, Y1_loc_p, R1_p ] = EmpArt_xyR_local( IN.B1 ) ;

IN.B1.kr = - KR1 ;
[ X1_loc_n, Y1_loc_n, R1_n ] = EmpArt_xyR_local( IN.B1 ) ;




% for ii = 2:size(PSI1,3)
%     X1_loc_p(:,:,ii) = X1_loc_p(:,:,1) ;
%     Y1_loc_p(:,:,ii) = Y1_loc_p(:,:,1) ;
%     R1_p(:,:,ii) = R1_p(:,:,1) ;
%
%     X1_loc_n(:,:,ii) = X1_loc_n(:,:,1) ;
%     Y1_loc_n(:,:,ii) = Y1_loc_n(:,:,1) ;
%     R1_n(:,:,ii) = R1_n(:,:,1) ;
% end


%%%%%%%%%%%%%%%%%%%%%%%%% Obtención de PSI2, ojo con carga es
diferente



% PSI2 = PSI1 + theta1 - theta2 + pi ;

% PSI2_n is calculated with negative KR1
aux1 = - R1_n.*sin(PSI1 + theta1 - theta2) - FP(1).*sin(theta2) +
FP(2).*cos(theta2) ;
aux2 = - R1_n.*cos(PSI1 + theta1 - theta2) + FP(1).*cos(theta2) +
FP(2).*sin(theta2) ;
PSI2_n = atan2(aux1, aux2) ;
R2_n_aux = cos(PSI2_n).*(aux2) + sin(PSI2_n).*(aux1) ;

% PSI2_p is calculated with positive KR1
aux1 = - R1_p.*sin(PSI1 + theta1 - theta2) - FP(1).*sin(theta2) +
FP(2).*cos(theta2) ;
aux2 = - R1_p.*cos(PSI1 + theta1 - theta2) + FP(1).*cos(theta2) +
FP(2).*sin(theta2) ;
PSI2_p = atan2(aux1, aux2) ;
R2_p_aux = cos(PSI2_p).*(aux2) + sin(PSI2_p).*(aux1) ;


pos_n = union(find(PSI2_n>2*pi), find(PSI2_n<0)) ;
PSI2_n(pos_n) = atan2(sin(PSI2_n(pos_n)),cos(PSI2_n(pos_n))) ;
PSI2_n(PSI2_n<0) = PSI2_n(PSI2_n<0) + 2*pi ;

pos_p = union(find(PSI2_p>2*pi), find(PSI2_p<0)) ;
PSI2_p(pos_p) = atan2(sin(PSI2_p(pos_p)),cos(PSI2_p(pos_p))) ;
PSI2_p(PSI2_p<0) = PSI2_p(PSI2_p<0) + 2*pi ;

% IN.B2.psi = PSI2(1,:,:) ;
%%%%%%%%%%%%%%%%%%%%%%%%%

% kr1+  kr2+
IN.B2.psi = PSI2_p ;
IN.B2.kr = KR2 ;
[ X2_loc_pp, Y2_loc_pp, R2_pp ] = EmpArt_xyR_local( IN.B2 ) ;
```

```
% kr1+  kr2-
IN.B2.psi = PSI2_p ;
IN.B2.kr = - KR2 ;
[ X2_loc_pn, Y2_loc_pn, R2_pn ] = EmpArt_xyR_local( IN.B2 ) ;

% kr1-  kr2+
IN.B2.psi = PSI2_n ;
IN.B2.kr = KR2 ;
[ X2_loc_np, Y2_loc_np, R2_np ] = EmpArt_xyR_local( IN.B2 ) ;

% kr1-  kr2-
IN.B2.psi = PSI2_n ;
IN.B2.kr = - KR2 ;
[ X2_loc_nn, Y2_loc_nn, R2_nn ] = EmpArt_xyR_local( IN.B2 ) ;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%



% for ii = 2:size(PSI1,1)
%     X2_loc_pp(ii,:,:) = X2_loc_pp(1,:,:) ;
%     Y2_loc_pp(ii,:,:) = Y2_loc_pp(1,:,:) ;
%     R2_pp(ii,:,:) = R2_pp(1,:,:) ;
%
%     X2_loc_pn(ii,:,:) = X2_loc_pn(1,:,:) ;
%     Y2_loc_pn(ii,:,:) = Y2_loc_pn(1,:,:) ;
%     R2_pn(ii,:,:) = R2_pn(1,:,:) ;
%
%     X2_loc_np(ii,:,:) = X2_loc_np(1,:,:) ;
%     Y2_loc_np(ii,:,:) = Y2_loc_np(1,:,:) ;
%     R2_np(ii,:,:) = R2_np(1,:,:) ;
%
%     X2_loc_nn(ii,:,:) = X2_loc_nn(1,:,:) ;
%     Y2_loc_nn(ii,:,:) = Y2_loc_nn(1,:,:) ;
%     R2_nn(ii,:,:) = R2_nn(1,:,:) ;
%
% end


%%%%%%%%%%%%%%%%%%%%%%% kr1+ kr2+ %%%%%%%%%%%%%%%%%%%%%%%
% Residue
resX   = X1_loc_p*cos(IN.B1.theta) - Y1_loc_p*sin(IN.B1.theta) - ...
         X2_loc_pp*cos(IN.B2.theta) + Y2_loc_pp*sin(IN.B2.theta) + ...
         IN.B1.x0 - IN.B2.x0 ;
resY   = X1_loc_p*sin(IN.B1.theta) + Y1_loc_p*cos(IN.B1.theta) - ...
         X2_loc_pp*sin(IN.B2.theta) - Y2_loc_pp*cos(IN.B2.theta) + ...
         IN.B1.y0 - IN.B2.y0 ;

% disp('resX_pp'); disp(resX(:,1,1)) ;
% disp('resY_pp'); disp(resY(:,1,1)) ;

%%%%%%%%%%%%% residuo de fuerzas cambia si carga no nula
% resR   = R1_p - R2_pp ;
% aux1 = - R1_p.*sin(PSI1 + theta1 - theta2) - FP(1).*sin(theta2) +
FP(2).*cos(theta2) ;
% aux2 = - R1_p.*cos(PSI1 + theta1 - theta2) + FP(1).*cos(theta2) +
FP(2).*sin(theta2) ;
% R2_p_aux = cos(PSI2_p).*(aux2) + sin(PSI2_p).*(aux1) ;
```

```
resR = R2_p_aux - R2_pp ;

% disp('resR_pp'); disp(resR(:,1,1)) ;

% Isolines and intersection of them (solution points)
[ psi1_pp, kr1_pp, kr2_pp ] = isoline_intersection( PSI1, KR1, KR2,
...
                                                    resX, resY, resR )
;


%%%%%%%%%%%%%%%%%%%%%%% kr1+ kr2- %%%%%%%%%%%%%%%%%%%%%%%
% Residue
resX  = X1_loc_p*cos(IN.B1.theta) - Y1_loc_p*sin(IN.B1.theta) - ...
        X2_loc_pn*cos(IN.B2.theta) + Y2_loc_pn*sin(IN.B2.theta) + ...
        IN.B1.x0 - IN.B2.x0 ;
resY  = X1_loc_p*sin(IN.B1.theta) + Y1_loc_p*cos(IN.B1.theta) - ...
        X2_loc_pn*sin(IN.B2.theta) - Y2_loc_pn*cos(IN.B2.theta) + ...
        IN.B1.y0 - IN.B2.y0 ;

% disp('resX_pn'); disp(resX(:,1,1)) ;
% disp('resY_pn'); disp(resY(:,1,1)) ;

%%%%%%%%%%%%% residuo de fuerzas cambia si catga no nula
% resR  = R1_p - R2_pn ;
% aux1 = - R1_p.*sin(PSI1 + theta1 - theta2) - FP(1).*sin(theta2) +
FP(2).*cos(theta2) ;
% aux2 = - R1_p.*cos(PSI1 + theta1 - theta2) + FP(1).*cos(theta2) +
FP(2).*sin(theta2) ;
% R2_p_aux = cos(PSI2_p).*(aux2) + sin(PSI2_p).*(aux1) ;
resR = R2_p_aux - R2_pn ;

% disp('resR_pn'); disp(resR(:,1,1)) ;

% Isolines and intersection of them (solution points)
[ psi1_pn, kr1_pn, kr2_pn ] = isoline_intersection( PSI1, KR1, -KR2,
...
                                                    resX, resY, resR )
;


%%%%%%%%%%%%%%%%%%%%%%% kr1- kr2+ %%%%%%%%%%%%%%%%%%%%%%%
% Residue
resX  = X1_loc_n*cos(IN.B1.theta) - Y1_loc_n*sin(IN.B1.theta) - ...
        X2_loc_np*cos(IN.B2.theta) + Y2_loc_np*sin(IN.B2.theta) + ...
        IN.B1.x0 - IN.B2.x0 ;
resY  = X1_loc_n*sin(IN.B1.theta) + Y1_loc_n*cos(IN.B1.theta) - ...
        X2_loc_np*sin(IN.B2.theta) - Y2_loc_np*cos(IN.B2.theta) + ...
        IN.B1.y0 - IN.B2.y0 ;

% disp('resX_np'); disp(resX(:,1,1)) ;
% disp('resY_np'); disp(resY(:,1,1)) ;

%%%%%%%%%%%%% residuo de fuerzas cambia si catga no nula
% resR  = R1_n - R2_np ;
% aux1 = - R1_n.*sin(PSI1 + theta1 - theta2) - FP(1).*sin(theta2) +
FP(2).*cos(theta2) ;
% aux2 = - R1_n.*cos(PSI1 + theta1 - theta2) + FP(1).*cos(theta2) +
FP(2).*sin(theta2) ;
```

```
% R2_n_aux = cos(PSI2_n).*(aux2) + sin(PSI2_n).*(aux1) ;
resR = R2_n_aux - R2_np ;

% disp('resR_np'); disp(resR(:,1,1)) ;

% Isolines and intersection of them (solution points)
[ psi1_np, kr1_np, kr2_np ] = isoline_intersection( PSI1, -KR1, KR2,
...
                                                     resX, resY, resR )
;


%%%%%%%%%%%%%%%%%%%%%%% kr1- kr2- %%%%%%%%%%%%%%%%%%%%%%
% Residue
resX   = X1_loc_n*cos(IN.B1.theta) - Y1_loc_n*sin(IN.B1.theta) - ...
         X2_loc_nn*cos(IN.B2.theta) + Y2_loc_nn*sin(IN.B2.theta) + ...
         IN.B1.x0 - IN.B2.x0 ;
resY   = X1_loc_n*sin(IN.B1.theta) + Y1_loc_n*cos(IN.B1.theta) - ...
         X2_loc_nn*sin(IN.B2.theta) - Y2_loc_nn*cos(IN.B2.theta) + ...
         IN.B1.y0 - IN.B2.y0 ;

% disp('resX_nn'); disp(resX(:,1,1)) ;
% disp('resY_nn'); disp(resY(:,1,1)) ;

%%%%%%%%%%%%% residuo de fuerzas cambia si catga no nula
% resR  = R1_n - R2_nn ;
% aux1 = - R1_n.*sin(PSI1 + theta1 - theta2) - FP(1).*sin(theta2) +
FP(2).*cos(theta2) ;
% aux2 = - R1_n.*cos(PSI1 + theta1 - theta2) + FP(1).*cos(theta2) +
FP(2).*sin(theta2) ;
% R2_n_aux = cos(PSI2_n).*(aux2) + sin(PSI2_n).*(aux1) ;
resR = R2_n_aux - R2_nn ;

% disp('resR_nn'); disp(resR(:,1,1)) ;

% Isolines and intersection of them (solution points)
[ psi1_nn, kr1_nn, kr2_nn ] = isoline_intersection( PSI1, -KR1, -KR2,
...
                                                     resX, resY, resR )
;




%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%
%%%%%%%%%%%%%%%%%%%%%% ACCURATE SOLUTIONS CALCULATION
%%%%%%%%%%%%%%%%%%%%%%
% Se toman las soluciones potenciales obtenidas del cumplimiento de
los
% residuos y se refinan usando Newton-Raphson con las ecuaciones de
las
% barras


psi1_sol = [ psi1_pp ;
             psi1_pn ;
```

```
             psi1_np ;
             psi1_nn ] ;
kr1_sol   = [ kr1_pp ;
             kr1_pn ;
             kr1_np ;
             kr1_nn ] ;
kr2_sol   = [ kr2_pp ;
             kr2_pn ;
             kr2_np ;
             kr2_nn ] ;

% Calculation of psi2, otra vez ojo!! cambiar con carga

IN.B1.kr  = kr1_pp ;
IN.B1.psi = psi1_pp ;
[ X1_loc_pp, Y1_loc_pp, R1_pp ] = EmpArt_xyR_local( IN.B1 ) ;
aux1 = - R1_pp.*sin(psi1_pp + theta1 - theta2) - FP(1).*sin(theta2) +
FP(2).*cos(theta2) ;
aux2 = - R1_pp.*cos(psi1_pp + theta1 - theta2) + FP(1).*cos(theta2) +
FP(2).*sin(theta2) ;
psi2_pp = atan2(aux1, aux2) ;

IN.B1.kr  = kr1_pn ;
IN.B1.psi = psi1_pn ;
[ X1_loc_pn, Y1_loc_pn, R1_pn ] = EmpArt_xyR_local( IN.B1 ) ;
aux1 = - R1_pn.*sin(psi1_pn + theta1 - theta2) - FP(1).*sin(theta2) +
FP(2).*cos(theta2) ;
aux2 = - R1_pn.*cos(psi1_pn + theta1 - theta2) + FP(1).*cos(theta2) +
FP(2).*sin(theta2) ;
psi2_pn = atan2(aux1, aux2) ;

IN.B1.kr  = kr1_np ;
IN.B1.psi = psi1_np ;
[ X1_loc_np, Y1_loc_np, R1_np ] = EmpArt_xyR_local( IN.B1 ) ;
aux1 = - R1_np.*sin(psi1_np + theta1 - theta2) - FP(1).*sin(theta2) +
FP(2).*cos(theta2) ;
aux2 = - R1_np.*cos(psi1_np + theta1 - theta2) + FP(1).*cos(theta2) +
FP(2).*sin(theta2) ;
psi2_np = atan2(aux1, aux2) ;

IN.B1.kr  = kr1_nn ;
IN.B1.psi = psi1_nn ;
[ X1_loc_nn, Y1_loc_nn, R1_nn ] = EmpArt_xyR_local( IN.B1 ) ;
aux1 = - R1_nn.*sin(psi1_nn + theta1 - theta2) - FP(1).*sin(theta2) +
FP(2).*cos(theta2) ;
aux2 = - R1_nn.*cos(psi1_nn + theta1 - theta2) + FP(1).*cos(theta2) +
FP(2).*sin(theta2) ;
psi2_nn = atan2(aux1, aux2) ;

psi2_sol = [ psi2_pp ;
             psi2_pn ;
             psi2_np ;
             psi2_nn  ] ;

% psi2_sol = psi1_sol + theta1 - theta2 + pi ;
pos = union(find(psi2_sol>2*pi), find(psi2_sol<0)) ;
psi2_sol(pos) = atan2(sin(psi2_sol(pos)),cos(psi2_sol(pos))) ;
psi2_sol(psi2_sol<0) = psi2_sol(psi2_sol<0) + 2*pi ;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```matlab
% Graphic representation of each solution
figure;
axis image ;
xlabel('x [m]'); ylabel('y [m]') ;
grid;
hold on ;

for ii = 1:numel(psi1_sol)
    IN.B1.psi = psi1_sol(ii) ;
    IN.B1.kr  = kr1_sol(ii) ;
    IN.B2.psi = psi2_sol(ii) ;
    IN.B2.kr  = kr2_sol(ii) ;

    [ IN ] = FK_NewtonRaphson( IN ) ;

    % Verification if the solution has been reached or not
    if IN.sol == 1
        psi1_sol(ii) = IN.B1.psi ;
        kr1_sol(ii)  = IN.B1.kr ;
        psi2_sol(ii) = IN.B2.psi ;
        kr2_sol(ii)  = IN.B2.kr ;

        GraphicRepresentation( IN, 100 ) ;
    else
        psi1_sol(ii) = NaN ;
        kr1_sol(ii)  = NaN ;
        psi2_sol(ii) = NaN ;
        kr2_sol(ii)  = NaN ;
    end

    % Final value solutions showed in Command Window
    fprintf('\nSolution %d\n', ii) ;
    fprintf('psi1 = %f [rad]\n', psi1_sol(ii)) ;
    fprintf('kr1 = %f [-]\n', kr1_sol(ii)) ;
    fprintf('psi2 = %f [rad]\n', psi2_sol(ii)) ;
    fprintf('kr2 = %f [-]\n\n', kr2_sol(ii)) ;
end

toc;

% Fitting are of graphic representation
xlimite = xlim ;
ylimite = ylim ;
xlim([xlimite(1)-0.05 xlimite(2)+0.05]) ;
ylim([ylimite(1)-0.05 ylimite(2)+0.05]) ;

save('FK_MultipleSolutions') ;
```

3.5. INTEGRALES ELÍPTICAS. FUNCIONES ASOCIADAS

```
function [ xi, yi ] = Elastic_shape_EmpArt( IN, n )
% Functions that calculates de position of the pinned end of a
clamped-
% pinned rod and the required load.

% IN :  Structure with the input values
%   IN.L :      Lenght of the rod [m]
%   IN.EI :     Inertia of the cross section [m^4]
%   IN.mode :   Mode of deformation
%   IN.psi :    Angle of the force (relative to the clamped angle)
[rad]
%   IN.kr :     Elliptic integral parameter (relative value)
%   IN.theta :  Angle of the clamped end [rad]
%   IN.x0 :     x position of the clamped end [m]
%   IN.y0 :     y position of the clamped end [m]

% Value of k
kmin = abs(cos(0.5*IN.psi)) ;
k    = sign(IN.kr).*kmin + (1-kmin).*IN.kr ;


% Range of phi
aux  = 1./k.*cos(0.5*IN.psi);
phi1 = asin(aux);
phi2 = (IN.mode-0.5)*pi ;
phi = linspace(phi1, phi2, n) ;


% Deformation of the rod in local coordinate
[F,E] = elliptic12( phi, k^2 );

alpha = F(end)-F(1);

aux1 = (2*E-2*E(1)-F+F(1))/alpha*IN.L;
aux2 = 2*(cos(phi)-cos(phi1))/alpha*IN.L*k;

xlocali = -aux1.*cos(IN.psi)-aux2.*sin(IN.psi);
ylocali = -aux1.*sin(IN.psi)+aux2.*cos(IN.psi);


% Deformation of the road in global coordinate
xi = xlocali*cos(IN.theta) - ylocali*sin(IN.theta) ;
yi = xlocali*sin(IN.theta) + ylocali*cos(IN.theta) ;

xi = xi + IN.x0 + IN.lambda*cos(IN.theta) ;
yi = yi + IN.y0 + IN.lambda*sin(IN.theta) ;

end

function [F,E,Z] = elliptic12(u,m,tol)
% ELLIPTIC12 evaluates the value of the Incomplete Elliptic Integrals
% of the First, Second Kind and Jacobi's Zeta Function.
%
%   [F,E,Z] = ELLIPTIC12(U,M,TOL) where U is a phase in radians, 0<M<1
is
%   the module and TOL is the tolerance (optional). Default value for
%   the tolerance is eps = 2.220e-16.
```

```
%
%   ELLIPTIC12 uses the method of the Arithmetic-Geometric Mean
%   and Descending Landen Transformation described in [1] Ch. 17.6,
%   to determine the value of the Incomplete Elliptic Integrals
%   of the First, Second Kind and Jacobi's Zeta Function [1], [2].
%
%       F(phi,m) = int(1/sqrt(1-m*sin(t)^2), t=0..phi);
%       E(phi,m) = int(sqrt(1-m*sin(t)^2), t=0..phi);
%       Z(phi,m) = E(u,m) - E(m)/K(m)*F(phi,m).
%
%   Tables generating code ([1], pp. 613-621):
%       [phi,alpha] = meshgrid(0:5:90, 0:2:90);                  %
modulus and phase in degrees
%       [F,E,Z] = elliptic12(pi/180*phi, sin(pi/180*alpha).^2);  %
values of integrals
%
%   See also ELLIPKE, ELLIPJ, ELLIPTIC12I, ELLIPTIC3, THETA, AGM.
%
%   References:
%   [1] M. Abramowitz and I.A. Stegun, "Handbook of Mathematical
Functions",
%       Dover Publications", 1965, Ch. 17.1 - 17.6 (by L.M. Milne-
Thomson).
%   [2] D. F. Lawden, "Elliptic Functions and Applications"
%       Springer-Verlag, vol. 80, 1989

% Moiseev Igor
% 34106, SISSA, via Beirut n. 2-4,  Trieste, Italy
% For support, please reply to
%     moiseev[at]sissa.it, moiseev.igor[at]gmail.com
%     Moiseev Igor,
%     34106, SISSA, via Beirut n. 2-4,  Trieste, Italy
%
% The code is optimized for ordered inputs produced by the functions
% meshgrid, ndgrid. To obtain maximum performace (up to 30%) for
singleton,
% 1-dimensional and random arrays remark call of the function
unique(.)
% and edit further code.

if nargin<3, tol = eps; end
if nargin<2, error('Not enough input arguments.'); end

if ~isreal(u) || ~isreal(m)
    error('Input arguments must be real. Use ELLIPTIC12i for complex
arguments.');
end

if length(m)==1, m = m(ones(size(u))); end
if length(u)==1, u = u(ones(size(m))); end
if ~isequal(size(m),size(u)), error('U and M must be the same size.');
end

F = zeros(size(u));
E = F;
Z = E;
m = m(:).';     % make a row vector
u = u(:).';
```

```
if any(m < 0) || any(m > 1), error('M must be in the range 0 <= M <=
1.'); end

I = uint32( find(m ~= 1 & m ~= 0) );
if ~isempty(I)
    [mu,J,K] = unique(m(I));   % extracts unique values from m
    K = uint32(K);
    mumax = length(mu);
    signU = sign(u(I));

    % pre-allocate space and augment if needed
    chunk = 7;
    a = zeros(chunk,mumax);
    c = a;
    b = a;
    a(1,:) = ones(1,mumax);
    c(1,:) = sqrt(mu);
    b(1,:) = sqrt(1-mu);
    n = uint32( zeros(1,mumax) );
    i = 1;
    while any(abs(c(i,:)) > tol)                                    %
Arithmetic-Geometric Mean of A, B and C
        i = i + 1;
        if i > size(a,1)
          a = [a; zeros(2,mumax)];
          b = [b; zeros(2,mumax)];
          c = [c; zeros(2,mumax)];
        end
        a(i,:) = 0.5 * (a(i-1,:) + b(i-1,:));
        b(i,:) = sqrt(a(i-1,:) .* b(i-1,:));
        c(i,:) = 0.5 * (a(i-1,:) - b(i-1,:));
        in = uint32( find((abs(c(i,:)) <= tol) & (abs(c(i-1,:)) >
tol)) );
        if ~isempty(in)
          [mi,ni] = size(in);
          n(in) = ones(mi,ni)*(i-1);
        end
    end

    mmax = length(I);
    mn = double(max(n));
    phin = zeros(1,mmax);     C  = zeros(1,mmax);
    Cp = C;   e  = uint32(C);  phin(:) = signU.*u(I);
    i = 0;   c2 = c.^2;
    while i < mn                                                    %
Descending Landen Transformation
        i = i + 1;
        in = uint32(find(n(K) > i));
        if ~isempty(in)
            phin(in) = atan(b(i,K(in))./a(i,K(in)).*tan(phin(in))) +
...
                pi.*ceil(phin(in)/pi - 0.5) + phin(in);
            e(in) = 2.^(i-1) ;
            C(in) = C(in)  + double(e(in(1)))*c2(i,K(in));
            Cp(in)= Cp(in) + c(i+1,K(in)).*sin(phin(in));
        end
    end

    Ff = phin ./ (a(mn,K).*double(e)*2);
    F(I) = Ff.*signU;                                               %
Incomplete Ell. Int. of the First Kind
```

```
    Z(I) = Cp.*signU;                                                    %
Jacobi Zeta Function
    E(I) = (Cp + (1 - 1/2*C) .* Ff).*signU;                              %
Incomplete Ell. Int. of the Second Kind
end

% Special cases: m == {0, 1}
m0 = find(m == 0);
if ~isempty(m0), F(m0) = u(m0); E(m0) = u(m0); Z(m0) = 0; end

m1 = find(m == 1);
um1 = abs(u(m1));
if ~isempty(m1),
    N = floor( (um1+pi/2)/pi );
    M = find(um1 < pi/2);

    F(m1(M)) = log(tan(pi/4 + u(m1(M))/2));
    F(m1(um1 >= pi/2)) = Inf.*sign(u(m1(um1 >= pi/2)));

    E(m1) = ((-1).^N .* sin(um1) + 2*N).*sign(u(m1));

    Z(m1) = (-1).^N .* sin(u(m1));
end

 function [ x, y, Fx, Fy ] = EmpArt_xyR( IN )
% Functions that calculates de position of the pinned end of a
clamped-
% pinned rod and the required load.

% IN :  Structure with the input values
%    IN.L :       Lenght of the rod [m]
%    IN.EI :      Inertia of the cross section [m^4]
%    IN.mode :    Mode of deformation
%    IN.psi :     Angle of the force (relative to the clamped angle)
[rad]
%    IN.kr :      Elliptic integral parameter (relative value)
%    IN.theta :   Angle of the clamped end [rad]
%    IN.x0 :      x position of the clamped end [m]
%    IN.y0 :      y position of the clamped end [m]


posNaN = find(not(abs(IN.kr)>=0)) ;
IN.kr(posNaN) = 0.5 ;


kmin = abs(cos(0.5*IN.psi)) ;
k    = sign(IN.kr).*kmin + (1-kmin).*IN.kr ;

aux  = 1./k.*cos(0.5*IN.psi);
phi1 = asin(aux);

[F1, E1] = elliptic12( phi1, k.^2 );
[F2, E2] = elliptic12( (IN.mode-0.5)*pi, k.^2 );

alpha = F2-F1;
R = alpha.^2*IN.EI/IN.L^2;

aux1 = (2*E2-2*E1-F2+F1)./alpha ;
```

```
aux2 = -2*cos(phi1)./alpha.*k;

xlocal = IN.L*(-aux1.*cos(IN.psi)-aux2.*sin(IN.psi));
ylocal = IN.L*(-aux1.*sin(IN.psi)+aux2.*cos(IN.psi));

x = xlocal.*cos(IN.theta) - ylocal.*sin(IN.theta) ;
y = xlocal.*sin(IN.theta) + ylocal.*cos(IN.theta) ;


 function [ x, y, R ] = EmpArt_xyR_local( IN )
% Functions that calculates de position of the pinned end of a
clamped-
% pinned rod and the required load.

% IN :  Structure with the input values
%   IN.L :      Lenght of the rod [m]
%   IN.EI :     Inertia of the cross section [m^4]
%   IN.mode :   Mode of deformation
%   IN.psi :    Angle of the force (relative to the clamped angle)
[rad]
%   IN.kr :     Elliptic integral parameter (relative value)


posNaN = find(not(abs(IN.kr)>=0)) ;
IN.kr(posNaN) = 0.5 ;


kmin = abs(cos(0.5*IN.psi)) ;
k    = sign(IN.kr).*kmin + (1-kmin).*IN.kr ;

aux  = 1./k.*cos(0.5*IN.psi);
phi1 = asin(aux);

[F1, E1] = elliptic12( phi1, k.^2 );
[F2, E2] = elliptic12( (IN.mode-0.5)*pi, k.^2 );

alpha = F2-F1;
R = alpha.^2*IN.EI/IN.L^2;

aux1 = (2*E2-2*E1-F2+F1)./alpha ;
aux2 = -2*cos(phi1)./alpha.*k;

x = IN.L*(-aux1.*cos(IN.psi)-aux2.*sin(IN.psi));
y = IN.L*(-aux1.*sin(IN.psi)+aux2.*cos(IN.psi));


if numel(IN.kr) == 1 && numel(posNaN)==1
    x = NaN ;
    y = NaN ;
    R = NaN ;
else
    x(posNaN) = NaN ;
    y(posNaN) = NaN ;
    R(posNaN) = NaN ;
end

end
```

```
function [ pos ] = Finding_roots( resid )


% tic ;
siz = size(resid.res1) ;

Ni = size(resid.res1, 1) ;
Nj = size(resid.res1, 2) ;
Nk = size(resid.res1, 3) ;
Np = size(resid.res1, 4) ;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
max_aux = zeros(size(resid.res1))*NaN ;
min_aux = max_aux ;
aux     = max_aux ;

for ii = 1:2
    for jj = 1:2
        for kk = 1:2
            for pp = 1:2
                aux(1:Ni-1, 1:Nj-1, 1:Nk-1, 1:Np-1) = resid.res1(
ii:Ni+ii-2, jj:Nj+jj-2, kk:Nk+kk-2, pp:Np+pp-2 ) ;

                max_aux = max(max_aux, aux) ;
                min_aux = min(min_aux, aux) ;
            end
        end
    end
end
pos_r1 = find(max_aux.*min_aux<0) ;

[II_ini, JJ_ini, KK_ini, PP_ini] = ind2sub(siz, pos_r1) ;



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
max_aux = resid.res2(pos_r1) ;
min_aux = max_aux ;

for ii = 0:1
    for jj = 0:1
        for kk = 0:1
            for pp = 0:1
                II = II_ini + ii ;
                JJ = JJ_ini + jj ;
                KK = KK_ini + kk ;
                PP = PP_ini + pp ;

                pos = sub2ind(siz, II, JJ, KK, PP) ;
                aux = resid.res2( pos ) ;

                max_aux = max(max_aux, aux) ;
                min_aux = min(min_aux, aux) ;
            end
        end
    end
end
```

```
pos_r2 = find(max_aux.*min_aux<0) ;

II_ini = II_ini(pos_r2) ;
JJ_ini = JJ_ini(pos_r2) ;
KK_ini = KK_ini(pos_r2) ;
PP_ini = PP_ini(pos_r2) ;


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
max_aux = resid.res3(sub2ind(siz, II_ini, JJ_ini, KK_ini, PP_ini)) ;
min_aux = max_aux ;

for ii = 0:1
    for jj = 0:1
        for kk = 0:1
            for pp = 0:1
                II = II_ini + ii ;
                JJ = JJ_ini + jj ;
                KK = KK_ini + kk ;
                PP = PP_ini + pp ;

                pos = sub2ind(siz, II, JJ, KK, PP) ;
                aux = resid.res3( pos ) ;

                max_aux = max(max_aux, aux) ;
                min_aux = min(min_aux, aux) ;
            end
        end
    end
end
pos_r3 = find(max_aux.*min_aux<0) ;

II_ini = II_ini(pos_r3) ;
JJ_ini = JJ_ini(pos_r3) ;
KK_ini = KK_ini(pos_r3) ;
PP_ini = PP_ini(pos_r3) ;



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
max_aux = resid.res4(sub2ind(siz, II_ini, JJ_ini, KK_ini, PP_ini)) ;
min_aux = max_aux ;

for ii = 0:1
    for jj = 0:1
        for kk = 0:1
            for pp = 0:1
                II = II_ini + ii ;
                JJ = JJ_ini + jj ;
                KK = KK_ini + kk ;
                PP = PP_ini + pp ;

                pos = sub2ind(siz, II, JJ, KK, PP) ;
                aux = resid.res4( pos ) ;

                max_aux = max(max_aux, aux) ;
                min_aux = min(min_aux, aux) ;
```

```
            end
        end
    end
end
pos_r4 = find(max_aux.*min_aux<0) ;

II_ini = II_ini(pos_r4) ;
JJ_ini = JJ_ini(pos_r4) ;
KK_ini = KK_ini(pos_r4) ;
PP_ini = PP_ini(pos_r4) ;


pos = sub2ind( siz, II_ini, JJ_ini, KK_ini, PP_ini ) ;


end

 function [ Jacob ] = FK_Jacobian( IN )
% Jacobian matrix ...

epsilon = 1e-6 ;


IN = FK_NewtonRaphson( IN ) ;
[ x_ref, y_ref, ~, ~ ] = EmpArt_xyR( IN.B1 ) ;

IN.B1.theta = IN.B1.theta + epsilon ;
IN = FK_NewtonRaphson( IN ) ;
sol = IN.sol ;
[ x_incr1, y_incr1, ~, ~ ] = EmpArt_xyR( IN.B1 ) ;
IN.B1.theta = IN.B1.theta - epsilon ;

IN.B2.theta = IN.B2.theta + epsilon ;
IN = FK_NewtonRaphson( IN ) ;
sol = sol * IN.sol ;
[ x_incr2, y_incr2, ~, ~ ] = EmpArt_xyR( IN.B1 ) ;


Jacob = [ x_incr1 x_incr2 ;
          y_incr1 y_incr2 ] ;
Jacob = Jacob - [ ones(1,2)*x_ref ; ones(1,2)*y_ref ] ;

if sol == 1
    Jacob = Jacob/epsilon ;
else
    Jacob = Jacob*NaN ;
end


end


function [ IN ] = FK_NewtonRaphson( IN )
% Solution of the Forward Kinematic Position problem

IN.sol = 1 ;
```

```
% Maximun error tolerance of the residue vector
tol = 1e-9 ;

% Initial residue
[resid, resid_aux] = Residue( IN ) ;

resid0 = zeros(size(resid)) ;

% % RESID = resid ;

while max(abs(resid)) > tol && max(abs(resid-resid0)) > tol*0.1

    resid0 = resid ;

    % Jacobian matrix of the residue respect to the variables
    J = Jacobian(IN, resid_aux) ;

    % In case of NaN in J, abort calculation
    if not(sum(sum(abs(J)))>=0)
        break ;
    end

    % In case of not full rank of J, abort calculation
    if rank(J)<4
        break ;
    end

    % Increment values of the variables
    delta = -J\resid0 ;

    % Update variables values
    [IN, delta] = Update_variables( IN, delta ) ;

    if IN.sol == 0
        break ;
    end

    [resid, resid_aux] = Residue( IN ) ;

    while max(abs(resid)) > max(abs(resid0)) && max(abs(delta)) > 1e-
24
        delta = 0.5*delta ;

        IN.B1.psi = IN.B1.psi - delta(1) ;
        IN.B1.kr  = IN.B1.kr  - delta(2) ;
        IN.B2.psi = IN.B2.psi - delta(3) ;
        IN.B2.kr  = IN.B2.kr  - delta(4) ;

        [resid, resid_aux] = Residue( IN ) ;
    end

% %     RESID = [ RESID  resid ] ;
% %
% %     if size(RESID,2) == 100
% %         pausa = 1 ;
% %     end

end
```

```
% Check if the residue is not NaN and below the tolerance error
if max(abs(resid)) > tol || not(sum(abs(resid))>=0)
    IN.sol = 0 ;
else
    IN.xp = resid_aux.B1(1) ;
    IN.yp = resid_aux.B1(2) ;
end




function [IN, delta] = Update_variables( IN, delta )

while IN.B1.psi+delta(1)>2*pi || IN.B1.psi+delta(1)<0
    delta = 0.5*delta ;

    if max(abs(delta)) < 1e-24
        IN.sol = 0 ;
        break ;
    end
end

while IN.B2.psi+delta(3)>2*pi || IN.B2.psi+delta(3)<0
    delta = 0.5*delta ;

    if max(abs(delta)) < 1e-24
        IN.sol = 0 ;
        break ;
    end
end

while sign(IN.B1.kr)~=sign(IN.B1.kr+delta(2)) ||
abs(IN.B1.kr+delta(2))>=1
    delta = 0.5*delta ;

    if max(abs(delta)) < 1e-24
        IN.sol = 0 ;
        break ;
    end
end

while sign(IN.B2.kr)~=sign(IN.B2.kr+delta(4)) ||
abs(IN.B2.kr+delta(4))>=1
    delta = 0.5*delta ;

    if max(abs(delta)) < 1e-24
        IN.sol = 0 ;
        break ;
    end
end

IN.B1.psi = IN.B1.psi + delta(1) ;
IN.B1.kr  = IN.B1.kr  + delta(2) ;
IN.B2.psi = IN.B2.psi + delta(3) ;
IN.B2.kr  = IN.B2.kr  + delta(4) ;
```

```
function [ Jacob ] = Jacobian( IN, resid_aux )

epsilon = 1e-12 ;

Jacob = zeros(4) ;

% Rod 1
IN.B1.psi = IN.B1.psi + epsilon ;
[ x_A, y_A, Fx_A, Fy_A ] = EmpArt_xyR( IN.B1 ) ;
Jacob(:,1) = [ x_A ;
               y_A ;
               Fx_A ;
               Fy_A ] - ...
               resid_aux.B1 ;
IN.B1.psi = IN.B1.psi - epsilon ;

IN.B1.kr = IN.B1.kr + epsilon ;
[ x_A, y_A, Fx_A, Fy_A ] = EmpArt_xyR( IN.B1 ) ;
Jacob(:,2) = [ x_A ;
               y_A ;
               Fx_A ;
               Fy_A ] - ...
               resid_aux.B1 ;

% Rod 2
IN.B2.psi = IN.B2.psi + epsilon ;
[ x_B, y_B, Fx_B, Fy_B ] = EmpArt_xyR( IN.B2 ) ;
Jacob(:,3) = [ - x_B ;
               - y_B ;
               Fx_B ;
               Fy_B ] - ...
               resid_aux.B2 ;
IN.B2.psi = IN.B2.psi - epsilon ;

IN.B2.kr = IN.B2.kr + epsilon ;
[ x_B, y_B, Fx_B, Fy_B ] = EmpArt_xyR( IN.B2 ) ;
Jacob(:,4) = [ - x_B ;
               - y_B ;
               Fx_B ;
               Fy_B ] - ...
               resid_aux.B2 ;

Jacob = Jacob/epsilon ;
```

```
function [ resid, resid_aux ] = Residue( IN )

[ x_A, y_A, Fx_A, Fy_A ] = EmpArt_xyR( IN.B1 ) ;
[ x_B, y_B, Fx_B, Fy_B ] = EmpArt_xyR( IN.B2 ) ;

resid_aux.B1 = [ x_A ;
                 y_A ;
```

```
                 Fx_A ;
                 Fy_A ] ;
resid_aux.B2 = [ - x_B ;
                 - y_B ;
                 Fx_B ;
                 Fy_B ] ;

resid = resid_aux.B1 + resid_aux.B2 ;
resid = resid + [ 0 ;
                  0 ;
                  - IN.B1.FP(1) ;
                  - IN.B1.FP(2) ] ;

 function [ grad ] = Grad_resxy( IN )
%UNTITLED Summary of this function goes here
%   Detailed explanation goes here

epsilon = 1e-8 ;

[ res_xy, IN ] = psi1_kr1_2_resxy( IN ) ;

grad.res_xy = res_xy ;

grad_res_x = zeros(2,1) ;
grad_res_y = zeros(2,1) ;

IN.B1.psi = IN.B1.psi + epsilon ;
[ res_xy_eps, IN ] = psi1_kr1_2_resxy( IN ) ;
grad_res_x(1) = (res_xy_eps(1) - res_xy(1))/epsilon ;
grad_res_y(1) = (res_xy_eps(2) - res_xy(2))/epsilon ;
IN.B1.psi = IN.B1.psi - epsilon ;

IN.B1.kr = IN.B1.kr + epsilon ;
[ res_xy_eps, IN ] = psi1_kr1_2_resxy( IN ) ;
grad_res_x(2) = (res_xy_eps(1) - res_xy(1))/epsilon ;
grad_res_y(2) = (res_xy_eps(2) - res_xy(2))/epsilon ;

grad.grad_resx = grad_res_x ;
grad.grad_resy = grad_res_y ;

end

 function [  ] = GraphicRepresentation( IN, n )
% Graphic representation of the mechanism

[ xi_B1, yi_B1 ] = Elastic_shape_EmpArt( IN.B1, n ) ;
[ xi_B2, yi_B2 ] = Elastic_shape_EmpArt( IN.B2, n ) ;

plot(xi_B1, yi_B1, '-b', 'linewidth', 2 ) ;
plot(xi_B2, yi_B2, '-r', 'linewidth', 2 ) ;


% Straight tangets to the clamped end of each rod
long1 = 0.2*IN.B1.L ;
long2 = 0.2*IN.B2.L ;

x1 = [0 long1*cos(IN.B1.theta) ] + IN.B1.x0 +
IN.B1.lambda*cos(IN.B1.theta) ;
```

```
y1 = [0 long1*sin(IN.B1.theta) ] + IN.B1.y0 +
IN.B1.lambda*sin(IN.B1.theta) ;

x2 = [0 long2*cos(IN.B2.theta) ] + IN.B2.x0 +
IN.B2.lambda*cos(IN.B2.theta) ;
y2 = [0 long2*sin(IN.B2.theta) ] + IN.B2.y0 +
IN.B2.lambda*sin(IN.B2.theta) ;

plot(x1,y1, '--k') ;
plot(x2,y2, '--k') ;

% External force representation
quiver (xi_B1(end), yi_B1(end), IN.B1.FP(1), IN.B1.FP(2), 'g',
'LineWidth', 1.5) ;

end


 function [ Jacob ] = IK_Jacobian( IN )
%UNTITLED3 Summary of this function goes here
%   Detailed explanation goes here

epsilon = 1e-6 ;


IN = IK_NewtonRaphson( IN ) ;
theta1_ref = IN.B1.theta ;
theta2_ref = IN.B2.theta ;

IN.B1.xp = IN.B1.xp + epsilon ;
IN.B2.xp = IN.B2.xp + epsilon ;
IN = IK_NewtonRaphson( IN ) ;
sol = IN.sol ;
theta1_incrx = IN.B1.theta ;
theta2_incrx = IN.B2.theta ;
IN.B1.xp = IN.B1.xp - epsilon ;
IN.B2.xp = IN.B2.xp - epsilon ;

IN.B1.yp = IN.B1.yp + epsilon ;
IN.B2.yp = IN.B2.yp + epsilon ;
IN = IK_NewtonRaphson( IN ) ;
sol = sol*IN.sol ;
theta1_incry = IN.B1.theta ;
theta2_incry = IN.B2.theta ;

Jacob = [ theta1_incrx theta1_incry ;
          theta2_incrx theta2_incry ] ;
Jacob = Jacob - [ ones(1,2)*theta1_ref ; ones(1,2)*theta2_ref ] ;

if sol == 1
    Jacob = Jacob/epsilon ;
else
    Jacob = Jacob*NaN ;
end


end
```

```
function [ K ] = StiffnessMatrix( IN )
%UNTITLED7 Summary of this function goes here
%   Detailed explanation goes here

epsilon = 1e-6 ;

K = zeros(2) ;

IN.B1.xp = IN.xp ;
IN.B2.xp = IN.xp ;
IN.B1.yp = IN.yp ;
IN.B2.yp = IN.yp ;

IN.B1.xp = IN.B1.xp + epsilon ;
IN.B2.xp = IN.B2.xp + epsilon ;
IN.B1 = IK_NewtonRaphson_rod( IN.B1 ) ;
IN.B2 = IK_NewtonRaphson_rod( IN.B2 ) ;
if IN.B1.sol*IN.B2.sol == 1
    K(:,1) = [ IN.B1.Fx + IN.B2.Fx ;
               IN.B1.Fy + IN.B2.Fy ]/epsilon ;
else
    K(:,1) = NaN ;
end
IN.B1.xp = IN.B1.xp - epsilon ;
IN.B2.xp = IN.B2.xp - epsilon ;

IN.B1.yp = IN.B1.yp + epsilon ;
IN.B2.yp = IN.B2.yp + epsilon ;
IN.B1 = IK_NewtonRaphson_rod( IN.B1 ) ;
IN.B2 = IK_NewtonRaphson_rod( IN.B2 ) ;
if IN.B1.sol*IN.B2.sol == 1
    K(:,2) = [ IN.B1.Fx + IN.B2.Fx ;
               IN.B1.Fy + IN.B2.Fy ]/epsilon ;
else
    K(:,2) = NaN ;
end
IN.B1.yp = IN.B1.yp - epsilon ;
IN.B2.yp = IN.B2.yp - epsilon ;

end

function [ K ] = StiffnessMatrix_SM( RODS, Fext )
% Stiffness matrix
K = zeros(2) ;

% Increment of xp and yp position
epsilon = 1e-5 ;



RODS = FK_ShootingMethod( RODS, Fext ) ;

RODS.OUT.xp = 0.5*(RODS.OUT.B1.p(1,end) + RODS.OUT.B2.p(1,end)) ;
RODS.OUT.yp = 0.5*(RODS.OUT.B1.p(2,end) + RODS.OUT.B2.p(2,end)) ;

RODS.IN.B1.theta = RODS.IN.B1.theta0 ;
RODS.IN.B1.px_0 = RODS.IN.B1.OA(1) ;
RODS.IN.B1.py_0 = RODS.IN.B1.OA(2) ;
RODS.IN.B1.n = RODS.IN.B1.n0 ;
```

```
RODS.IN.B2.theta = RODS.IN.B2.theta0 ;
RODS.IN.B2.px_0 = RODS.IN.B2.OA(1) ;
RODS.IN.B2.py_0 = RODS.IN.B2.OA(2) ;
RODS.IN.B2.n = RODS.IN.B2.n0 ;

RODS_ref = RODS ;

% Increment in xp
RODS.IN.B1.px_l = RODS_ref.OUT.xp + 0.5*epsilon ;
RODS.IN.B2.px_l = RODS_ref.OUT.xp + 0.5*epsilon ;
RODS.IN.B1.py_l = RODS_ref.OUT.yp ;
RODS.IN.B2.py_l = RODS_ref.OUT.yp ;


[ IN1, OUT1 ] = IK_SM_rod( RODS.IN.B1 ) ;
[ IN2, OUT2 ] = IK_SM_rod( RODS.IN.B2 ) ;

if OUT1.sol==1 && OUT2.sol==1
    K(:,1) = (IN1.n + IN2.n) ;

    RODS.IN.B1.px_l = RODS_ref.OUT.xp - 0.5*epsilon ;
    RODS.IN.B2.px_l = RODS_ref.OUT.xp - 0.5*epsilon ;

    [ IN1, OUT1 ] = IK_SM_rod( RODS.IN.B1 ) ;
    [ IN2, OUT2 ] = IK_SM_rod( RODS.IN.B2 ) ;

    if OUT1.sol==1 && OUT2.sol==1
        K(:,1) = K(:,1) - (IN1.n + IN2.n) ;
    else
        K(:,1) = NaN ;
    end

else
    K(:,1) = NaN ;
end


RODS = RODS_ref ;

% Increment in yp
RODS.IN.B1.py_l = RODS_ref.OUT.yp + 0.5*epsilon ;
RODS.IN.B2.py_l = RODS_ref.OUT.yp + 0.5*epsilon ;
RODS.IN.B1.px_l = RODS_ref.OUT.xp ;
RODS.IN.B2.px_l = RODS_ref.OUT.xp ;

[ IN1, OUT1 ] = IK_SM_rod( RODS.IN.B1 ) ;
[ IN2, OUT2 ] = IK_SM_rod( RODS.IN.B2 ) ;

if OUT1.sol==1 && OUT2.sol==1
    K(:,2) = (IN1.n + IN2.n) ;

    RODS.IN.B1.py_l = RODS_ref.OUT.yp - 0.5*epsilon ;
    RODS.IN.B2.py_l = RODS_ref.OUT.yp - 0.5*epsilon ;

    [ IN1, OUT1 ] = IK_SM_rod( RODS.IN.B1 ) ;
    [ IN2, OUT2 ] = IK_SM_rod( RODS.IN.B2 ) ;
```

```
    if OUT1.sol==1 && OUT2.sol==1
        K(:,2) = K(:,2) - (IN1.n + IN2.n) ;
    else
        K(:,2) = NaN ;
    end

else
    K(:,1) = NaN ;
end


K = K/epsilon ;


end
```

4. CÓDIGO DE LOS PROGRAMAS DEL MECANISMO 3PFR

4.1. INTEGRACIÓN NUMÉRICA. PROBLEMA INVERSO

```
% Initial design parameters of the mechanism
clear; close all;


%%%%%%%%%%%%%%%%%%%%%%%%%%%%% ROD PROPERTIES
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
diam = 1.5e-3 ;       % Diameter of the section [m]
rad = 0.5*diam ;
I = 0.25*pi*rad^4 ;  % Intertial of the cross-section [m^4]

L  = 1 ;              % Length of the rod [m]
d  = 0.10 ;           % Distance of the rigid part at the pinned end [m]

E = 210e9 ;           % Young's modulus [Pa]

N = 41 ;              % Number of discretitaiton points of the rod



%%%%%%%%%%%%%%%%%% GEOMETRIC DEFINITON OF THE MECHANISM
%%%%%%%%%%%%%%%%%%
% Angle of the guide of each linear actuator
alpha1 = 0/180*pi ;
alpha2 = 120/180*pi ;
alpha3 = -120/180*pi ;

% Angle of the rod at its clamped end
theta0_1 = 0/180*pi ;              % [rad]
theta0_2 = 120/180*pi ;       % [rad]
theta0_3 = -120/180*pi ;      % [rad]

% Reference point through which each of the guides passes
OA1 = [ cos(-150/180*pi); sin(-150/180*pi) ] ;       % [m]
OA2 = [ cos(-30/180*pi); sin(-30/180*pi) ] ;         % [m]
OA3 = [ 0; 1 ] ;                                     % [m]

% Reference point and angle of the end effector plattfomr
OP = [0; 0] ;    % [m]
theta = 60*pi/180 ;     % [rad]

% Local vector that depicts the distance between de reference point
and
% points in which the end effector plattform is linked to rods.
PB1_loc = 0.2*[ cos(-150/180*pi); sin(-150/180*pi) ] ;  % [m]
PB2_loc = 0.2*[ cos(-30/180*pi); sin(-30/180*pi) ] ;    % [m]
PB3_loc = 0.2*[ 0; 1 ] ;                                % [m]



%%%%%%%%%%%% CREATION OF THE STRUCTURE FOR THE SHOOTING METHOD
%%%%%%%%%%%%
% "GV" means here "guess value", a value that is not known a priori
```

```
RODS.IN.B1.alpha  = alpha1 ;
RODS.IN.B1.L      = L ;
RODS.IN.B1.d      = d ;
RODS.IN.B1.r      = PB1_loc ;
RODS.IN.B1.OA     = OA1 ;
RODS.IN.B1.N      = N ;
RODS.IN.B1.EI     = E*I ;
RODS.IN.B1.theta0 = theta0_1 ;
RODS.IN.B1.n0     = 0.04*[ cos(-150/180*pi); sin(-150/180*pi) ] ;   %
GV
RODS.IN.B1.m0     = -0.002 ;                                         %
GV
RODS.IN.B1.lambda = 0 ;                                              %
GV

RODS.IN.B2.alpha  = alpha2 ;
RODS.IN.B2.L      = L ;
RODS.IN.B2.d      = d ;
RODS.IN.B2.r      = PB2_loc ;
RODS.IN.B2.OA     = OA2 ;
RODS.IN.B2.N      = N ;
RODS.IN.B2.EI     = E*I ;
RODS.IN.B2.theta0 = theta0_2 ;
RODS.IN.B2.n0     = 0.04*[ cos(-30/180*pi); sin(-30/180*pi) ] ;     %
GV
RODS.IN.B2.m0     = -0.002 ;                                         %
GV
RODS.IN.B2.lambda = 0 ;                                              %
GV

RODS.IN.B3.alpha  = alpha3 ;
RODS.IN.B3.L      = L ;
RODS.IN.B3.d      = d ;
RODS.IN.B3.r      = PB3_loc ;
RODS.IN.B3.OA     = OA3 ;
RODS.IN.B3.N      = N ;
RODS.IN.B3.EI     = E*I ;
RODS.IN.B3.theta0 = theta0_3 ;
RODS.IN.B3.n0     = 0.04*[ 0; 1 ] ;                                  %
GV
RODS.IN.B3.m0     = -0.002 ;                                         %
GV
RODS.IN.B3.lambda = 0 ;                                              %
GV

RODS.IN.OP = OP ;
RODS.IN.theta = theta ;


% Solving the Inverse Kinematic Problem in order to get the solution
for
% the initial position ;
RODS = IK_ShootingMethod( RODS, [0;0], 0 ) ;

% Update de reference point from which lambda is measured
% RODS.IN.B1.OA = RODS.IN.B1.OA + RODS.IN.B1.lambda*[
cos(RODS.IN.B1.alpha); sin(RODS.IN.B1.alpha) ] ;
% RODS.IN.B2.OA = RODS.IN.B2.OA + RODS.IN.B2.lambda*[
cos(RODS.IN.B2.alpha); sin(RODS.IN.B2.alpha) ] ;
```

```
% RODS.IN.B3.OA = RODS.IN.B3.OA + RODS.IN.B3.lambda*[
cos(RODS.IN.B3.alpha); sin(RODS.IN.B3.alpha) ] ;
% RODS.IN.B1.lambda = 0 ;
% RODS.IN.B2.lambda = 0 ;
% RODS.IN.B2.lambda = 0 ;

RODS = IK_ShootingMethod( RODS, [0;0], 0 ) ;


% Save de structure data
save('RODS', 'RODS') ;


% Plotting the mechanism
figure;
axis equal; grid; hold on;
GraficRepresentation( RODS ) ;

if RODS.OUT.sol == 0
    printf('Aborted calculation\n') ;
end



% Algorithm that solves de Inverse Kinematic position problem through
the
% the use of the Shootin Method
clear; close all;

% Creation (1) or not(0) of a .gif file
graf_gif = 0 ;


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% INPUT DATA
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Loading data of the mechanism in the initial position
load('RODS');

% Data for the Inverse Kinematic position problem
data = xlsread('IK_inputs_workspace.xlsx') ;

incr_xp = data(:,1) ;        % Increment of x coordinate of the end
effector
                             % respect to value that xp takes in the
initial
                             % position
incr_yp = data(:,2) ;        % Increment of y coordinate of the end
effector
                             % respect to value that yp takes in the
initial
                             % position
incr_theta = data(:,3) ;     % increment of the angle of the end
effector
                             % respect to value that takes in the
initial
                             % position (in rad)
Fext    = data(:,4:5) ;      % External forces. The first column is the
                             % x component of the force and the second
                             % colum, the y component
```

```
Mext = data(:,6) ;          % External moment.


% Inital values of the angles of actuators
OP_ref    = RODS.IN.OP ;
theta_ref = RODS.IN.theta ;

% Number of positions to be calculated
Nii = length(incr_xp) ;



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% OUTPUT DATA
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
RESULTS.LAMBDA1 = zeros(Nii, 1) ;
RESULTS.LAMBDA2 = zeros(Nii, 1) ;
RESULTS.LAMBDA3 = zeros(Nii, 1) ;
RESULTS.FORCE1  = zeros(Nii, 1) ;
RESULTS.FORCE2  = zeros(Nii, 1) ;
RESULTS.FORCE3  = zeros(Nii, 1) ;

RESULTS.FK_J11 = zeros(Nii, 1) ;
RESULTS.FK_J12 = zeros(Nii, 1) ;
RESULTS.FK_J13 = zeros(Nii, 1) ;
RESULTS.FK_J21 = zeros(Nii, 1) ;
RESULTS.FK_J22 = zeros(Nii, 1) ;
RESULTS.FK_J23 = zeros(Nii, 1) ;
RESULTS.FK_J31 = zeros(Nii, 1) ;
RESULTS.FK_J32 = zeros(Nii, 1) ;
RESULTS.FK_J33 = zeros(Nii, 1) ;
RESULTS.detJFK = zeros(Nii, 1) ;

RESULTS.IK_J11 = zeros(Nii, 1) ;
RESULTS.IK_J12 = zeros(Nii, 1) ;
RESULTS.IK_J13 = zeros(Nii, 1) ;
RESULTS.IK_J21 = zeros(Nii, 1) ;
RESULTS.IK_J22 = zeros(Nii, 1) ;
RESULTS.IK_J23 = zeros(Nii, 1) ;
RESULTS.IK_J31 = zeros(Nii, 1) ;
RESULTS.IK_J32 = zeros(Nii, 1) ;
RESULTS.IK_J33 = zeros(Nii, 1) ;
RESULTS.detJIK = zeros(Nii, 1) ;



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% CALCULATION
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
h = figure(1);
if graf_gif == 1
    axis tight manual
    filename = 'IK_testAnimated.gif';
end


for ii = 1:Nii

    % Saving last RODS structure in case the shooting method does not
    % converge
    RODS0 = RODS;
```

```
    % Update the new end effector position
    RODS.IN.OP    = OP_ref    + [incr_xp(ii); incr_yp(ii)] ;
    RODS.IN.theta = theta_ref + incr_theta(ii) ;


    % Shooting method
    RODS = IK_ShootingMethod( RODS, Fext(ii,:)', Mext(ii) ) ;


    % Forward Kinematic Jacobian matrix
    FK_Jacob = FK_Jacobian( RODS, Fext(ii,:)', Mext(ii) ) ;

    % Inverse Kinematic Jacobian matrix
    IK_Jacob = IK_Jacobian( RODS, Fext(ii,:)', Mext(ii) ) ;

    if RODS.OUT.sol == 0
        %         fprintf('Aborted calculation\n') ;
        %         break ;

        % Solution is not valid. Invalid data are saved as NaN
        RESULTS.LAMBDA1(ii) = NaN ;
        RESULTS.LAMBDA2(ii) = NaN ;
        RESULTS.LAMBDA3(ii) = NaN ;
        RESULTS.FORCE1(ii)  = NaN ;
        RESULTS.FORCE2(ii)  = NaN ;
        RESULTS.FORCE3(ii)  = NaN ;

        RESULTS.FK_J11(ii) = NaN ;
        RESULTS.FK_J12(ii) = NaN ;
        RESULTS.FK_J13(ii) = NaN ;
        RESULTS.FK_J21(ii) = NaN ;
        RESULTS.FK_J22(ii) = NaN ;
        RESULTS.FK_J23(ii) = NaN ;
        RESULTS.FK_J31(ii) = NaN ;
        RESULTS.FK_J32(ii) = NaN ;
        RESULTS.FK_J33(ii) = NaN ;
        RESULTS.detJFK(ii) = NaN ;

        RESULTS.IK_J11(ii) = NaN ;
        RESULTS.IK_J12(ii) = NaN ;
        RESULTS.IK_J13(ii) = NaN ;
        RESULTS.IK_J21(ii) = NaN ;
        RESULTS.IK_J22(ii) = NaN ;
        RESULTS.IK_J23(ii) = NaN ;
        RESULTS.IK_J31(ii) = NaN ;
        RESULTS.IK_J32(ii) = NaN ;
        RESULTS.IK_J33(ii) = NaN ;
        RESULTS.detJIK(ii) = NaN ;


        % If shooting method does not converge RODS is not valid so
the
        % previous RODS structure has to be used in next iteration
        RODS = RODS0;

    else

        % Storing results
```

```
RESULTS.LAMBDA1(ii) = RODS.IN.B1.lambda ;
RESULTS.LAMBDA2(ii) = RODS.IN.B2.lambda ;
RESULTS.LAMBDA3(ii) = RODS.IN.B3.lambda ;
RESULTS.FORCE1(ii)  = RODS.OUT.B1.Force ;
RESULTS.FORCE2(ii)  = RODS.OUT.B2.Force ;
RESULTS.FORCE3(ii)  = RODS.OUT.B3.Force ;

RESULTS.FK_J11(ii) = FK_Jacob(1,1) ;
RESULTS.FK_J12(ii) = FK_Jacob(1,2) ;
RESULTS.FK_J13(ii) = FK_Jacob(1,3) ;
RESULTS.FK_J21(ii) = FK_Jacob(2,1) ;
RESULTS.FK_J22(ii) = FK_Jacob(2,2) ;
RESULTS.FK_J23(ii) = FK_Jacob(2,3) ;
RESULTS.FK_J31(ii) = FK_Jacob(3,1) ;
RESULTS.FK_J32(ii) = FK_Jacob(3,2) ;
RESULTS.FK_J33(ii) = FK_Jacob(3,3) ;
RESULTS.detJFK(ii) = det(FK_Jacob) ;

RESULTS.IK_J11(ii) = IK_Jacob(1,1) ;
RESULTS.IK_J12(ii) = IK_Jacob(1,2) ;
RESULTS.IK_J13(ii) = IK_Jacob(1,3) ;
RESULTS.IK_J21(ii) = IK_Jacob(2,1) ;
RESULTS.IK_J22(ii) = IK_Jacob(2,2) ;
RESULTS.IK_J23(ii) = IK_Jacob(2,3) ;
RESULTS.IK_J31(ii) = IK_Jacob(3,1) ;
RESULTS.IK_J32(ii) = IK_Jacob(3,2) ;
RESULTS.IK_J33(ii) = IK_Jacob(3,3) ;
RESULTS.detJIK(ii) = det(IK_Jacob) ;


% Drawing the mechanism
clf;
axis equal; grid; hold on ;
GraficRepresentation( RODS ) ;
plot(OP_ref(1)+incr_xp(1:ii), OP_ref(2)+incr_yp(1:ii), '.-m')
;
pause(1e-3) ;


% Creation of a .gif file
if graf_gif == 1

    drawnow
    % Capture the plot as an image
    frame = getframe(h);
    im = frame2im(frame);
    [imind,cm] = rgb2ind(im,256);

    % Write to the GIF File
    if ii == 1
        imwrite(imind,cm,filename,'gif', 'Loopcount',inf);
    else
        imwrite(imind,cm,filename,'gif','WriteMode','append');
    end

end


clc;
```

```
        fprintf('%f \n%%',(ii/Nii*100));

    end

end



matrix = [RESULTS.LAMBDA1 RESULTS.LAMBDA2 RESULTS.LAMBDA3 ] ;
xlRange = ['H2:J' num2str(Nii+1)] ;
xlswrite('IK_inputs_workspace.xlsx', matrix, xlRange) ;

matrix = [ RESULTS.FORCE1 RESULTS.FORCE2 RESULTS.FORCE3 ] ;
xlRange = ['K2:M' num2str(Nii+1)] ;
xlswrite('IK_inputs_workspace.xlsx', matrix, xlRange) ;

matrix = [ RESULTS.FK_J11 RESULTS.FK_J12 RESULTS.FK_J13 ...
           RESULTS.FK_J21 RESULTS.FK_J22 RESULTS.FK_J23 ...
           RESULTS.FK_J31 RESULTS.FK_J32 RESULTS.FK_J33 ] ;
xlRange = ['O2:W' num2str(Nii+1)] ;
xlswrite('IK_inputs_workspace.xlsx', matrix, xlRange) ;

xlRange = ['X2:X' num2str(Nii+1)] ;
xlswrite('IK_inputs_workspace.xlsx', RESULTS.detJFK, xlRange) ;

matrix = [ RESULTS.IK_J11 RESULTS.IK_J12 RESULTS.IK_J13 ...
           RESULTS.IK_J21 RESULTS.IK_J22 RESULTS.IK_J23 ...
           RESULTS.IK_J31 RESULTS.IK_J32 RESULTS.IK_J33 ] ;
xlRange = ['Z2:AH' num2str(Nii+1)] ;
xlswrite('IK_inputs_workspace.xlsx', matrix, xlRange) ;

xlRange = ['AI2:AI' num2str(Nii+1)] ;
xlswrite('IK_inputs_workspace.xlsx', RESULTS.detJIK, xlRange) ;




%%%%%%%%%%%%%%%%%%%%%%%%%%% PLOTTING RESULTS
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


data = xlsread('IK_inputs_workspace.xlsx') ;
Coord_x = OP_ref(1) + data(:,1);
Coord_y = OP_ref(2) + data(:,2);

[X, Y] = meshgrid(Coord_x, Coord_y);



% Lambda1, Lambda2, Lambda 3 values as functions of (x,y)
Lambda_1 = data(:,8);
Lambda_2 = data(:,9);
Lambda_3 = data(:,10);

LAMBDA1 = griddata(Coord_x, Coord_y, Lambda_1, X, Y) ;
LAMBDA2 = griddata(Coord_x, Coord_y, Lambda_2, X, Y) ;
LAMBDA3 = griddata(Coord_x, Coord_y, Lambda_3, X, Y) ;
```

```
fig1 = figure;
figure(fig1);

plot3(Coord_x, Coord_y, Lambda_1, '.r', 'MarkerSize', 7);
hold on; grid; daspect([2 2 1]);

plot3(Coord_x, Coord_y, Lambda_2, '.b', 'MarkerSize',7);

plot3(Coord_x, Coord_y, Lambda_3, '.g', 'MarkerSize',7);

mesh(X,Y,LAMBDA1, 'FaceColor', 'r', 'EdgeColor', 'r', 'FaceAlpha', 0.,
'EdgeAlpha', 0.2) ;
mesh(X,Y,LAMBDA2, 'FaceColor', 'b', 'EdgeColor', 'b', 'FaceAlpha', 0.,
'EdgeAlpha', 0.2) ;
mesh(X,Y,LAMBDA3, 'FaceColor', 'g', 'EdgeColor', 'g', 'FaceAlpha', 0.,
'EdgeAlpha', 0.2) ;

legend('Lambda 1', 'Lambda 2', 'Lambda 3');
xlabel('X Coordinate');
ylabel('Y Coordinate');
zlabel(['{\lambda}_' num2str(1) ', {\lambda}_' num2str(2) ',
{\lambda}_' num2str(3)]);


% |J_IK| as a function of (x,y)
det_JIK = data(:,35);
DJIK = griddata(Coord_x, Coord_y, det_JIK, X, Y) ;

fig2 = figure;
figure(fig2);

plot3(Coord_x, Coord_y, det_JIK, '.r', 'MarkerSize', 7);
grid;
xlabel('X Coordinate');
ylabel('Y Coordinate');
zlabel('|J_I_K|');

hold on;
mesh(X,Y,DJIK, 'FaceColor', 'r', 'EdgeColor', 'r', 'FaceAlpha', 0.3,
'EdgeAlpha', 0.3) ;


% Inverse problem singularities are marked
tol = 1e-4;
for jj = 1:length(det_JIK)
    if abs(det_JIK(jj)) <= tol
        plot3(Coord_x(jj), Coord_y(jj),
det_JIK(jj),'og','MarkerSize',7);
    end

end


% |J_FK| as a function of (x,y)
det_JFK = data(:,24);
DJFK = griddata(Coord_x, Coord_y, det_JFK, X, Y) ;
```

```
fig3 = figure;
figure(fig3);

plot3(Coord_x, Coord_y, det_JFK, '.g', 'MarkerSize', 7);
grid;
xlabel('X Coordinate');
ylabel('Y Coordinate');
zlabel('|J_F_K|');

hold on;
surf(X,Y,DJFK, 'FaceColor', 'g', 'EdgeColor', 'g', 'FaceAlpha', 0.3,
'EdgeAlpha', 0.3) ;


% Forward problem singularities are marked
tol = 1e-4;
for jj = 1:length(det_JFK)
    if abs(det_JFK(jj)) <= tol
        plot3(Coord_x(jj), Coord_y(jj),
det_JFK(jj),'ob','MarkerSize',7);
    end

end
```

4.2. INTEGRACIÓN NUMÉRICA. PROBLEMA DIRECTO

```
% Algorithm that solves de Forward Kinematic position problem through
the
% the use of the Shootin Method
clear; close all;

% Creation (1) or not(0) of a .gif file
graf_gif = 0 ;


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% INPUT DATA
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Loading data of the mechanism in the initial position
load('RODS');

% Data for the Inverse Kinematic position problem
data = xlsread('FK_inputs.xlsx') ;

incr_lambda1 = data(:,1) ;  % Increment of lambda1 respect to initial
position
                            % position
incr_lambda2 = data(:,2) ;  % Increment of lambda2 respect to initial
position
                            % position
incr_lambda3 = data(:,3) ;  % Increment of lambda2 respect to initial
position
                            % position
Fext     = data(:,4:5) ;    % External forces. The first column is the
                            % x component of the force and the second
                            % colum, the y component
Mext = data(:,6) ;          % External moment.


% Number of positions to be calculated
Nii = length(incr_lambda1) ;



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% OUTPUT DATA
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
RESULTS.XP     = zeros(Nii, 1) ;
RESULTS.YP     = zeros(Nii, 1) ;
RESULTS.ANGLE  = zeros(Nii, 1) ;
RESULTS.FORCE1 = zeros(Nii, 1) ;
RESULTS.FORCE2 = zeros(Nii, 1) ;
RESULTS.FORCE3 = zeros(Nii, 1) ;

RESULTS.FK_J11 = zeros(Nii, 1) ;
RESULTS.FK_J12 = zeros(Nii, 1) ;
RESULTS.FK_J13 = zeros(Nii, 1) ;
RESULTS.FK_J21 = zeros(Nii, 1) ;
RESULTS.FK_J22 = zeros(Nii, 1) ;
RESULTS.FK_J23 = zeros(Nii, 1) ;
RESULTS.FK_J31 = zeros(Nii, 1) ;
RESULTS.FK_J32 = zeros(Nii, 1) ;
RESULTS.FK_J33 = zeros(Nii, 1) ;
RESULTS.detJFK = zeros(Nii, 1) ;

RESULTS.IK_J11 = zeros(Nii, 1) ;
```

```
RESULTS.IK_J12 = zeros(Nii, 1) ;
RESULTS.IK_J13 = zeros(Nii, 1) ;
RESULTS.IK_J21 = zeros(Nii, 1) ;
RESULTS.IK_J22 = zeros(Nii, 1) ;
RESULTS.IK_J23 = zeros(Nii, 1) ;
RESULTS.IK_J31 = zeros(Nii, 1) ;
RESULTS.IK_J32 = zeros(Nii, 1) ;
RESULTS.IK_J33 = zeros(Nii, 1) ;
RESULTS.detJIK = zeros(Nii, 1) ;



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% CALCULATION
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
h = figure(1);
if graf_gif == 1
    axis tight manual
    filename = 'FK_testAnimated.gif';
end

for ii = 1:Nii

    % Update the position of the clamped end of the rod
    RODS.IN.B1.lambda = incr_lambda1(ii) ;
    RODS.IN.B2.lambda = incr_lambda2(ii) ;
    RODS.IN.B3.lambda = incr_lambda3(ii) ;

    % Shooting method
    RODS = FK_ShootingMethod( RODS, Fext(ii,:)', Mext(ii) ) ;

    % Forward Kinematic Jacobian matrix
    FK_Jacob = FK_Jacobian( RODS, Fext(ii,:)', Mext(ii) ) ;

    % Inverse Kinematic Jacobian matrix
    IK_Jacob = IK_Jacobian( RODS, Fext(ii,:)', Mext(ii) ) ;

    if RODS.OUT.sol == 0
        fprintf('Aborted calculation\n') ;
        break ;
    end

    % Storing results
    RESULTS.XP(ii)      = RODS.IN.OP(1) ;
    RESULTS.YP(ii)      = RODS.IN.OP(2) ;
    RESULTS.ANGLE(ii)   = RODS.IN.theta ;
    RESULTS.FORCE1(ii) = RODS.OUT.B1.Force ;
    RESULTS.FORCE2(ii) = RODS.OUT.B2.Force ;
    RESULTS.FORCE3(ii) = RODS.OUT.B3.Force ;

    RESULTS.FK_J11(ii) = FK_Jacob(1,1) ;
    RESULTS.FK_J12(ii) = FK_Jacob(1,2) ;
    RESULTS.FK_J13(ii) = FK_Jacob(1,3) ;
    RESULTS.FK_J21(ii) = FK_Jacob(2,1) ;
    RESULTS.FK_J22(ii) = FK_Jacob(2,2) ;
    RESULTS.FK_J23(ii) = FK_Jacob(2,3) ;
    RESULTS.FK_J31(ii) = FK_Jacob(3,1) ;
    RESULTS.FK_J32(ii) = FK_Jacob(3,2) ;
    RESULTS.FK_J33(ii) = FK_Jacob(3,3) ;
    RESULTS.detJFK(ii) = det(FK_Jacob) ;
```

```
    RESULTS.IK_J11(ii) = IK_Jacob(1,1) ;
    RESULTS.IK_J12(ii) = IK_Jacob(1,2) ;
    RESULTS.IK_J13(ii) = IK_Jacob(1,3) ;
    RESULTS.IK_J21(ii) = IK_Jacob(2,1) ;
    RESULTS.IK_J22(ii) = IK_Jacob(2,2) ;
    RESULTS.IK_J23(ii) = IK_Jacob(2,3) ;
    RESULTS.IK_J31(ii) = IK_Jacob(3,1) ;
    RESULTS.IK_J32(ii) = IK_Jacob(3,2) ;
    RESULTS.IK_J33(ii) = IK_Jacob(3,3) ;
    RESULTS.detJIK(ii) = det(IK_Jacob) ;

    % Drawing the mechanism
    clf;
    axis equal; grid; hold on ;
    GraficRepresentation( RODS ) ;
    plot(RESULTS.XP(1:ii), RESULTS.YP(1:ii), '.-m') ;
    pause(1e-3) ;


    % Creation of a .gif file
    if graf_gif == 1

        drawnow
        % Capture the plot as an image
        frame = getframe(h);
        im = frame2im(frame);
        [imind,cm] = rgb2ind(im,256);

        % Write to the GIF File
        if ii == 1
            imwrite(imind,cm,filename,'gif', 'Loopcount',inf);
        else
            imwrite(imind,cm,filename,'gif','WriteMode','append');
        end

    end


    clc;
    fprintf('%f \n%%',(ii/Nii*100));

end



matrix = [RESULTS.XP RESULTS.YP RESULTS.ANGLE ] ;
xlRange = ['H2:J' num2str(Nii+1)] ;
xlswrite('FK_inputs.xlsx', matrix, xlRange) ;

matrix = [ RESULTS.FORCE1 RESULTS.FORCE2 RESULTS.FORCE3 ] ;
xlRange = ['K2:M' num2str(Nii+1)] ;
xlswrite('FK_inputs.xlsx', matrix, xlRange) ;

matrix = [ RESULTS.FK_J11 RESULTS.FK_J12 RESULTS.FK_J13 ...
           RESULTS.FK_J21 RESULTS.FK_J22 RESULTS.FK_J23 ...
           RESULTS.FK_J31 RESULTS.FK_J32 RESULTS.FK_J33 ] ;
xlRange = ['O2:W' num2str(Nii+1)] ;
```

```
xlswrite('FK_inputs.xlsx', matrix, xlRange) ;

xlRange = ['X2:X' num2str(Nii+1)] ;
xlswrite('FK_inputs.xlsx', RESULTS.detJFK, xlRange) ;

matrix = [ RESULTS.IK_J11 RESULTS.IK_J12 RESULTS.IK_J13 ...
           RESULTS.IK_J21 RESULTS.IK_J22 RESULTS.IK_J23 ...
           RESULTS.IK_J31 RESULTS.IK_J32 RESULTS.IK_J33 ] ;
xlRange = ['Z2:AH' num2str(Nii+1)] ;
xlswrite('FK_inputs.xlsx', matrix, xlRange) ;

xlRange = ['AI2:AI' num2str(Nii+1)] ;
xlswrite('FK_inputs.xlsx', RESULTS.detJIK, xlRange) ;
```

4.3. INTEGRACIÓN NUMÉRICA. FUNCIONES ASOCIADAS

```
function [ Jacob ] = FK_Jacobian( RODS, Fext, Mext )
% Calculation of the Jacobian for the FK position problem
Jacob = zeros(3) ;

RODS_ref = RODS ;

% Increment of value of each lambda to calculate nummerically the
Jacobian
epsilon = 1e-6 ;

% Increment of lambda 1
RODS.IN.B1.lambda = RODS.IN.B1.lambda + epsilon ;
RODS = FK_ShootingMethod( RODS, Fext, Mext ) ;
if RODS.OUT.sol == 1
    Jacob(:,1) = [ RODS.IN.OP ;
                   RODS.IN.theta ] - ...
                 [ RODS_ref.IN.OP ;
                   RODS_ref.IN.theta ] ;
    RODS.IN.B1.lambda = RODS.IN.B1.lambda - epsilon ;
else
    Jacob(:,1) = NaN ;
end


% Increment of lambda 2
RODS.IN.B2.lambda = RODS.IN.B2.lambda + epsilon ;
RODS = FK_ShootingMethod( RODS, Fext, Mext ) ;
if RODS.OUT.sol == 1
    Jacob(:,2) = [ RODS.IN.OP ;
                   RODS.IN.theta ] - ...
                 [ RODS_ref.IN.OP ;
                   RODS_ref.IN.theta ] ;
    RODS.IN.B2.lambda = RODS.IN.B2.lambda - epsilon ;
else
    Jacob(:,2) = NaN ;
end


% Increment of lambda 3
RODS.IN.B3.lambda = RODS.IN.B3.lambda + epsilon ;
RODS = FK_ShootingMethod( RODS, Fext, Mext ) ;
if RODS.OUT.sol == 1
    Jacob(:,3) = [ RODS.IN.OP ;
                   RODS.IN.theta ] - ...
                 [ RODS_ref.IN.OP ;
                   RODS_ref.IN.theta ] ;
    RODS.IN.B3.lambda = RODS.IN.B3.lambda - epsilon ;
else
    Jacob(:,3) = NaN ;
end


Jacob = Jacob/epsilon ;

end


function [ RODS ] = FK_ShootingMethod( RODS, Fext, Mext )
```

```
% Shooting method to solve the Forward Kinematic position problem

% Absolute and relative tolerance of the element of the residue vector
tol = 1e-8 ;


% Runge-Kuta method of order 4 for each rod
[ RODS.OUT.B1 ] = RK4_rod( RODS.IN.B1 ) ;
[ RODS.OUT.B2 ] = RK4_rod( RODS.IN.B2 ) ;
[ RODS.OUT.B3 ] = RK4_rod( RODS.IN.B3 ) ;

% Update the new guess position at pinned end (known value)
R = [ cos(RODS.IN.theta) -sin(RODS.IN.theta) ;
      sin(RODS.IN.theta)  cos(RODS.IN.theta) ] ;

RODS.IN.B1.pL = R*RODS.IN.B1.r + RODS.IN.OP ;
RODS.IN.B2.pL = R*RODS.IN.B2.r + RODS.IN.OP ;
RODS.IN.B3.pL = R*RODS.IN.B3.r + RODS.IN.OP ;

% Initial residue vector
resid = Residue( RODS, Fext, Mext ) ;

% Residue vector of the former iteration. Before the iteration (while
% loop), this residue vector is null
resid0 = zeros(size(resid)) ;


while max(abs(resid)) > tol && max(abs(resid-resid0)) > tol

    resid0 = resid ;

    % Jacobian of the residue vector respect the variables that acts
as
    % guess values
    J = Jacobian( RODS, Fext ) ;

    % Increment of the guess values
    delta = -J\resid0 ;

    % Updating guess values
    RODS.IN.B1.m0 = RODS.IN.B1.m0 + delta(1) ;
    RODS.IN.B1.n0 = RODS.IN.B1.n0 + delta(2:3) ;
    RODS.IN.B2.m0 = RODS.IN.B2.m0 + delta(4) ;
    RODS.IN.B2.n0 = RODS.IN.B2.n0 + delta(5:6) ;
    RODS.IN.B3.m0 = RODS.IN.B3.m0 + delta(7) ;
    RODS.IN.B3.n0 = RODS.IN.B3.n0 + delta(8:9) ;
    RODS.IN.OP    = RODS.IN.OP    + delta(10:11) ;
    RODS.IN.theta = RODS.IN.theta + delta(12) ;

    % Runge-Kuta method of order 4 for each rod with the updated guess
    % values
    [ RODS.OUT.B1 ] = RK4_rod( RODS.IN.B1 ) ;
    [ RODS.OUT.B2 ] = RK4_rod( RODS.IN.B2 ) ;
    [ RODS.OUT.B3 ] = RK4_rod( RODS.IN.B3 ) ;

    % Update the new guess position at pinned end (known value)
    R = [ cos(RODS.IN.theta) -sin(RODS.IN.theta) ;
          sin(RODS.IN.theta)  cos(RODS.IN.theta) ] ;
```

```
    RODS.IN.B1.pL = R*RODS.IN.B1.r + RODS.IN.OP ;
    RODS.IN.B2.pL = R*RODS.IN.B2.r + RODS.IN.OP ;
    RODS.IN.B3.pL = R*RODS.IN.B3.r + RODS.IN.OP ;

    % New resude vector
    resid = Residue( RODS, Fext, Mext ) ;

    % In case in which the new residue vector is bigger that the
previous
    % one, the incrementation of the guess values are forced to take
half
    % of value calculated. This operation is applied iteratively until
the
    % new residue vector is smaller that the previous one
    while max(abs(resid)) > max(abs(resid0)) &&
max(abs(delta))>tol*1e-6
        delta = 0.5*delta ;

        RODS.IN.B1.m0 = RODS.IN.B1.m0 - delta(1) ;
        RODS.IN.B1.n0 = RODS.IN.B1.n0 - delta(2:3) ;
        RODS.IN.B2.m0 = RODS.IN.B2.m0 - delta(4) ;
        RODS.IN.B2.n0 = RODS.IN.B2.n0 - delta(5:6) ;
        RODS.IN.B3.m0 = RODS.IN.B3.m0 - delta(7) ;
        RODS.IN.B3.n0 = RODS.IN.B3.n0 - delta(8:9) ;
        RODS.IN.OP    = RODS.IN.OP    - delta(10:11) ;
        RODS.IN.theta = RODS.IN.theta - delta(12) ;

        [ RODS.OUT.B1 ] = RK4_rod( RODS.IN.B1 ) ;
        [ RODS.OUT.B2 ] = RK4_rod( RODS.IN.B2 ) ;
        [ RODS.OUT.B3 ] = RK4_rod( RODS.IN.B3 ) ;

        % Update the new guess position at pinned end (known value)
        R = [ cos(RODS.IN.theta) -sin(RODS.IN.theta) ;
              sin(RODS.IN.theta)  cos(RODS.IN.theta) ] ;

        RODS.IN.B1.pL = R*RODS.IN.B1.r + RODS.IN.OP ;
        RODS.IN.B2.pL = R*RODS.IN.B2.r + RODS.IN.OP ;
        RODS.IN.B3.pL = R*RODS.IN.B3.r + RODS.IN.OP ;

        resid = Residue( RODS, Fext, Mext ) ;
    end

end


% If the solution has not been reached or the reside is NaN, the
variable
% "sol" take the value 0
if max(abs(resid)) > tol || not(sum(abs(resid))>=0)
    RODS.OUT.sol = 0 ;
else
    RODS.OUT.sol = 1 ;

    % Froce applied at clamped end projected in the direction of each
    % linear guide
    RODS.OUT.B1.Force = cos(RODS.IN.B1.alpha)*RODS.IN.B1.n0(1) + ...
                        sin(RODS.IN.B1.alpha)*RODS.IN.B1.n0(2) ;
    RODS.OUT.B2.Force = cos(RODS.IN.B2.alpha)*RODS.IN.B2.n0(1) + ...
```

```
                    sin(RODS.IN.B2.alpha)*RODS.IN.B2.n0(2) ;
    RODS.OUT.B3.Force = cos(RODS.IN.B3.alpha)*RODS.IN.B3.n0(1) + ...
                    sin(RODS.IN.B3.alpha)*RODS.IN.B3.n0(2) ;

end




function [ Jacob ] = Jacobian( RODS, Fext )
% Functions that calculates numerically the Jacobian of the residue
vector
% respect to the variables of the problem (guess values)
Jacob = zeros(12) ;


% Increment of the variables (guess values) to calculate numerically
the
% Jacobian
epsilon = 1e-11 ;


% Variables for rod 1
RODS.IN.B1.m0 = RODS.IN.B1.m0 + epsilon ;
OUT_aux = RK4_rod( RODS.IN.B1 ) ;
Jacob([1:3 12],1) = [ OUT_aux.m(end) ;
                      OUT_aux.p(:,end) ;
                      OUT_aux.m(end) + OUT_aux.pxn] - ...
                    [ RODS.OUT.B1.m(end) ;
                      RODS.OUT.B1.p(:,end) ;
                      RODS.OUT.B1.m(end) + RODS.OUT.B1.pxn ] ;
Jacob([1:3 12],1) = Jacob([1:3 12],1)/epsilon ;
RODS.IN.B1.m0 = RODS.IN.B1.m0 - epsilon ;

RODS.IN.B1.n0(1) = RODS.IN.B1.n0(1) + epsilon ;
OUT_aux = RK4_rod( RODS.IN.B1 ) ;
Jacob([1:3 12],2) = [ OUT_aux.m(end) ;
                      OUT_aux.p(:,end) ;
                      OUT_aux.m(end) + OUT_aux.pxn] - ...
                    [ RODS.OUT.B1.m(end) ;
                      RODS.OUT.B1.p(:,end) ;
                      RODS.OUT.B1.m(end) + RODS.OUT.B1.pxn ] ;
Jacob([1:3 12],2) = Jacob([1:3 12],2)/epsilon ;
RODS.IN.B1.n0(1) = RODS.IN.B1.n0(1) - epsilon ;

RODS.IN.B1.n0(2) = RODS.IN.B1.n0(2) + epsilon ;
OUT_aux = RK4_rod( RODS.IN.B1 ) ;
Jacob([1:3 12],3) = [ OUT_aux.m(end) ;
                      OUT_aux.p(:,end) ;
                      OUT_aux.m(end) + OUT_aux.pxn] - ...
                    [ RODS.OUT.B1.m(end) ;
                      RODS.OUT.B1.p(:,end) ;
                      RODS.OUT.B1.m(end) + RODS.OUT.B1.pxn ] ;
Jacob([1:3 12],3) = Jacob([1:3 12],3)/epsilon ;
% RODS.IN.B1.n0(2) = RODS.IN.B1.n0(2) - epsilon ;

Jacob(2:3,12) = [  sin(RODS.IN.theta)*RODS.IN.B1.r(1) + ...
                   cos(RODS.IN.theta)*RODS.IN.B1.r(2) ;
                  -cos(RODS.IN.theta)*RODS.IN.B1.r(1) + ...
```

```
                    sin(RODS.IN.theta)*RODS.IN.B1.r(2) ] ;


% Variables for rod 2
RODS.IN.B2.m0 = RODS.IN.B2.m0 + epsilon ;
OUT_aux = RK4_rod( RODS.IN.B2 ) ;
Jacob([4:6 12],4) = [ OUT_aux.m(end) ;
                      OUT_aux.p(:,end) ;
                      OUT_aux.m(end) + OUT_aux.pxn] - ...
                    [ RODS.OUT.B2.m(end) ;
                      RODS.OUT.B2.p(:,end) ;
                      RODS.OUT.B2.m(end) + RODS.OUT.B2.pxn ] ;
Jacob([4:6 12],4) = Jacob([4:6 12],4)/epsilon ;
RODS.IN.B2.m0 = RODS.IN.B2.m0 - epsilon ;

RODS.IN.B2.n0(1) = RODS.IN.B2.n0(1) + epsilon ;
OUT_aux = RK4_rod( RODS.IN.B2 ) ;
Jacob([4:6 12],5) = [ OUT_aux.m(end) ;
                      OUT_aux.p(:,end) ;
                      OUT_aux.m(end) + OUT_aux.pxn] - ...
                    [ RODS.OUT.B2.m(end) ;
                      RODS.OUT.B2.p(:,end) ;
                      RODS.OUT.B2.m(end) + RODS.OUT.B2.pxn ] ;
Jacob([4:6 12],5) = Jacob([4:6 12],5)/epsilon ;
RODS.IN.B2.n0(1) = RODS.IN.B2.n0(1) - epsilon ;

RODS.IN.B2.n0(2) = RODS.IN.B2.n0(2) + epsilon ;
OUT_aux = RK4_rod( RODS.IN.B2 ) ;
Jacob([4:6 12],6) = [ OUT_aux.m(end) ;
                      OUT_aux.p(:,end) ;
                      OUT_aux.m(end) + OUT_aux.pxn] - ...
                    [ RODS.OUT.B2.m(end) ;
                      RODS.OUT.B2.p(:,end) ;
                      RODS.OUT.B2.m(end) + RODS.OUT.B2.pxn ] ;
Jacob([4:6 12],6) = Jacob([4:6 12],6)/epsilon ;
% RODS.IN.B2.n0(2) = RODS.IN.B2.n0(2) - epsilon ;

Jacob(5:6,12) = [  sin(RODS.IN.theta)*RODS.IN.B2.r(1) + ...
                   cos(RODS.IN.theta)*RODS.IN.B2.r(2) ;
                  -cos(RODS.IN.theta)*RODS.IN.B2.r(1) + ...
                   sin(RODS.IN.theta)*RODS.IN.B2.r(2) ] ;


% Variables for rod 3
RODS.IN.B3.m0 = RODS.IN.B3.m0 + epsilon ;
OUT_aux = RK4_rod( RODS.IN.B3 ) ;
Jacob([7:9 12],7) = [ OUT_aux.m(end) ;
                      OUT_aux.p(:,end) ;
                      OUT_aux.m(end) + OUT_aux.pxn] - ...
                    [ RODS.OUT.B3.m(end) ;
                      RODS.OUT.B3.p(:,end) ;
                      RODS.OUT.B3.m(end) + RODS.OUT.B3.pxn ] ;
Jacob([7:9 12],7) = Jacob([7:9 12],7)/epsilon ;
RODS.IN.B3.m0 = RODS.IN.B3.m0 - epsilon ;

RODS.IN.B3.n0(1) = RODS.IN.B3.n0(1) + epsilon ;
OUT_aux = RK4_rod( RODS.IN.B3 ) ;
Jacob([7:9 12],8) = [ OUT_aux.m(end) ;
                      OUT_aux.p(:,end) ;
                      OUT_aux.m(end) + OUT_aux.pxn] - ...
```

```
                    [ RODS.OUT.B3.m(end) ;
                      RODS.OUT.B3.p(:,end) ;
                      RODS.OUT.B3.m(end) + RODS.OUT.B3.pxn ] ;
Jacob([7:9 12],8) = Jacob([7:9 12],8)/epsilon ;
RODS.IN.B3.n0(1) = RODS.IN.B3.n0(1) - epsilon ;

RODS.IN.B3.n0(2) = RODS.IN.B3.n0(2) + epsilon ;
OUT_aux = RK4_rod( RODS.IN.B3 ) ;
Jacob([7:9 12],9) = [ OUT_aux.m(end) ;
                      OUT_aux.p(:,end) ;
                      OUT_aux.m(end) + OUT_aux.pxn] - ...
                    [ RODS.OUT.B3.m(end) ;
                      RODS.OUT.B3.p(:,end) ;
                      RODS.OUT.B3.m(end) + RODS.OUT.B3.pxn ] ;
Jacob([7:9 12],9) = Jacob([7:9 12],9)/epsilon ;
% RODS.IN.B3.n0(2) = RODS.IN.B3.n0(2) - epsilon ;

Jacob(8:9,12) = [  sin(RODS.IN.theta)*RODS.IN.B3.r(1) + ...
                   cos(RODS.IN.theta)*RODS.IN.B3.r(2) ;
                  -cos(RODS.IN.theta)*RODS.IN.B3.r(1) + ...
                   sin(RODS.IN.theta)*RODS.IN.B3.r(2) ] ;


% Variables of the position and orientation of the end effecto
plattform
Jacob(2:3:9,10) = -1 ;
Jacob(3:3:9,11) = -1 ;
Jacob(12,10) = -Fext(2) ;
Jacob(12,11) =  Fext(1) ;

% Unit elements
Jacob(10,2:3:9) = 1 ;
Jacob(11,3:3:9) = 1 ;




function [ resid ] = Residue( RODS, Fext, Mext )
% Function of the residue vector of the whole mechanism

resid = [ RODS.OUT.B1.m(end) ;
          RODS.OUT.B1.p(:,end) - RODS.IN.B1.pL ;
          RODS.OUT.B2.m(end) ;
          RODS.OUT.B2.p(:,end) - RODS.IN.B2.pL ;
          RODS.OUT.B3.m(end) ;
          RODS.OUT.B3.p(:,end) - RODS.IN.B3.pL ;
          RODS.IN.B1.n0 + RODS.IN.B2.n0 + RODS.IN.B3.n0 - Fext ;
          RODS.OUT.B1.m(end) + RODS.OUT.B2.m(end) + RODS.OUT.B3.m(end)
+ ...
          RODS.OUT.B1.pxn + RODS.OUT.B2.pxn + RODS.OUT.B3.pxn + ...
          - RODS.IN.OP(1)*Fext(2)+RODS.IN.OP(2)*Fext(1) - Mext] ;




function [ OUT ] = RK4_rod ( IN )
% Functions that solves de initial value problem using the method of
% Runge-Kutta of order four.
%   IN:  Structure in which are geometric and mechanical parameters
```

```
%       IN.L        Length of the rod [m]
%       IN.N        Number of nodes of the rod
%       IN.EI       Bending stiffness of the rod [Pa]
%       IN.d        Distance of the rigid part located in pinned end
of the
%                   rod [m]
%       IN.alpha    Angle of the guide [rad]
%       In.theta0   Angle of the rod at clamped end [rad]
%       IN.lambda   Value of lambda [m]
%       IN.m0       Bending at the clamped end [N.m]
%       IN.n0       Internal force at the clamped end (2x1 vector) [N]

%   OUT:    Structure in which are saved the solution
%       OUT.p       array (2xIN.N) with the x,y component of the
%                   centroid position of the rod
%       OUT.m       array (1xIN.N) with the bending moment in each
node
%                   of the rod
%       OUT.theta   array (1xIN.N) with the angle of the tangent of
%                   the rod
%       OUT.Ener    Elastic deformation enegy of the rod using the
Simpson
%                   rule


% Increment of length
ds = IN.L/(IN.N-1) ;

%%%%%%%%%%%%%%%%%%%%% Variables to save the solution
%%%%%%%%%%%%%%%%%%%%%
OUT.p     = zeros(2,IN.N+1) ;   % Centroid position vector function
(x,y)
OUT.theta = zeros(1,IN.N+1) ;   % Angle function of the tangent of the
rod
OUT.m     = zeros(1,IN.N+1) ;   % Bending moment function

% Variables at the first point (clamped end)
OUT.p(:,1)    = IN.OA + IN.lambda*[cos(IN.alpha); sin(IN.alpha)];
OUT.theta(1) = IN.theta0 ;
OUT.m(1)     = IN.m0 ;

% Vector with the values of the dependent variables at the clamped end
var = [ OUT.p(:,1) ;
        OUT.theta(1) ;
        OUT.m(1) ] ;

% "For" loop to integrate the system of differential equations through
the
% rod
for ii = 2:IN.N

    % Runge-Kutta method of order four
    k1 = ds *  Right_function( IN.EI, IN.n0, var ) ;
    k2 = ds *  Right_function( IN.EI, IN.n0, var + 0.5*k1 ) ;
    k3 = ds *  Right_function( IN.EI, IN.n0, var + 0.5*k2 ) ;
    k4 = ds *  Right_function( IN.EI, IN.n0, var + k3 ) ;

    var = var + (k1 + 2*k2 + 2*k3 + k4)/6 ;
```

```
    OUT.p(:,ii)    = var(1:2) ;
    OUT.theta(ii) = var(3) ;
    OUT.m(ii)      = var(4) ;

end


% Results variables at the pinned end (rigid part)
OUT.p(:,end)    = OUT.p(:,end-1) + ...
                  IN.d*[cos(OUT.theta(end-1)); sin(OUT.theta(end-1))] ;
OUT.theta(end) = OUT.theta(end-1) ;
OUT.m(end)      = OUT.m(end-1) + (OUT.p(1,end-1)-OUT.p(1,end))*IN.n0(2)
-...
                  (OUT.p(2,end-1)-OUT.p(2,end))*IN.n0(1)   ;


% Vector product of pinned end position and force applied on it
OUT.pxn = OUT.p(1,end)*IN.n0(2)-OUT.p(2,end)*IN.n0(1) ;


% Elastic energy calculated throught the the Simpson's rule
m2 = OUT.m(1:end-1).^2 ;          % Square of the internal moemnt
funciton

OUT.Ener = ds/3 * ( m2(1) + m2(end) + 4*sum(m2(2:2:(IN.N-1))) + ...
            2*sum(m2(3:2:(IN.N-2))) ) ;
OUT.Ener = OUT.Ener/IN.EI ;




function [ func ] = Right_function( EI, n, var )
% Value of the right part of the system of differential equations
% var(1)     px
% var(2)     py
% var(3)     theta
% var(4)     moment

func = [ cos(var(3)) ;
         sin(var(3)) ;
         var(4)/EI ;
         n(1)*sin(var(3)) - n(2)*cos(var(3)) ] ;




function [  ] = GraficRepresentation( RODS )
% Plotting each rod and end effector plattform of the mechanism

% Rod 1
plot(RODS.OUT.B1.p(1,1:end-1),RODS.OUT.B1.p(2,1:end-1), '.-', 'color',
[0 0.6275 0]) ;
plot(RODS.OUT.B1.p(1,end-1:end),RODS.OUT.B1.p(2,end-1:end), '-',
'color', [0 0.6275 0]) ;

% Rod 2
plot(RODS.OUT.B2.p(1,1:end-1),RODS.OUT.B2.p(2,1:end-1), '.-', 'color',
[0 0 0.6275]) ;
```

```
plot(RODS.OUT.B2.p(1,end-1:end),RODS.OUT.B2.p(2,end-1:end),'-',
'color', [0 0 0.6275]) ;

% Rod 3
plot(RODS.OUT.B3.p(1,1:end-1),RODS.OUT.B3.p(2,1:end-1),'.-', 'color',
[0.6275 0 0]) ;
plot(RODS.OUT.B3.p(1,end-1:end),RODS.OUT.B3.p(2,end-1:end),'-',
'color', [0.6275 0 0]) ;

% End effector plattform
x_vert = [ RODS.IN.B1.pL(1) RODS.IN.B2.pL(1) RODS.IN.B3.pL(1)
RODS.IN.B1.pL(1) ] ;
y_vert = [ RODS.IN.B1.pL(2) RODS.IN.B2.pL(2) RODS.IN.B3.pL(2)
RODS.IN.B1.pL(2) ] ;
plot(x_vert, y_vert, 'k', 'linewidth', 2) ;

% % End effector orientation
% const = norm(RODS.IN.B1.r) ;
% quiver( RODS.IN.OP(1), RODS.IN.OP(2), const*cos(RODS.IN.theta),
const*sin(RODS.IN.theta), 'r' ) ;
% quiver( RODS.IN.OP(1), RODS.IN.OP(2), -const*sin(RODS.IN.theta),
const*cos(RODS.IN.theta), 'g' ) ;

% Value of lambda
x1_guide = [ 0 RODS.IN.B1.lambda*cos(RODS.IN.B1.alpha) ] +
RODS.IN.B1.OA(1) ;
y1_guide = [ 0 RODS.IN.B1.lambda*sin(RODS.IN.B1.alpha) ] +
RODS.IN.B1.OA(2) ;
plot(x1_guide, y1_guide, '-', 'color', [0 0.6275 0], 'linewidth', 4) ;

x2_guide = [ 0 RODS.IN.B2.lambda*cos(RODS.IN.B2.alpha) ] +
RODS.IN.B2.OA(1) ;
y2_guide = [ 0 RODS.IN.B2.lambda*sin(RODS.IN.B2.alpha) ] +
RODS.IN.B2.OA(2) ;
plot(x2_guide, y2_guide, '-', 'color', [0 0 0.6275], 'linewidth', 4) ;

x3_guide = [ 0 RODS.IN.B3.lambda*cos(RODS.IN.B3.alpha) ] +
RODS.IN.B3.OA(1) ;
y3_guide = [ 0 RODS.IN.B3.lambda*sin(RODS.IN.B3.alpha) ] +
RODS.IN.B3.OA(2) ;
plot(x3_guide, y3_guide, '-', 'color', [0.6275 0 0], 'linewidth', 4) ;

end


function [ Jacob ] = IK_Jacobian( RODS, Fext, Mext )
% Calculation of the Jacobian for the FK position problem
Jacob = zeros(3) ;

RODS_ref = RODS ;

% Increment of value of each lambda to calculate nummerically the
Jacobian
epsilon = 1e-6 ;

% Increment of OP, x component
RODS.IN.OP(1) = RODS.IN.OP(1) + epsilon ;
RODS = IK_ShootingMethod( RODS, Fext, Mext ) ;
if RODS.OUT.sol == 1
```

```
    Jacob(:,1) = [ RODS.IN.B1.lambda ;
                   RODS.IN.B2.lambda;
                   RODS.IN.B3.lambda ] - ...
                 [ RODS_ref.IN.B1.lambda ;
                   RODS_ref.IN.B2.lambda;
                   RODS_ref.IN.B3.lambda ] ;
    RODS.IN.OP(1) = RODS.IN.OP(1) - epsilon ;
else
    Jacob(:,1) = NaN ;
end

% Increment of OP, y component
RODS.IN.OP(2) = RODS.IN.OP(2) + epsilon ;
RODS = IK_ShootingMethod( RODS, Fext, Mext ) ;
if RODS.OUT.sol == 1
    Jacob(:,2) = [ RODS.IN.B1.lambda ;
                   RODS.IN.B2.lambda;
                   RODS.IN.B3.lambda ] - ...
                 [ RODS_ref.IN.B1.lambda ;
                   RODS_ref.IN.B2.lambda;
                   RODS_ref.IN.B3.lambda ] ;
    RODS.IN.OP(2) = RODS.IN.OP(2) - epsilon ;
else
    Jacob(:,2) = NaN ;
end

% Increment of the angle of end effector platform orientation
RODS.IN.theta = RODS.IN.theta + epsilon ;
RODS = IK_ShootingMethod( RODS, Fext, Mext ) ;
if RODS.OUT.sol == 1
    Jacob(:,3) = [ RODS.IN.B1.lambda ;
                   RODS.IN.B2.lambda;
                   RODS.IN.B3.lambda ] - ...
                 [ RODS_ref.IN.B1.lambda ;
                   RODS_ref.IN.B2.lambda;
                   RODS_ref.IN.B3.lambda ] ;
else
    Jacob(:,3) = NaN ;
end


Jacob = Jacob/epsilon ;

end


function [ RODS ] = IK_ShootingMethod( RODS, Fext, Mext )
% Shooting method to solve the Inverse Kinematic position problem

% Absolute and relative tolerance of the element of the residue vector
tol = 1e-8 ;


% Update de new position at pinned end (known value)
R = [ cos(RODS.IN.theta) -sin(RODS.IN.theta) ;
      sin(RODS.IN.theta)  cos(RODS.IN.theta) ] ;

RODS.IN.B1.pL = R*RODS.IN.B1.r + RODS.IN.OP ;
RODS.IN.B2.pL = R*RODS.IN.B2.r + RODS.IN.OP ;
```

```
RODS.IN.B3.pL = R*RODS.IN.B3.r + RODS.IN.OP ;

% Runge-Kuta method of order 4 for each rod
[ RODS.OUT.B1 ] = RK4_rod( RODS.IN.B1 ) ;
[ RODS.OUT.B2 ] = RK4_rod( RODS.IN.B2 ) ;
[ RODS.OUT.B3 ] = RK4_rod( RODS.IN.B3 ) ;

% Initial residue vector
resid = Residue( RODS, Fext, Mext ) ;

% Residue vector of the former iteration. Before the iteration (while
% loop), this residue vector is null
resid0 = zeros(size(resid)) ;

while max(abs(resid)) > tol && max(abs(resid-resid0)) > tol

    resid0 = resid ;

    % Jacobian of the residue vector respect the variables that acts
as
    % guess values
    J = Jacobian( RODS ) ;

    % Increment of the guess values
    delta = -J\resid0 ;

    % Updating guess values
    RODS.IN.B1.m0     = RODS.IN.B1.m0     + delta(1) ;
    RODS.IN.B1.n0     = RODS.IN.B1.n0     + delta(2:3) ;
    RODS.IN.B1.lambda = RODS.IN.B1.lambda + delta(4) ;
    RODS.IN.B2.m0     = RODS.IN.B2.m0     + delta(5) ;
    RODS.IN.B2.n0     = RODS.IN.B2.n0     + delta(6:7) ;
    RODS.IN.B2.lambda = RODS.IN.B2.lambda + delta(8) ;
    RODS.IN.B3.m0     = RODS.IN.B3.m0     + delta(9) ;
    RODS.IN.B3.n0     = RODS.IN.B3.n0     + delta(10:11) ;
    RODS.IN.B3.lambda = RODS.IN.B3.lambda + delta(12) ;

    % Runge-Kuta method of order 4 for each rod with the updated guess
    % values
    [ RODS.OUT.B1 ] = RK4_rod( RODS.IN.B1 ) ;
    [ RODS.OUT.B2 ] = RK4_rod( RODS.IN.B2 ) ;
    [ RODS.OUT.B3 ] = RK4_rod( RODS.IN.B3 ) ;


    % New resude vector
    resid = Residue( RODS, Fext, Mext ) ;

    % In case in which the new residue vector is bigger that the
previous
    % one, the incrementation of the guess values are forced to take
half
    % of value calculated. This operation is applied iteratively until
the
    % new residue vector is smaller that the previous one
    while max(abs(resid)) > max(abs(resid0)) &&
max(abs(delta))>tol*1e-6
        delta = 0.5*delta ;

        RODS.IN.B1.m0     = RODS.IN.B1.m0     - delta(1) ;
```

```
        RODS.IN.B1.n0      = RODS.IN.B1.n0      - delta(2:3) ;
        RODS.IN.B1.lambda = RODS.IN.B1.lambda - delta(4) ;
        RODS.IN.B2.m0      = RODS.IN.B2.m0      - delta(5) ;
        RODS.IN.B2.n0      = RODS.IN.B2.n0      - delta(6:7) ;
        RODS.IN.B2.lambda = RODS.IN.B2.lambda - delta(8) ;
        RODS.IN.B3.m0      = RODS.IN.B3.m0      - delta(9) ;
        RODS.IN.B3.n0      = RODS.IN.B3.n0      - delta(10:11) ;
        RODS.IN.B3.lambda = RODS.IN.B3.lambda - delta(12) ;

        [ RODS.OUT.B1 ] = RK4_rod( RODS.IN.B1 ) ;
        [ RODS.OUT.B2 ] = RK4_rod( RODS.IN.B2 ) ;
        [ RODS.OUT.B3 ] = RK4_rod( RODS.IN.B3 ) ;
        resid = Residue( RODS, Fext, Mext ) ;

    end

end


% If the solution has not been reached or the reside is NaN, the
variable
% "sol" take the value 0
if max(abs(resid)) > tol || not(sum(abs(resid))>=0)
    RODS.OUT.sol = 0 ;
else
    RODS.OUT.sol = 1 ;

    % Froce applied at clamped end projected in the direction of each
    % linear guide
    RODS.OUT.B1.Force = cos(RODS.IN.B1.alpha)*RODS.IN.B1.n0(1) + ...
                        sin(RODS.IN.B1.alpha)*RODS.IN.B1.n0(2) ;
    RODS.OUT.B2.Force = cos(RODS.IN.B2.alpha)*RODS.IN.B2.n0(1) + ...
                        sin(RODS.IN.B2.alpha)*RODS.IN.B2.n0(2) ;
    RODS.OUT.B3.Force = cos(RODS.IN.B3.alpha)*RODS.IN.B3.n0(1) + ...
                        sin(RODS.IN.B3.alpha)*RODS.IN.B3.n0(2) ;

end




function [ Jacob ] = Jacobian( RODS )
% Functions that calculates numerically the Jacobian of the residue
vector
% respect to the variables of the problem (guess values)
Jacob = zeros(12) ;


% Increment of the variables (guess values) to calculate numerically
the
% Jacobian
epsilon = 1e-11 ;


% Variables for rod 1
RODS.IN.B1.m0 = RODS.IN.B1.m0 + epsilon ;
OUT_aux = RK4_rod( RODS.IN.B1 ) ;
Jacob([1:3 12],1) = [ OUT_aux.m(end) ;
```

```
                          OUT_aux.p(:,end) ;
                          OUT_aux.m(end) + OUT_aux.pxn] - ...
                        [ RODS.OUT.B1.m(end) ;
                          RODS.OUT.B1.p(:,end) ;
                          RODS.OUT.B1.m(end) + RODS.OUT.B1.pxn ] ;
Jacob([1:3 12],1) = Jacob([1:3 12],1)/epsilon ;
RODS.IN.B1.m0 = RODS.IN.B1.m0 - epsilon ;

RODS.IN.B1.n0(1) = RODS.IN.B1.n0(1) + epsilon ;
OUT_aux = RK4_rod( RODS.IN.B1 ) ;
Jacob([1:3 12],2) = [ OUT_aux.m(end) ;
                          OUT_aux.p(:,end) ;
                          OUT_aux.m(end) + OUT_aux.pxn] - ...
                        [ RODS.OUT.B1.m(end) ;
                          RODS.OUT.B1.p(:,end) ;
                          RODS.OUT.B1.m(end) + RODS.OUT.B1.pxn ] ;
Jacob([1:3 12],2) = Jacob([1:3 12],2)/epsilon ;
RODS.IN.B1.n0(1) = RODS.IN.B1.n0(1) - epsilon ;

RODS.IN.B1.n0(2) = RODS.IN.B1.n0(2) + epsilon ;
OUT_aux = RK4_rod( RODS.IN.B1 ) ;
Jacob([1:3 12],3) = [ OUT_aux.m(end) ;
                          OUT_aux.p(:,end) ;
                          OUT_aux.m(end) + OUT_aux.pxn] - ...
                        [ RODS.OUT.B1.m(end) ;
                          RODS.OUT.B1.p(:,end) ;
                          RODS.OUT.B1.m(end) + RODS.OUT.B1.pxn ] ;
Jacob([1:3 12],3) = Jacob([1:3 12],3)/epsilon ;
RODS.IN.B1.n0(2) = RODS.IN.B1.n0(2) - epsilon ;

Jacob([2:3 12],4) = [ cos(RODS.IN.B1.theta0) ;
                          sin(RODS.IN.B1.theta0) ;
                          cos(RODS.IN.B1.theta0)*RODS.IN.B1.n0(2) - ...
                          sin(RODS.IN.B1.theta0)*RODS.IN.B1.n0(1) ] ;


% Variables for rod 2
RODS.IN.B2.m0 = RODS.IN.B2.m0 + epsilon ;
OUT_aux = RK4_rod( RODS.IN.B2 ) ;
Jacob([4:6 12],5) = [ OUT_aux.m(end) ;
                          OUT_aux.p(:,end) ;
                          OUT_aux.m(end) + OUT_aux.pxn] - ...
                        [ RODS.OUT.B2.m(end) ;
                          RODS.OUT.B2.p(:,end) ;
                          RODS.OUT.B2.m(end) + RODS.OUT.B2.pxn ] ;
Jacob([4:6 12],5) = Jacob([4:6 12],5)/epsilon ;
RODS.IN.B2.m0 = RODS.IN.B2.m0 - epsilon ;

RODS.IN.B2.n0(1) = RODS.IN.B2.n0(1) + epsilon ;
OUT_aux = RK4_rod( RODS.IN.B2 ) ;
Jacob([4:6 12],6) = [ OUT_aux.m(end) ;
                          OUT_aux.p(:,end) ;
                          OUT_aux.m(end) + OUT_aux.pxn] - ...
                        [ RODS.OUT.B2.m(end) ;
                          RODS.OUT.B2.p(:,end) ;
                          RODS.OUT.B2.m(end) + RODS.OUT.B2.pxn ] ;
Jacob([4:6 12],6) = Jacob([4:6 12],6)/epsilon ;
RODS.IN.B2.n0(1) = RODS.IN.B2.n0(1) - epsilon ;

RODS.IN.B2.n0(2) = RODS.IN.B2.n0(2) + epsilon ;
```

```
OUT_aux = RK4_rod( RODS.IN.B2 ) ;
Jacob([4:6 12],7) = [ OUT_aux.m(end) ;
                      OUT_aux.p(:,end) ;
                      OUT_aux.m(end) + OUT_aux.pxn] - ...
                    [ RODS.OUT.B2.m(end) ;
                      RODS.OUT.B2.p(:,end) ;
                      RODS.OUT.B2.m(end) + RODS.OUT.B2.pxn ] ;
Jacob([4:6 12],7) = Jacob([4:6 12],7)/epsilon ;
RODS.IN.B2.n0(2) = RODS.IN.B2.n0(2) - epsilon ;

Jacob([5:6 12],8) = [ cos(RODS.IN.B2.theta0) ;
                      sin(RODS.IN.B2.theta0) ;
                      cos(RODS.IN.B2.theta0)*RODS.IN.B2.n0(2) - ...
                      sin(RODS.IN.B2.theta0)*RODS.IN.B2.n0(1) ] ;


% Variables for rod 3
RODS.IN.B3.m0 = RODS.IN.B3.m0 + epsilon ;
OUT_aux = RK4_rod( RODS.IN.B3 ) ;
Jacob([7:9 12],9) = [ OUT_aux.m(end) ;
                      OUT_aux.p(:,end) ;
                      OUT_aux.m(end) + OUT_aux.pxn] - ...
                    [ RODS.OUT.B3.m(end) ;
                      RODS.OUT.B3.p(:,end) ;
                      RODS.OUT.B3.m(end) + RODS.OUT.B3.pxn ] ;
Jacob([7:9 12],9) = Jacob([7:9 12],9)/epsilon ;
RODS.IN.B3.m0 = RODS.IN.B3.m0 - epsilon ;

RODS.IN.B3.n0(1) = RODS.IN.B3.n0(1) + epsilon ;
OUT_aux = RK4_rod( RODS.IN.B3 ) ;
Jacob([7:9 12],10) = [ OUT_aux.m(end) ;
                       OUT_aux.p(:,end) ;
                       OUT_aux.m(end) + OUT_aux.pxn] - ...
                     [ RODS.OUT.B3.m(end) ;
                       RODS.OUT.B3.p(:,end) ;
                       RODS.OUT.B3.m(end) + RODS.OUT.B3.pxn ] ;
Jacob([7:9 12],10) = Jacob([7:9 12],10)/epsilon ;
RODS.IN.B3.n0(1) = RODS.IN.B3.n0(1) - epsilon ;

RODS.IN.B3.n0(2) = RODS.IN.B3.n0(2) + epsilon ;
OUT_aux = RK4_rod( RODS.IN.B3 ) ;
Jacob([7:9 12],11) = [ OUT_aux.m(end) ;
                       OUT_aux.p(:,end) ;
                       OUT_aux.m(end) + OUT_aux.pxn] - ...
                     [ RODS.OUT.B3.m(end) ;
                       RODS.OUT.B3.p(:,end) ;
                       RODS.OUT.B3.m(end) + RODS.OUT.B3.pxn ] ;
Jacob([7:9 12],11) = Jacob([7:9 12],11)/epsilon ;
RODS.IN.B3.n0(2) = RODS.IN.B3.n0(2) - epsilon ;

Jacob([8:9 12],12) = [ cos(RODS.IN.B3.theta0) ;
                       sin(RODS.IN.B3.theta0) ;
                       cos(RODS.IN.B3.theta0)*RODS.IN.B3.n0(2) - ...
                       sin(RODS.IN.B3.theta0)*RODS.IN.B3.n0(1) ] ;

% Unit elements
Jacob(10,[2 6 10]) = 1 ;
Jacob(11,[3 7 11]) = 1 ;
```

```
function [ resid ] = Residue( RODS, Fext, Mext )
% Function of the residue vector of the whole mechanism

resid = [ RODS.OUT.B1.m(end) ;
          RODS.OUT.B1.p(:,end) - RODS.IN.B1.pL ;
          RODS.OUT.B2.m(end) ;
          RODS.OUT.B2.p(:,end) - RODS.IN.B2.pL ;
          RODS.OUT.B3.m(end) ;
          RODS.OUT.B3.p(:,end) - RODS.IN.B3.pL ;
          RODS.IN.B1.n0 + RODS.IN.B2.n0 + RODS.IN.B3.n0 - Fext ;
          RODS.OUT.B1.m(end) + RODS.OUT.B2.m(end) + RODS.OUT.B3.m(end)
+ ...
          RODS.OUT.B1.pxn + RODS.OUT.B2.pxn + RODS.OUT.B3.pxn + ...
          - RODS.IN.OP(1)*Fext(2)+RODS.IN.OP(2)*Fext(1) - Mext] ;
```

```
function [ OUT ] = RK4_rod ( IN )
% Functions that solves de initial value problem using the method of
% Runge-Kutta of order four.
%   IN:  Structure in which are geometric and mechanical parameters
%       IN.L        Length of the rod [m]
%       IN.N        Number of nodes of the rod
%       IN.EI       Bending stiffness of the rod [Pa]
%       IN.d        Distance of the rigid part located in pinned end
of the
%                   rod [m]
%       IN.alpha    Angle of the guide [rad]
%       In.theta0   Angle of the rod at clamped end [rad]
%       IN.lambda   Value of lambda [m]
%       IN.m0       Bending at the clamped end [N.m]
%       IN.n0       Internal force at the clamped end (2x1 vector) [N]

%   OUT:    Structure in which are saved the solution
%       OUT.p       array (2xIN.N) with the x,y component of the
%                   centroid position of the rod
%       OUT.m       array (1xIN.N) with the bending moment in each
node
%                   of the rod
%       OUT.theta   array (1xIN.N) with the angle of the tangent of
%                   the rod
%       OUT.Ener    Elastic deformation enegy of the rod using the
Simpson
%                   rule


% Increment of length
ds = IN.L/(IN.N-1) ;

%%%%%%%%%%%%%%%%%%%%%% Variables to save the solution
%%%%%%%%%%%%%%%%%%%%%
OUT.p     = zeros(2,IN.N+1) ;   % Centroid position vector function
(x,y)
OUT.theta = zeros(1,IN.N+1) ;   % Angle function of the tangent of the
rod
OUT.m     = zeros(1,IN.N+1) ;   % Bending moment function
```

```
% Variables at the first point (clamped end)
OUT.p(:,1)    = IN.OA + IN.lambda*[cos(IN.alpha); sin(IN.alpha)];
OUT.theta(1) = IN.theta0 ;
OUT.m(1)      = IN.m0 ;

% Vector with the values of the dependent variables at the clamped end
var = [ OUT.p(:,1) ;
        OUT.theta(1) ;
        OUT.m(1) ] ;

% "For" loop to integrate the system of differential equations through
the
% rod
for ii = 2:IN.N

    % Runge-Kutta method of order four
    k1 = ds *  Right_function( IN.EI, IN.n0, var ) ;
    k2 = ds *  Right_function( IN.EI, IN.n0, var + 0.5*k1 ) ;
    k3 = ds *  Right_function( IN.EI, IN.n0, var + 0.5*k2 ) ;
    k4 = ds *  Right_function( IN.EI, IN.n0, var + k3 ) ;

    var = var + (k1 + 2*k2 + 2*k3 + k4)/6 ;

    OUT.p(:,ii)    = var(1:2) ;
    OUT.theta(ii) = var(3) ;
    OUT.m(ii)      = var(4) ;

end


% Results variables at the pinned end (rigid part)
OUT.p(:,end)    = OUT.p(:,end-1) + ...
                  IN.d*[cos(OUT.theta(end-1)); sin(OUT.theta(end-1))] ;
OUT.theta(end) = OUT.theta(end-1) ;
OUT.m(end)      = OUT.m(end-1) + (OUT.p(1,end-1)-OUT.p(1,end))*IN.n0(2)
-...
                  (OUT.p(2,end-1)-OUT.p(2,end))*IN.n0(1)   ;


% Vector product of pinned end position and force applied on it
OUT.pxn = OUT.p(1,end)*IN.n0(2)-OUT.p(2,end)*IN.n0(1) ;


% Elastic energy calculated throught the the Simpson's rule
m2 = OUT.m(1:end-1).^2 ;          % Square of the internal moemnt
funciton

OUT.Ener = ds/3 * ( m2(1) + m2(end) + 4*sum(m2(2:2:(IN.N-1))) + ...
           2*sum(m2(3:2:(IN.N-2))) ) ;
OUT.Ener = OUT.Ener/IN.EI ;



function [ func ] = Right_function( EI, n, var )
% Value of the right part of the system of differential equations
% var(1)     px
% var(2)     py
```

```
% var(3)    theta
% var(4)    moment

func = [ cos(var(3)) ;
         sin(var(3)) ;
         var(4)/EI ;
         n(1)*sin(var(3)) - n(2)*cos(var(3)) ] ;
```

4.4. INTEGRALES ELÍPTICAS. PROBLEMA DIRECTO

```
%%%%%%%% ALGORITHM THAT OBTAINS ALL THE SOLUTIONS OF THE FK PROBLEM
%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%% 3 DOF CLOSED-LOOP MECHANISM
%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%% 3 FLEXIBLE CLAMPED-PINNED RODS AND A TRIANGLULAR ACTUATOR
%%%%%%%%

clc; close all; clear;
addpath('EllipticIntegrals_functions') ;
tic ;

% Tolerance
tol = 1 ;
tolM = 0.5 ;

% Mechanism type selection: opt = 1 for PFR
%                           opt = 2 for RFR
opt = 1 ;

%% INPUT DATA

if opt == 1  % PFR
    % Reference point through which each of the guides passes
    OO1 = [cos(-150/180*pi); sin(-150/180*pi)] ;
    OO2 = [cos(-30/180*pi); sin(-30/180*pi)] ;
    OO3 = [0; 1] ;

    % Angle of the guide of each linear actuator
    alpha1 = 0 ;
    alpha2 = 120/180*pi ;
    alpha3 = -120/180*pi ;

    % Angles of the actuators
    theta1 = alpha1 ;
    theta2 = alpha2 ;
    theta3 = alpha3 ;

    % INPUT: Distance from the reference point to the actuators
    lambda1 = 0.1 ;
    lambda2 = 0.1 ;
    lambda3 = 0.1 ;

    % Positions of the actuators
    OA1 = OO1 + lambda1*[cos(alpha1); sin(alpha1)] ;
    OA2 = OO2 + lambda2*[cos(alpha2); sin(alpha2)] ;
    OA3 = OO3 + lambda3*[cos(alpha3); sin(alpha3)] ;

elseif opt == 2  %RFR
    % Positions of the actuators
    OA1 = [cos(-150/180*pi); sin(-150/180*pi)] ;
    OA2 = [cos(-30/180*pi); sin(-30/180*pi)] ;
```

```
    OA3 = [0; 1] ;

    % INPUT: Angles of the actuators
    theta1 = 45/180*pi ;
    theta2 = 180/180*pi ;
    theta3 = -60/180*pi ;
end

% Load at triangular actuator centre (P point)
FP = [0.3; -0.2] ;

% Mode of deflexion to be considered in the solution
mode1 = 1 ;
mode2 = 1 ;
mode3 = 1 ;

%% GEOMETRIC AND MECHANICAL PROPERTIES
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Rod length [m]
L = 1 ;

% Distance of the rigid part at the pinned end
d  = 0.10 ;

% Triangular actuator side length
a = 0.5 ;

% Cross-section properties
diam = 1.5e-3  ;
r = 0.5*diam ;
I = 0.25*pi*r^4 ;

% Young's modulus
E = 210e9 ;

%% CREATION OF 'IN' DATA STRUCTURE
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

IN.B1.L          = L ;
IN.B1.d          = d ;
IN.B1.a          = a ;
IN.B1.EI         = E*I ;
IN.B1.mode       = mode1 ;
IN.B1.x0         = OA1(1) ;
IN.B1.y0         = OA1(2) ;
IN.B1.lambda     = 0 ;
IN.B1.theta      = theta1 ;

IN.B2.L          = L ;
IN.B2.d          = d ;
IN.B2.a          = a ;
IN.B2.EI         = E*I ;
IN.B2.mode       = mode2 ;
IN.B2.x0         = OA2(1) ;
IN.B2.y0         = OA2(2) ;
IN.B2.lambda     = 0 ;
IN.B2.theta      = theta2 ;
```

```
IN.B3.L          = L ;
IN.B3.d          = d ;
IN.B3.a          = a ;
IN.B3.EI         = E*I ;
IN.B3.mode       = mode3 ;
IN.B3.x0         = OA3(1) ;
IN.B3.y0         = OA3(2) ;
IN.B3.lambda     = 0 ;
IN.B3.theta      = theta3 ;

IN.FP            = FP ;


%% DISCRETIZATION OF KR1, PSI1, PSI2, PSI3
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

nt   = 21 ;  % Number of divisions in positive/negative kr
npsi = 20 ;  % Number of divisions in psi

% Discretization of kr with exponential relationship
p = 0.25 ;
[A, B] = p2AB(p) ;

t = linspace(0,1,nt) ;
y = A*(exp(B*t)-1) ;
y(1) = 1e-6 ;
y(end) = 1-1e-6 ;

kr  = [-fliplr(y) y] ;

% Discretization of psi with equidistant
psi      = linspace(0, 2*pi, npsi) ;
psi(1)   = psi(1) + 1e-3 ;
psi(end) = psi(end) - 1e-3 ;


[KR1, PSI1, PSI2] = meshgrid(kr, psi, psi) ;

for ii = 1:length(psi)
    KR1(:,:,:,ii)  = KR1(:,:,:,1) ;
    PSI1(:,:,:,ii) = PSI1(:,:,:,1) ;
    PSI2(:,:,:,ii) = PSI2(:,:,:,1) ;
end

PSI3 = zeros(size(KR1)) ;
for ii = 1:size(KR1,1)
    for jj = 1:size(KR1,2)
        for kk = 1:size(KR1,3)
            PSI3(ii,jj,kk,:) = psi ;
        end
    end
end

clc; disp('Discretization done') ;

%% CALCULATION
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% tic ;
% Calculation of end position and load for each combination of
(psi,kr) in
% rod 1
IN.B1.kr  = KR1 ;
IN.B1.psi = PSI1 ;

[X1_loc, Y1_loc, R1] = EmpArt_xyR_local(IN.B1) ;


% Calcluation of R2, R3 loads through the force balance
aux1 = FP(1) - R1.*cos(theta1 + PSI1) ;
aux2 = FP(2) - R1.*sin(theta1 + PSI1) ;

R2 = (aux1.*sin(theta3 + PSI3) - aux2.*cos(theta3 + PSI3))./sin(theta3
+ PSI3 - theta2 - PSI2) ;
R3 = (aux1 - R2.*cos(theta2 + PSI2))./cos(theta3 + PSI3) ;


% Calculation of kr from each (psi, R) combination in rods 2 and 3
KR2 = zeros(size(PSI2)) ;
KR2(:,:,:,:,4,4) = 0 ;
KR3 = zeros(size(PSI3)) ;
KR3(:,:,:,:,4,4) = 0 ;
clc; disp('Calculating kr2, kr3...') ;
for ii = 1:size(PSI2,1)
    for jj = 1:size(PSI2,2)
        for kk = 1:size(PSI2,3)
            for ll = 1:size(PSI2,4)
                clc; toc; disp([ii jj kk ll]) ;
                % kr2
                IN.B2.psi = PSI2(ii,jj,kk,ll) ;
                R2_aux = R2(ii,jj,kk,ll) ;
                kr2 = kr_calc(IN.B2, R2_aux) ;

                % kr3
                IN.B3.psi = PSI3(ii,jj,kk,ll) ;
                R3_aux = R3(ii,jj,kk,ll) ;
                kr3 = kr_calc(IN.B3, R3_aux) ;

                % Building KR2, KR3 matrixes
                for mm = 1:4
                    KR2(ii,jj,kk,ll,:,mm) = kr2 ;
                    KR3(ii,jj,kk,ll,mm,:) = kr3 ;
                end
            end
        end
    end
end
clc; disp('kr2, kr3 calculation done') ;
t1 = toc ;

% KR2, KR3 are 6D matrixes
% KR2(ii,jj,kk,ll,:,mm) contains all the possible kr values for
% PSI2(ii,jj,kk,ll) and R2(ii,jj,kk,ll)

% kr2, kr3 son matrices de 5D. En matrices de 4D se inserta en cada
% elemento un vector de longitud 4, que será la dimensión 5. Cada
vector
```

```
% contiene las posibles soluciones de kr2/kr3 para cada combinación de
kr1,
% psi1, psi2, psi3

% KR2 = kr2 ;
% for ii = 1:size(kr2,5)
%     KR2(:,:,:,:,:,ii) = kr2 ;
% end
%
% KR3 = kr3;
% KR3(:,:,:,:,:,4) = 0 ;
% for ii = 1:size(kr3,5)
%     KR3(:,:,:,:,ii,:) = kr3 ;
% end

% Expanding all matrixes to 6D to consider all the possible
combinations of
% psi1, psi2, psi3, kr1, kr2, kr3
for ii = 1:4
    KR1(:,:,:,:,ii) = KR1(:,:,:,:,1) ;
    PSI1(:,:,:,:,ii) = PSI1(:,:,:,:,1) ;
    PSI2(:,:,:,:,ii) = PSI2(:,:,:,:,1) ;
    PSI3(:,:,:,:,ii) = PSI3(:,:,:,:,1) ;
end

for ii = 1:4
    KR1(:,:,:,:,:,ii) = KR1(:,:,:,:,:,1) ;
    PSI1(:,:,:,:,:,ii) = PSI1(:,:,:,:,:,1) ;
    PSI2(:,:,:,:,:,ii) = PSI2(:,:,:,:,:,1) ;
    PSI3(:,:,:,:,:,ii) = PSI3(:,:,:,:,:,1) ;
end

% Pinned end position and load in global reference
IN.B1.kr  = KR1 ;
IN.B1.psi = PSI1 ;
[OB1_x, OB1_y, F1_x, F1_y] = EmpArt_xyR(IN.B1) ;

IN.B2.kr  = KR2 ;
IN.B2.psi = PSI2 ;
[OB2_x, OB2_y, F2_x, F2_y] = EmpArt_xyR(IN.B2) ;

IN.B3.kr  = KR3 ;
IN.B3.psi = PSI3 ;
[OB3_x, OB3_y, F3_x, F3_y] = EmpArt_xyR(IN.B3) ;

% Triangular platform centre (P point). This is where external load is
% applied
OP_x = (OB1_x + OB2_x + OB3_x)./3 ;
OP_y = (OB1_y + OB2_y + OB3_y)./3 ;

%% RESIDUES
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Momentum balance
PB1_x = OB1_x - OP_x ;
PB1_y = OB1_y - OP_y ;
PB2_x = OB2_x - OP_x ;
PB2_y = OB2_y - OP_y ;
PB3_x = OB3_x - OP_x ;
PB3_y = OB3_y - OP_y ;
```

```
M1 = F1_x.*PB1_y - F1_y.*PB1_x ;
M2 = F2_x.*PB2_y - F2_y.*PB2_x ;
M3 = F3_x.*PB3_y - F3_y.*PB3_x ;

Resid_M = M1 + M2 + M3 ;

% Geometry
Resid_12 = sqrt((OB2_x - OB1_x).^2 + (OB2_y - OB1_y).^2) - a ;
Resid_23 = sqrt((OB3_x - OB2_x).^2 + (OB3_y - OB2_y).^2) - a ;
Resid_31 = sqrt((OB1_x - OB3_x).^2 + (OB1_y - OB3_y).^2) - a ;

save('FK_data_30psi_7') ;
t2 = toc ;
% load('FK_data') ;
%% FINDING SOLUTIONS
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
pos = find(abs(Resid_M)<0.2*tolM & abs(Resid_12)<tol &
abs(Resid_23)<tol & abs(Resid_31)<tol) ;

psi1_sol = PSI1(pos) ;
psi2_sol = PSI2(pos) ;
psi3_sol = PSI3(pos) ;
kr1_sol  = KR1(pos) ;
kr2_sol  = KR2(pos) ;
kr3_sol  = KR3(pos) ;


%% REFINEMENT AND GRAPHIC REPRESENTATION

solcounter = 0 ;
fprintf('%d possible solutions\n', length(psi1_sol)) ; toc ;

figure;
axis image ;
xlabel('x [m]'); ylabel('y [m]') ;
grid;
hold on ;

for ii = 1:length(psi1_sol)
    IN.B1.psi = psi1_sol(ii) ;
    IN.B1.kr  = kr1_sol(ii) ;
    IN.B2.psi = psi2_sol(ii) ;
    IN.B2.kr  = kr2_sol(ii) ;
    IN.B3.psi = psi3_sol(ii) ;
    IN.B3.kr  = kr3_sol(ii) ;

    fprintf('Newton-Raphson solution %d\n',ii) ;
    IN = FK_NewtonRaphson(IN) ;

    if IN.sol == 1
        psi1_sol(ii) = IN.B1.psi ;
        kr1_sol(ii)  = IN.B1.kr ;
        psi2_sol(ii) = IN.B2.psi ;
        kr2_sol(ii)  = IN.B2.kr ;
        psi3_sol(ii) = IN.B3.psi ;
        kr3_sol(ii)  = IN.B3.kr ;

        GraphicRepresentation(IN, 100) ;
```

```
        solcounter = solcounter + 1 ;
        fprintf('Solution %d\n', solcounter) ;
        fprintf('psi1 = %f [rad]\n', psi1_sol(ii)) ;
        fprintf('kr1 = %f [-]\n', kr1_sol(ii)) ;
        fprintf('psi2 = %f [rad]\n', psi2_sol(ii)) ;
        fprintf('kr2 = %f [-]\n', kr2_sol(ii)) ;
        fprintf('psi3 = %f [rad]\n', psi3_sol(ii)) ;
        fprintf('kr3 = %f [-]\n', kr3_sol(ii)) ;

    elseif IN.sol == 0
        psi1_sol(ii) = NaN ;
        kr1_sol(ii)  = NaN ;
        psi2_sol(ii) = NaN ;
        kr2_sol(ii)  = NaN ;
        psi3_sol(ii) = NaN ;
        kr3_sol(ii)  = NaN ;
    end

end
```

```
function [ A, B ] = p2AB( k )

tol = 1e-6 ;

B = linspace(-100,0,1e3+1) ;

res = exp(B).*(B-k)+k ;

aux = res(1:end-1).*res(2:end) ;

ii = find(aux<0) ;

B1 = B(ii) ;         res1 = res(ii) ;
B2 = B(ii+1) ;       res2 = res(ii+1) ;

while abs(res2) > tol

    B3 = B2 - res2/(res2-res1)*(B2-B1) ;
    res3 = exp(B3).*(B3-k)+k ;

    B1 = B2 ;
    B2 = B3 ;

    res1 = res2 ;
    res2 = res3 ;

end

B = B2 ;
A = 1/(exp(B)-1) ;

% % A = (1-1e-6)*A ;

end


function [ xi, yi ] = Elastic_shape_EmpArt( IN, n )
% Functions that calculates de position of the pinned end of a
clamped-
% pinned rod and the required load.

% IN :  Structure with the input values
%   IN.L :      Lenght of the rod [m]
%   IN.EI :     Inertia of the cross section [m^4]
%   IN.mode :   Mode of deformation
%   IN.psi :    Angle of the force (relative to the clamped angle)
[rad]
%   IN.kr :     Elliptic integral parameter (relative value)
%   IN.theta :  Angle of the clamped end [rad]
%   IN.x0 :     x position of the clamped end [m]
%   IN.y0 :     y position of the clamped end [m]

% Value of k
kmin = abs(cos(0.5*IN.psi)) ;
k    = sign(IN.kr).*kmin + (1-kmin).*IN.kr ;
```

```
% Range of phi
aux  = 1./k.*cos(0.5*IN.psi);
phi1 = asin(aux);
phi2 = (IN.mode-0.5)*pi ;
phi = linspace(phi1, phi2, n) ;


% Deformation of the rod in local coordinate
[F,E] = elliptic12( phi, k^2 );

alpha = F(end)-F(1);

aux1 = (2*E-2*E(1)-F+F(1))/alpha*IN.L;
aux2 = 2*(cos(phi)-cos(phi1))/alpha*IN.L*k;

xlocali = -aux1.*cos(IN.psi)-aux2.*sin(IN.psi);
ylocali = -aux1.*sin(IN.psi)+aux2.*cos(IN.psi);


% Deformation of the road in global coordinate
xi = xlocali*cos(IN.theta) - ylocali*sin(IN.theta) ;
yi = xlocali*sin(IN.theta) + ylocali*cos(IN.theta) ;

xi = xi + IN.x0 + IN.lambda*cos(IN.theta) ;
yi = yi + IN.y0 + IN.lambda*sin(IN.theta) ;

end

function [F,E,Z] = elliptic12(u,m,tol)
% ELLIPTIC12 evaluates the value of the Incomplete Elliptic Integrals
% of the First, Second Kind and Jacobi's Zeta Function.
%
%   [F,E,Z] = ELLIPTIC12(U,M,TOL) where U is a phase in radians, 0<M<1
is
%   the module and TOL is the tolerance (optional). Default value for
%   the tolerance is eps = 2.220e-16.
%
%   ELLIPTIC12 uses the method of the Arithmetic-Geometric Mean
%   and Descending Landen Transformation described in [1] Ch. 17.6,
%   to determine the value of the Incomplete Elliptic Integrals
%   of the First, Second Kind and Jacobi's Zeta Function [1], [2].
%
%       F(phi,m) = int(1/sqrt(1-m*sin(t)^2), t=0..phi);
%       E(phi,m) = int(sqrt(1-m*sin(t)^2), t=0..phi);
%       Z(phi,m) = E(u,m) - E(m)/K(m)*F(phi,m).
%
%   Tables generating code ([1], pp. 613-621):
%       [phi,alpha] = meshgrid(0:5:90, 0:2:90);                  %
modulus and phase in degrees
%       [F,E,Z] = elliptic12(pi/180*phi, sin(pi/180*alpha).^2);  %
values of integrals
%
%   See also ELLIPKE, ELLIPJ, ELLIPTIC12I, ELLIPTIC3, THETA, AGM.
%
%   References:
%   [1] M. Abramowitz and I.A. Stegun, "Handbook of Mathematical
Functions",
```

```
%       Dover Publications", 1965, Ch. 17.1 - 17.6 (by L.M. Milne-
Thomson).
%   [2] D. F. Lawden, "Elliptic Functions and Applications"
%       Springer-Verlag, vol. 80, 1989

% Moiseev Igor
% 34106, SISSA, via Beirut n. 2-4,  Trieste, Italy
% For support, please reply to
%     moiseev[at]sissa.it, moiseev.igor[at]gmail.com
%     Moiseev Igor,
%     34106, SISSA, via Beirut n. 2-4,  Trieste, Italy
%
% The code is optimized for ordered inputs produced by the functions
% meshgrid, ndgrid. To obtain maximum performace (up to 30%) for
singleton,
% 1-dimensional and random arrays remark call of the function
unique(.)
% and edit further code.

if nargin<3, tol = eps; end
if nargin<2, error('Not enough input arguments.'); end

if ~isreal(u) || ~isreal(m)
    error('Input arguments must be real. Use ELLIPTIC12i for complex
arguments.');
end

if length(m)==1, m = m(ones(size(u))); end
if length(u)==1, u = u(ones(size(m))); end
if ~isequal(size(m),size(u)), error('U and M must be the same size.');
end

F = zeros(size(u));
E = F;
Z = E;
m = m(:).';     % make a row vector
u = u(:).';

if any(m < 0) || any(m > 1), error('M must be in the range 0 <= M <=
1.'); end

I = uint32( find(m ~= 1 & m ~= 0) );
if ~isempty(I)
    [mu,J,K] = unique(m(I));   % extracts unique values from m
    K = uint32(K);
    mumax = length(mu);
    signU = sign(u(I));

    % pre-allocate space and augment if needed
    chunk = 7;
    a = zeros(chunk,mumax);
    c = a;
    b = a;
    a(1,:) = ones(1,mumax);
    c(1,:) = sqrt(mu);
    b(1,:) = sqrt(1-mu);
    n = uint32( zeros(1,mumax) );
    i = 1;
    while any(abs(c(i,:)) > tol)                                    %
Arithmetic-Geometric Mean of A, B and C
```

```
        i = i + 1;
        if i > size(a,1)
          a = [a; zeros(2,mumax)];
          b = [b; zeros(2,mumax)];
          c = [c; zeros(2,mumax)];
        end
        a(i,:) = 0.5 * (a(i-1,:) + b(i-1,:));
        b(i,:) = sqrt(a(i-1,:) .* b(i-1,:));
        c(i,:) = 0.5 * (a(i-1,:) - b(i-1,:));
        in = uint32( find((abs(c(i,:)) <= tol) & (abs(c(i-1,:)) >
tol)) );
        if ~isempty(in)
          [mi,ni] = size(in);
          n(in) = ones(mi,ni)*(i-1);
        end
    end

    mmax = length(I);
    mn = double(max(n));
    phin = zeros(1,mmax);     C  = zeros(1,mmax);
    Cp = C;  e  = uint32(C);  phin(:) = signU.*u(I);
    i = 0;   c2 = c.^2;
    while i < mn                                                    %
Descending Landen Transformation
        i = i + 1;
        in = uint32(find(n(K) > i));
        if ~isempty(in)
            phin(in) = atan(b(i,K(in))./a(i,K(in)).*tan(phin(in))) +
...
                pi.*ceil(phin(in)/pi - 0.5) + phin(in);
            e(in) = 2.^(i-1) ;
            C(in) = C(in)  + double(e(in(1)))*c2(i,K(in));
            Cp(in)= Cp(in) + c(i+1,K(in)).*sin(phin(in));
        end
    end

    Ff = phin ./ (a(mn,K).*double(e)*2);
    F(I) = Ff.*signU;                                               %
Incomplete Ell. Int. of the First Kind
    Z(I) = Cp.*signU;                                               %
Jacobi Zeta Function
    E(I) = (Cp + (1 - 1/2*C) .* Ff).*signU;                         %
Incomplete Ell. Int. of the Second Kind
end

% Special cases: m == {0, 1}
m0 = find(m == 0);
if ~isempty(m0), F(m0) = u(m0); E(m0) = u(m0); Z(m0) = 0; end

m1 = find(m == 1);
um1 = abs(u(m1));
if ~isempty(m1),
    N = floor( (um1+pi/2)/pi );
    M = find(um1 < pi/2);

    F(m1(M)) = log(tan(pi/4 + u(m1(M))/2));
    F(m1(um1 >= pi/2)) = Inf.*sign(u(m1(um1 >= pi/2)));

    E(m1) = ((-1).^N .* sin(um1) + 2*N).*sign(u(m1));
```

```
    Z(m1) = (-1).^N .* sin(u(m1));
end

function [ x, y, Fx, Fy ] = EmpArt_xyR( IN )
% Functions that calculates de position of the pinned end of a
clamped-
% pinned rod and the required load.

% IN :  Structure with the input values
%   IN.L :      Lenght of the rod [m]
%   IN.EI :     Inertia of the cross section [m^4]
%   IN.mode :   Mode of deformation
%   IN.psi :    Angle of the force (relative to the clamped angle)
[rad]
%   IN.kr :     Elliptic integral parameter (relative value)
%   IN.theta :  Angle of the clamped end [rad]
%   IN.x0 :     x position of the clamped end [m]
%   IN.y0 :     y position of the clamped end [m]


posNaN = find(not(abs(IN.kr)>=0)) ;
IN.kr(posNaN) = 0.5 ;


kmin = abs(cos(0.5*IN.psi)) ;
k    = sign(IN.kr).*kmin + (1-kmin).*IN.kr ;

aux  = 1./k.*cos(0.5*IN.psi);
phi1 = asin(aux);

[F1, E1] = elliptic12( phi1, k.^2 );
[F2, E2] = elliptic12( (IN.mode-0.5)*pi, k.^2 );

alpha = F2-F1;
R = alpha.^2*IN.EI/IN.L^2;

aux1 = (2*E2-2*E1-F2+F1)./alpha ;
aux2 = -2*cos(phi1)./alpha.*k;

xlocal = IN.L*(-aux1.*cos(IN.psi)-aux2.*sin(IN.psi));
ylocal = IN.L*(-aux1.*sin(IN.psi)+aux2.*cos(IN.psi));

x = xlocal.*cos(IN.theta) - ylocal.*sin(IN.theta) ;
y = xlocal.*sin(IN.theta) + ylocal.*cos(IN.theta) ;

Fx = R.*cos(IN.psi+IN.theta) ;
Fy = R.*sin(IN.psi+IN.theta) ;

x = x + IN.x0 + IN.lambda*cos(IN.theta) ;
y = y + IN.y0 + IN.lambda*sin(IN.theta) ;


if numel(IN.kr) == 1 && numel(posNaN)==1
    x  = NaN ;
    y  = NaN ;
    Fx = NaN ;
    Fy = NaN ;
else
```

```
    x(posNaN)   = NaN ;
    y(posNaN)   = NaN ;
    Fx(posNaN)  = NaN ;
    Fy(posNaN)  = NaN ;
end


end
function [ x, y, R ] = EmpArt_xyR_local( IN )
% Functions that calculates de position of the pinned end of a
clamped-
% pinned rod and the required load.

% IN :  Structure with the input values
%   IN.L :      Lenght of the rod [m]
%   IN.EI :     Inertia of the cross section [m^4]
%   IN.mode :   Mode of deformation
%   IN.psi :    Angle of the force (relative to the clamped angle)
[rad]
%   IN.kr :     Elliptic integral parameter (relative value)


posNaN = find(not(abs(IN.kr)>=0)) ;
IN.kr(posNaN) = 0.5 ;


kmin = abs(cos(0.5*IN.psi)) ;
k    = sign(IN.kr).*kmin + (1-kmin).*IN.kr ;

aux  = 1./k.*cos(0.5*IN.psi);
phi1 = asin(aux);

[F1, E1] = elliptic12( phi1, k.^2 );
[F2, E2] = elliptic12( (IN.mode-0.5)*pi, k.^2 );

alpha = F2-F1;
R = alpha.^2*IN.EI/IN.L^2;

aux1 = (2*E2-2*E1-F2+F1)./alpha ;
aux2 = -2*cos(phi1)./alpha.*k;

x = IN.L*(-aux1.*cos(IN.psi)-aux2.*sin(IN.psi));
y = IN.L*(-aux1.*sin(IN.psi)+aux2.*cos(IN.psi));


if numel(IN.kr) == 1 && numel(posNaN)==1
    x = NaN ;
    y = NaN ;
    R = NaN ;
else
    x(posNaN)  = NaN ;
    y(posNaN)  = NaN ;
    R(posNaN)  = NaN ;
end

end
```

```
function [ Jacob ] = FK_Jacobian( IN )
% Jacobian matrix ...

epsilon = 1e-6 ;


IN = FK_NewtonRaphson( IN ) ;
[ x_ref, y_ref, ~, ~ ] = EmpArt_xyR( IN.B1 ) ;

IN.B1.theta = IN.B1.theta + epsilon ;
IN = FK_NewtonRaphson( IN ) ;
sol = IN.sol ;
[ x_incr1, y_incr1, ~, ~ ] = EmpArt_xyR( IN.B1 ) ;
IN.B1.theta = IN.B1.theta - epsilon ;

IN.B2.theta = IN.B2.theta + epsilon ;
IN = FK_NewtonRaphson( IN ) ;
sol = sol * IN.sol ;
[ x_incr2, y_incr2, ~, ~ ] = EmpArt_xyR( IN.B1 ) ;


Jacob = [ x_incr1 x_incr2 ;
          y_incr1 y_incr2 ] ;
Jacob = Jacob - [ ones(1,2)*x_ref ; ones(1,2)*y_ref ] ;

if sol == 1
    Jacob = Jacob/epsilon ;
else
    Jacob = Jacob*NaN ;
end



end


function [ IN ] = FK_NewtonRaphson( IN )
% This function uses Newton-Raphson method to solve the FK problem

tol = 1e-9 ;

IN.sol = 1 ;

resid  = Residue(IN) ;
resid0 = zeros(size(resid)) ;

% niter = 0 ;
while max(abs(resid))>tol && max(abs(resid-resid0))>tol*0.1
%     niter = niter + 1 ;
%     fprintf('Iteration number %d\n', niter) ;
%     fprintf('kr1 = %1.15f\n\n', IN.B1.kr) ;

    resid0 = resid ;

    J = Jacobian(IN) ;
    % Checking if J matrix is valid
```

```
    % In case of NaN in J, abort calculation
    if not(sum(sum(abs(J)))>=0)
        break ;
    end
    % In case of not full rank of J, abort calculation
    if rank(J)<6
        break ;
    end

    delta = -J\resid0 ;

    [IN, delta] = Update_variables(IN, delta) ;
    if IN.sol == 0
        break ;
    end

    resid = Residue(IN) ;

    while max(abs(resid)) > max(abs(resid0)) && max(abs(delta)) > 1e-
24
%          niter = niter + 1 ;
%          fprintf('Inner iteration number %d\n', niter) ;
%          fprintf('kr1 = %1.15f\n\n', IN.B1.kr) ;

        delta = 0.5*delta ;

        IN.B1.psi = IN.B1.psi - delta(1) ;
        IN.B1.kr  = IN.B1.kr  - delta(2) ;
        IN.B2.psi = IN.B2.psi - delta(3) ;
        IN.B2.kr  = IN.B2.kr  - delta(4) ;
        IN.B3.psi = IN.B3.psi - delta(5) ;
        IN.B3.kr  = IN.B3.kr  - delta(6) ;

        resid = Residue(IN) ;
    end
end

if max(abs(resid)) > tol || not(sum(abs(resid))>=0)
    IN.sol = 0 ;
else
    % P point position is saved for graphic representation
    [ OB1_x, OB1_y, F1_x, F1_y ] = EmpArt_xyR( IN.B1 ) ;
    [ OB2_x, OB2_y, F2_x, F2_y ] = EmpArt_xyR( IN.B2 ) ;
    [ OB3_x, OB3_y, F3_x, F3_y ] = EmpArt_xyR( IN.B3 ) ;

    OP_x = (OB1_x + OB2_x + OB3_x)./3 ;
    OP_y = (OB1_y + OB2_y + OB3_y)./3 ;

    IN.xp = OP_x ;
    IN.yp = OP_y ;
end




function [IN, delta] = Update_variables( IN, delta )
```

```
while IN.B1.psi+delta(1)>2*pi || IN.B1.psi+delta(1)<0
    delta = 0.5*delta ;

    if max(abs(delta)) < 1e-24
        IN.sol = 0 ;
        break ;
    end
end

while IN.B2.psi+delta(3)>2*pi || IN.B2.psi+delta(3)<0
    delta = 0.5*delta ;

    if max(abs(delta)) < 1e-24
        IN.sol = 0 ;
        break ;
    end
end

while IN.B3.psi+delta(5)>2*pi || IN.B3.psi+delta(5)<0
    delta = 0.5*delta ;

    if max(abs(delta)) < 1e-24
        IN.sol = 0 ;
        break ;
    end
end


while sign(IN.B1.kr)~=sign(IN.B1.kr+delta(2)) ||
abs(IN.B1.kr+delta(2))>=1
    delta = 0.5*delta ;

    if max(abs(delta)) < 1e-24
        IN.sol = 0 ;
        break ;
    end
end

while sign(IN.B2.kr)~=sign(IN.B2.kr+delta(4)) ||
abs(IN.B2.kr+delta(4))>=1
    delta = 0.5*delta ;

    if max(abs(delta)) < 1e-24
        IN.sol = 0 ;
        break ;
    end
end

while sign(IN.B3.kr)~=sign(IN.B3.kr+delta(6)) ||
abs(IN.B3.kr+delta(6))>=1
    delta = 0.5*delta ;

    if max(abs(delta)) < 1e-24
        IN.sol = 0 ;
        break ;
    end
end
```

```
IN.B1.psi = IN.B1.psi + delta(1) ;
IN.B1.kr  = IN.B1.kr  + delta(2) ;
IN.B2.psi = IN.B2.psi + delta(3) ;
IN.B2.kr  = IN.B2.kr  + delta(4) ;
IN.B3.psi = IN.B3.psi + delta(5) ;
IN.B3.kr  = IN.B3.kr  + delta(6) ;
```

```
function [ Jacob ] = Jacobian( IN )

epsilon = 1e-12 ;

Jacob = zeros(6) ;

if abs(IN.B1.kr + epsilon)>=1 || abs(IN.B2.kr + epsilon)>=1 ||
abs(IN.B3.kr + epsilon)>=1
    Jacob = NaN*Jacob ;
else
    resid_aux = Residue(IN) ;

    % Rod 1
    IN.B1.psi = IN.B1.psi + epsilon ;
    Jacob(:,1) = Residue(IN) - resid_aux ;
    IN.B1.psi = IN.B1.psi - epsilon ;

    IN.B1.kr = IN.B1.kr + epsilon ;
    Jacob(:,2) = Residue(IN) - resid_aux ;
    IN.B1.kr = IN.B1.kr - epsilon ;

    % Rod 2
    IN.B2.psi = IN.B2.psi + epsilon ;
    Jacob(:,3) = Residue(IN) - resid_aux ;
    IN.B2.psi = IN.B2.psi - epsilon ;

    IN.B2.kr = IN.B2.kr + epsilon ;
    Jacob(:,4) = Residue(IN) - resid_aux ;
    IN.B2.kr = IN.B2.kr - epsilon ;

    % Rod 3
    IN.B3.psi = IN.B3.psi + epsilon ;
    Jacob(:,5) = Residue(IN) - resid_aux ;
    IN.B3.psi = IN.B3.psi - epsilon ;

    IN.B3.kr = IN.B3.kr + epsilon ;
    Jacob(:,6) = Residue(IN) - resid_aux ;
    IN.B3.kr = IN.B3.kr - epsilon ;

    Jacob = Jacob/epsilon ;
end
```

```
function [ resid ] = Residue( IN )

[ OB1_x, OB1_y, F1_x, F1_y ] = EmpArt_xyR( IN.B1 ) ;
[ OB2_x, OB2_y, F2_x, F2_y ] = EmpArt_xyR( IN.B2 ) ;
[ OB3_x, OB3_y, F3_x, F3_y ] = EmpArt_xyR( IN.B3 ) ;

Resid_12 = sqrt((OB2_x - OB1_x).^2 + (OB2_y - OB1_y).^2) - IN.B1.a ;
Resid_23 = sqrt((OB3_x - OB2_x).^2 + (OB3_y - OB2_y).^2) - IN.B1.a ;
Resid_31 = sqrt((OB1_x - OB3_x).^2 + (OB1_y - OB3_y).^2) - IN.B1.a ;

Resid_Fx = - F1_x - F2_x - F3_x + IN.FP(1) ;
Resid_Fy = - F1_y - F2_y - F3_y + IN.FP(2) ;

OP_x = (OB1_x + OB2_x + OB3_x)./3 ;
OP_y = (OB1_y + OB2_y + OB3_y)./3 ;

PB1_x = OB1_x - OP_x ;
PB1_y = OB1_y - OP_y ;
PB2_x = OB2_x - OP_x ;
PB2_y = OB2_y - OP_y ;
PB3_x = OB3_x - OP_x ;
PB3_y = OB3_y - OP_y ;

M1 = F1_x.*PB1_y - F1_y.*PB1_x ;
M2 = F2_x.*PB2_y - F2_y.*PB2_x ;
M3 = F3_x.*PB3_y - F3_y.*PB3_x ;

Resid_M = M1 + M2 + M3 ;

resid = [Resid_12 ;
         Resid_23 ;
         Resid_31 ;
         Resid_Fx ;
         Resid_Fy ;
         Resid_M ] ;



function [  ] = GraphicRepresentation( IN, n )
% Graphic representation of the mechanism

[ xi_B1, yi_B1 ] = Elastic_shape_EmpArt( IN.B1, n ) ;
[ xi_B2, yi_B2 ] = Elastic_shape_EmpArt( IN.B2, n ) ;
[ xi_B3, yi_B3 ] = Elastic_shape_EmpArt( IN.B3, n ) ;

plot(xi_B1, yi_B1, '-b', 'linewidth', 2 ) ;
plot(xi_B2, yi_B2, '-r', 'linewidth', 2 ) ;
plot(xi_B3, yi_B3, '-g', 'linewidth', 2 ) ;

% Triangular platform
plot([xi_B1(end) xi_B2(end) xi_B3(end) xi_B1(end)], [yi_B1(end)
yi_B2(end) yi_B3(end) yi_B1(end)], 'k', 'linewidth', 2) ;


% Straight tangets to the clamped end of each rod
long1 = 0.2*IN.B1.L ;
long2 = 0.2*IN.B2.L ;
```

```
long3 = 0.2*IN.B3.L ;

x1 = [0 long1*cos(IN.B1.theta) ] + IN.B1.x0 ;
y1 = [0 long1*sin(IN.B1.theta) ] + IN.B1.y0 ;

x2 = [0 long2*cos(IN.B2.theta) ] + IN.B2.x0 ;
y2 = [0 long2*sin(IN.B2.theta) ] + IN.B2.y0 ;

x3 = [0 long3*cos(IN.B3.theta) ] + IN.B3.x0 ;
y3 = [0 long3*sin(IN.B3.theta) ] + IN.B3.y0 ;

plot(x1,y1, '--k') ;
plot(x2,y2, '--k') ;
plot(x3,y3, '--k') ;

% External force representation
quiver (IN.xp, IN.yp, IN.FP(1), IN.FP(2), 'm', 'LineWidth', 1.5) ;

end


function [ K ] = StiffnessMatrix( IN )
%UNTITLED7 Summary of this function goes here
%   Detailed explanation goes here

epsilon = 1e-6 ;

K = zeros(2) ;

IN.B1.xp = IN.xp ;
IN.B2.xp = IN.xp ;
IN.B1.yp = IN.yp ;
IN.B2.yp = IN.yp ;

IN.B1.xp = IN.B1.xp + epsilon ;
IN.B2.xp = IN.B2.xp + epsilon ;
IN.B1 = IK_NewtonRaphson_rod( IN.B1 ) ;
IN.B2 = IK_NewtonRaphson_rod( IN.B2 ) ;
if IN.B1.sol*IN.B2.sol == 1
    K(:,1) = [ IN.B1.Fx + IN.B2.Fx ;
               IN.B1.Fy + IN.B2.Fy ]/epsilon ;
else
    K(:,1) = NaN ;
end
IN.B1.xp = IN.B1.xp - epsilon ;
IN.B2.xp = IN.B2.xp - epsilon ;

IN.B1.yp = IN.B1.yp + epsilon ;
IN.B2.yp = IN.B2.yp + epsilon ;
IN.B1 = IK_NewtonRaphson_rod( IN.B1 ) ;
IN.B2 = IK_NewtonRaphson_rod( IN.B2 ) ;
if IN.B1.sol*IN.B2.sol == 1
    K(:,2) = [ IN.B1.Fx + IN.B2.Fx ;
               IN.B1.Fy + IN.B2.Fy ]/epsilon ;
else
    K(:,2) = NaN ;
end
IN.B1.yp = IN.B1.yp - epsilon ;
IN.B2.yp = IN.B2.yp - epsilon ;
```

```
end


function [ K ] = StiffnessMatrix_SM( RODS, Fext )
% Stiffness matrix
K = zeros(2) ;

% Increment of xp and yp position
epsilon = 1e-5 ;



RODS = FK_ShootingMethod( RODS, Fext ) ;

RODS.OUT.xp = 0.5*(RODS.OUT.B1.p(1,end) + RODS.OUT.B2.p(1,end)) ;
RODS.OUT.yp = 0.5*(RODS.OUT.B1.p(2,end) + RODS.OUT.B2.p(2,end)) ;

RODS.IN.B1.theta = RODS.IN.B1.theta0 ;
RODS.IN.B1.px_0 = RODS.IN.B1.OA(1) ;
RODS.IN.B1.py_0 = RODS.IN.B1.OA(2) ;
RODS.IN.B1.n = RODS.IN.B1.n0 ;

RODS.IN.B2.theta = RODS.IN.B2.theta0 ;
RODS.IN.B2.px_0 = RODS.IN.B2.OA(1) ;
RODS.IN.B2.py_0 = RODS.IN.B2.OA(2) ;
RODS.IN.B2.n = RODS.IN.B2.n0 ;

RODS_ref = RODS ;

% Increment in xp
RODS.IN.B1.px_l = RODS_ref.OUT.xp + 0.5*epsilon ;
RODS.IN.B2.px_l = RODS_ref.OUT.xp + 0.5*epsilon ;
RODS.IN.B1.py_l = RODS_ref.OUT.yp ;
RODS.IN.B2.py_l = RODS_ref.OUT.yp ;


[ IN1, OUT1 ] = IK_SM_rod( RODS.IN.B1 ) ;
[ IN2, OUT2 ] = IK_SM_rod( RODS.IN.B2 ) ;

if OUT1.sol==1 && OUT2.sol==1
    K(:,1) = (IN1.n + IN2.n) ;

    RODS.IN.B1.px_l = RODS_ref.OUT.xp - 0.5*epsilon ;
    RODS.IN.B2.px_l = RODS_ref.OUT.xp - 0.5*epsilon ;

    [ IN1, OUT1 ] = IK_SM_rod( RODS.IN.B1 ) ;
    [ IN2, OUT2 ] = IK_SM_rod( RODS.IN.B2 ) ;

    if OUT1.sol==1 && OUT2.sol==1
        K(:,1) = K(:,1) - (IN1.n + IN2.n) ;
    else
        K(:,1) = NaN ;
    end

else
    K(:,1) = NaN ;
end
```

```
RODS = RODS_ref ;

% Increment in yp
RODS.IN.B1.py_l = RODS_ref.OUT.yp + 0.5*epsilon ;
RODS.IN.B2.py_l = RODS_ref.OUT.yp + 0.5*epsilon ;
RODS.IN.B1.px_l = RODS_ref.OUT.xp ;
RODS.IN.B2.px_l = RODS_ref.OUT.xp ;

[ IN1, OUT1 ] = IK_SM_rod( RODS.IN.B1 ) ;
[ IN2, OUT2 ] = IK_SM_rod( RODS.IN.B2 ) ;

if OUT1.sol==1 && OUT2.sol==1
    K(:,2) = (IN1.n + IN2.n) ;

    RODS.IN.B1.py_l = RODS_ref.OUT.yp - 0.5*epsilon ;
    RODS.IN.B2.py_l = RODS_ref.OUT.yp - 0.5*epsilon ;

    [ IN1, OUT1 ] = IK_SM_rod( RODS.IN.B1 ) ;
    [ IN2, OUT2 ] = IK_SM_rod( RODS.IN.B2 ) ;

    if OUT1.sol==1 && OUT2.sol==1
        K(:,2) = K(:,2) - (IN1.n + IN2.n) ;
    else
        K(:,2) = NaN ;
    end

else
    K(:,1) = NaN ;
end


K = K/epsilon ;


end
```