Grado en Robótica
EPSE-USC Lugo

Códigos Python útiles para Física II

Ángel Piñeiro, Septiembre de 2021

Índice

1. Dibuja figuras geométricas

Cuando queremos mover un robot a lo largo de una trayectoria, es necesario calcular las coordenadas de los puntos del espacio por los que queremos pasar. A veces nos interesan trayectorias rectilíneas pero otras veces es más interesante describir otro tipo de trayectorias para sortear objetos o para evitar configuraciones que pueden forzar los motores del robot, por ejemplo. Veamos cómo programar los puntos de diferentes trayectorias de manera paramétrica en Python.

1.1. Segmento rectilíneo

La siguiente función genera los puntos de un segmento rectilíneo utilizando como argumentos el punto central del segmento, el vector que define su dirección, la longitud de la línea y la resolución deseada (número de puntos utilizados para interpolar la trayectoria).

```
def create_line(center, dir_vector, length, steps):
    v = np.array(dir_vector)
    v = v / np.linalg.norm(v)
    start = np.array(center) - length * 0.5 * v     # punto inicial del segmento

    # El número de steps define la resolución del segmento
    points = np.zeros((steps+1, 3))

    # Generamos los puntos del segmento
    for i in range(steps+1):
        x = length / steps * i
        points[i,:] = x * v + start

    # Devolvemos el array que contiene los puntos del segmento
    return points
```

1.2. Paralelogramo

La siguiente función genera los puntos de un paralelogramo utilizando como argumentos el punto central, los vectores que siguen la dirección de dos lados contiguos, las longitudes de los lados y la resolución deseada (número de puntos utilizados para interpolar la trayectoria por cada línea).

```
def create_paralelogram(center, v1, v2, l1, l2, steps):
    v1 = np.array(v1)
    v1 = v1 / np.linalg.norm(v1)
    v2 = np.array(v2)
    v2 = v2 / np.linalg.norm(v2)

    # vértices del paralelogramo
    s1 = np.array(center) - l1 * 0.5 * v1 - l2 * 0.5 * v2
    s2 = np.array(center) + l1 * 0.5 * v1 - l2 * 0.5 * v2
    s3 = np.array(center) + l1 * 0.5 * v1 + l2 * 0.5 * v2
    s4 = np.array(center) - l1 * 0.5 * v1 + l2 * 0.5 * v2

    points = np.zeros((4*(steps+1)+1, 3))

    # Generamos los puntos de cada segmento
    for i in range(steps+1):
        x1 = l1 / steps * i; x2 = l2 / steps * i;
        points[i,:] = x1 * v1 + s1
        points[steps+1+i,:] = x2 * v2 + s2
        points[2*steps+2+i,:] = -x1 * v1 + s3
        points[3*steps+3+i,:] = -x2 * v2 + s4
    points[-1,:] = s1 #repetimos el primer punto porque es una figura cerrada

    # Devolvemos el array que contiene los puntos del segmento
    return points
```

1.3. Polígono Regular

La siguiente función genera los puntos de un polígono regular contenido en una circunferencia, utilizando como argumentos el punto central, un vector perpendicular, el radio de la circunferencia y la resolución deseada (número total de vértices del polígono). Si usamos suficientes puntos se parece a una circunferencia.

```
def create_polygon(center, normal_vector, radius, steps):
    n = np.array(normal_vector)
    n = n / np.linalg.norm(n)
    c = np.array(center)

    # Generamos un vector aleatorio no nulo y que no esté alineado con n
    while True:
        temp = np.random.rand(3)
        if np.linalg.norm(temp) < 1e-5: continue
        temp = temp / np.linalg.norm(temp)
        if np.dot(temp, n) < 0.8: break

    # El producto vectorial de n y temp es un vector que va en la
    # dirección radial de la circunferencia que contiene al polígono
    u = np.cross(n, temp)

    # El producto vectorial del vector normal y del vector
    # radial es un vector tangente a la circunferencia
    v = np.cross(n, u)

    points = np.zeros((steps+1, 3))

    # Generamos los puntos del polígono
    for i in range(steps):
        angle = 2.0 * np.pi / steps * i
        points[i,:] = radius * (np.cos(angle) * u + np.sin(angle) * v) + c
    points[-1] = radius * u + c

    # Devolvemos el array que contiene los puntos del polígono
    return points
```

1.4. Graficar el resultado

Como resultado de los anteriores códigos, obtenemos un array de puntos en 2 dimensiones que se puede representar en una gráfica:

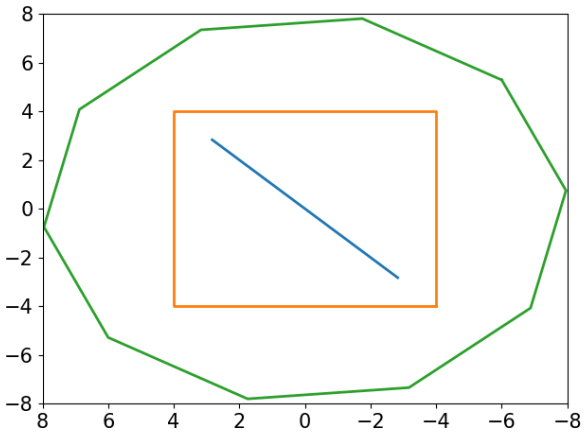
```
import numpy as np
import matplotlib.pyplot as plt

center=[0,0,0]
dir_vector=[1,1,0]
length=8
steps=10
line = create_line(center, dir_vector, length, steps)

v1=[0,1,0]
v2=[1,0,0]
l1=8
l2=8
steps=10
paral = create_paralelogram(center, v1, v2, l1, l2, steps)

normal_vector=[0,0,1]
radius=8
steps=10
polygon = create_polygon(center, normal_vector, radius, steps)
```

```
# Generamos gráfica con círculo y elipses
plt.plot(line[:,0],line[:,1], linewidth=2.0)
plt.plot(paral[:,0],paral[:,1], linewidth=2.0)
plt.plot(polygon[:,0],polygon[:,1], linewidth=2.0)
limitplot=8
plt.ylim(top = limitplot, bottom = -limitplot)
plt.xlim(left = limitplot, right = -limitplot)
plt.xticks(fontsize= 15)
plt.yticks(fontsize= 15)
plt.tight_layout()
plt.show()
```



Posibles ejercicios: Prueba a modificar el código anterior para dibujar puntos en lugar de líneas y propón funciones para figuras geométricas más elaboradas.

2. Ejecución de código Python desde la terminal de Linux

Muchas veces cuando trabajamos con Robots es necesario comunicarse con ellos desde una terminal bash, y ejecutar desde ella códigos en Python u otro lenguaje. Veamos unos ejemplos simples de cómo trabajar de esta manera.

2.1. Comandos básicos de bash

Hay muchas páginas web en las que se resumen los principales comandos de bash, con ejemplos de ejecución. Ver por ejemplo http://www.cse.lehigh.edu/~brian/course/2013/cunix/handouts/bash.quickref.pdf ó https://devhints.io/bash.
Un comando especialmente interesante es el bucle, ya que nos permite enviar valores a códigos que están parametrizados. Veamos un ejemplo de bucle en bash:

```
for i in range {1..10..2}
do
    echo $i
done
```

Este código también se puede escribir de manera compacta en una sola línea:

```
for i in range {1..10..2}; do echo $i; done
```

Ejecuta el código para entender cómo funciona y haz diferentes pruebas.

2.2. Códigos de Python parametrizados

Podemos reescribir el código que genera las gráficas de la sección 1.4 de manera que admita parámetros para las figuras geométricas que dibuja y así poder llamarla desde una terminal bash pasándole valores. Primero introduce las anteriores funciones en un archivo al que llamaremos "mylib.pyz que importaremos como un módulo en nuestro código:

```
#!/usr/bin/env python
#
# By Angel.Pineiro at usc.es
# Version September, 2021
#
# EJEMPLO DE USO:
# python mkplots.py -c [0,0,0] -d 8 -s 10
#
#

import numpy as np
import matplotlib.pyplot as plt
import optparse
from mylib import *

desc="Código Python parametrizado."
def main():
    parser = optparse.OptionParser(description=desc, version='%prog version 1.0')
    parser.add_option('-c', '--center', help='Centro de la figura', action='store')
    parser.add_option('-d', '--dimension', help='Longitud de los segmentos o radio del polígono', actio
    parser.add_option('-s', '--steps', help='Número de puntos', action='store')
    parser.set_defaults(center='0 0 0', dimension=8, steps=10)
    options, arguments = parser.parse_args()
#*****************************************************************************

    c_i=str(options.center).split()
    center=[]
    for i in range (0,3,1): center.append(np.float(c_i[i]))

    print(center)
```

```
    dir_vector=[1,1,0]
    length=float(options.dimension)
    l1=length
    l2=length
    radius=length
    steps=int(options.steps)
    v1=[0,1,0]
    v2=[1,0,0]
    normal_vector=[0,0,1]

    line = create_line(center, dir_vector, length, steps)
    paral = create_paralelogram(center, v1, v2, l1, l2, steps)
    polygon = create_polygon(center, normal_vector, radius, steps)

    # Generamos gráfica con círculo y elipses
    plt.plot(line[:,0],line[:,1], linewidth=2.0)
    plt.plot(paral[:,0],paral[:,1], linewidth=2.0)
    plt.plot(polygon[:,0],polygon[:,1], linewidth=2.0)
    limitplot=8
    plt.ylim(top = limitplot, bottom = -limitplot)
    plt.xlim(left = limitplot, right = -limitplot)
    plt.xticks(fontsize= 15)
    plt.yticks(fontsize= 15)
    plt.tight_layout()
    plt.show()

if __name__=="__main__" :
    main()
```

2.3. Paso de parámetros a códigos de Python desde la terminal

Para ejecutar el código anterior desde la terminal basta con teclear:

```
python mkplots.py -c '0 0 0' -d 8 -s 10
```

Ahora supongamos que queremos reejecutar el código para que haga las figuras de distinto tamaño:

```
python mkplots.py -c '0 0 0' -d 4 -s 10
```

O también podemos ejecutar el código en un bucle, pasándole el índice del bucle como parámetro:

```
for i in range {1..10..2}; do python mkplots.py -c '0 0 0' -d $i -s 10; done
```

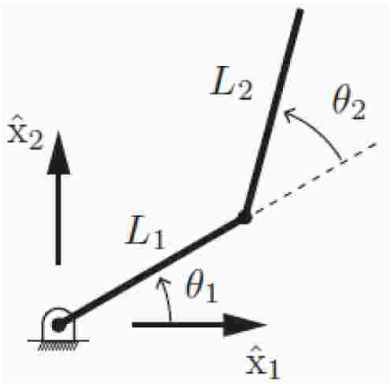
Igual que cambiamos las dimensiones de las figuras, también podemos cambiar su posición, o cualquier otro parámetro, sin modificar el código.

3. Singularidades en un Robot 2R

3.1. Teoría

Supongamos que tenemos un brazo articulado de 2 grados de libertad, con 2 eslabones y 2 articulaciones de revolución: una en la conexión de la primera articulación con la base y otra en la unión entre las 2 articulaciones. A este dispositivo le vamos a llamar Robot 2R y a su extremo, donde iría acoplada una herramienta, le vamos a llamar elemento terminal.

En un Robot 2R como el de la Figura, las coordenadas (x,y) del elemento terminal en coordenadas cartesianas vienen dadas por las siguientes ecuaciones:



$$x_1 = L_1\cos(\theta_1) + L_2\cos(\theta_1+\theta_2)$$
$$x_2 = L_1\,\mathrm{sen}(\theta_1) + L_2\,\mathrm{sen}(\theta_1+\theta_2)$$

Las derivadas de estas ecuaciones nos dan las velocidades del elemento terminal:

$$\dot{x}_1 = -L_1\dot{\theta}_1\,\mathrm{sen}(\theta_1) - L_2(\dot{\theta}_1+\dot{\theta}_2)\,\mathrm{sen}(\theta_1+\theta_2)$$
$$\dot{x}_2 = L_1\dot{\theta}_1\cos(\theta_1) + L_2(\dot{\theta}_1+\dot{\theta}_2)\cos(\theta_1+\theta_2)$$

que, escritas en forma matricial:

$$\begin{bmatrix}\dot{x}_1\\ \dot{x}_2\end{bmatrix} = \begin{bmatrix}-L_1\,\mathrm{sen}(\theta_1) - L_2\,\mathrm{sen}(\theta_1+\theta_2) & -L_2\,\mathrm{sen}(\theta_1+\theta_2)\\ L_1\cos(\theta_1) + L_2\cos(\theta_1+\theta_2) & L_2\cos(\theta_1+\theta_2)\end{bmatrix}\begin{bmatrix}\dot{\theta}_1\\ \dot{\theta}_2\end{bmatrix} \Rightarrow \boxed{\dot{x} = J(\theta)\dot{\theta}}$$

Donde $J(\theta)$ se llama matriz Jacobiana del robot. La anterior ecuación se puede escribir en forma vectorial:

$$\boxed{v = J_1(\theta)\dot{\theta}_1 + J_2(\theta)\dot{\theta}_2}$$

Si $J_1(\theta)$ y $J_2(\theta)$ son paralelos, el determinante es nulo y estamos ante lo que se denomina una configuración singular. En una configuración singular el movimiento del robot está restringido a determinadas direcciones independientemente de cómo se muevan las articulaciones.

3.2. Resolución del problema utilizando cálculo simbólico

Encontrar las configuraciones singulares de un robot implica resolver el determinante de la matriz Jacobiana parametrizado con las coordenadas de las articulaciones (los ángulos θ_1 y θ_2 en este caso:

```
import sympy as sp

#Definicíón de variables simbólicas
t1=sp.var("t1") # theta1
t2=sp.var("t2") # theta2

L1=sp.var("L1") # longitud del primer eslabón
L2=sp.var("L2") # longitud del segundo eslabón

# Elementos de la matriz Jacobiana
a=-L1*sp.sin(t1)-L2*sp.sin(t1+t2)
b=L1*sp.cos(t1)+L2*sp.cos(t1+t2)
c=-L2*sp.sin(t1+t2)
d=L2*sp.cos(t1+t2)

m=sp.Matrix([[a,c],[b,d]]) # Matriz Jacobiana
sp.solve(m.det()) # Resolución del determinante
```

Ejecutando el código anterior, obtenemos el siguiente resultado general:

```
[{L1: 0}, {L2: 0}, {t2: -t1 - 2*atan(1/tan(t1/2))}, {t2: -t1 + 2*atan(tan(t1/2))}]
```

Si queremos el resultado para un robot en el que los eslabones son idénticos (longitud=1 en unidades relativas) y el primer eslabón está horizontal, escribimos:

```
sp.solve(m.subs({t1:0, L1:1, L2:1}).det())
```

Y obtenemos como posibles resultados que el robot tiene 2 configuraciones singulares cuando el segundo eslabón forma un ángulo de 0 grados o 180 grados con el primero.

En clase de Física II discutiremos estos resultados!!