2R-simulation

November 12, 2023

1 Simulation of a 2R-arm in Python

The aim of the present note is to simulate a planar serial manipulator with two revolute joints in Python; the tool is a pencil (and, hence, will be represented by a point).

We choose JupyterLab as the development environment, so we can go step by step.

We have to load some necessary libraries; the first is a math-related library called `numpy`.

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

We can now use the commands implemented in numpy, but we have to add a leading "np." For example, the value of π equals:

[2]:
```
np.pi
```

[2]:
```
3.141592653589793
```

The sine of π can be accessed by

[3]:
```
np.sin(np.pi)
```

[3]:
```
1.2246467991473532e-16
```

As you can see, there may be numerical errors.

Notice that powers 3^2 are written as `3**2`.

[4]:
```
3**2
```

[4]:
```
9
```

You may use Python together with numpy as a quite huge calculator: if you want to calculate the numerical value for

$$3^7 \cdot \sin\frac{\pi}{4} \cdot e^2,$$

then the following expression will do it for you.

[5]:
```
3**7 * np.sin(np.pi/4) * np.exp(2)
```

[5]:
```
11426.750611304113
```

You can also extend the abilities of this calculator by adding new functions. Let us see how it works by writing a function that converts degrees to radians. Clearly, if α is an angle given in degree, then the corresponding value in radian is $\alpha \cdot \pi/180°$.

[6]:
```
def deg2rad(alpha):           # the colon ":" is essential here!
    return alpha/180*np.pi    # the indent shows Python that this line belongs to␣
 ↪the definition
```

Let us test the new function

[7]:
```
deg2rad(180)
```

[7]:
```
3.141592653589793
```

To compute the cosine of 120°, we have just to type the following

[8]:
```
np.cos(deg2rad(180))
```

[8]:
```
-1.0
```

Conversely, we also want to convert from radians to degrees:

[9]:
```
def rad2deg(t):
    return t/np.pi/180
```

Another important concept of Python are variables. In our example, we want to store the values for the lengths l_1, l_2 of the upper arm, and the lower arm, respectively.

[10]:
```
l_1, l_2 = 6, 7
```

Whenever you will need the length l_2, then you can call it by its name `l2`:

[11]:
```
l_2
```

[11]:
```
7
```

You can also store the result of a computation to a variable:

[12]:
```
l_all = l_1+l_2+1
l_all
```

[12]:
```
14
```

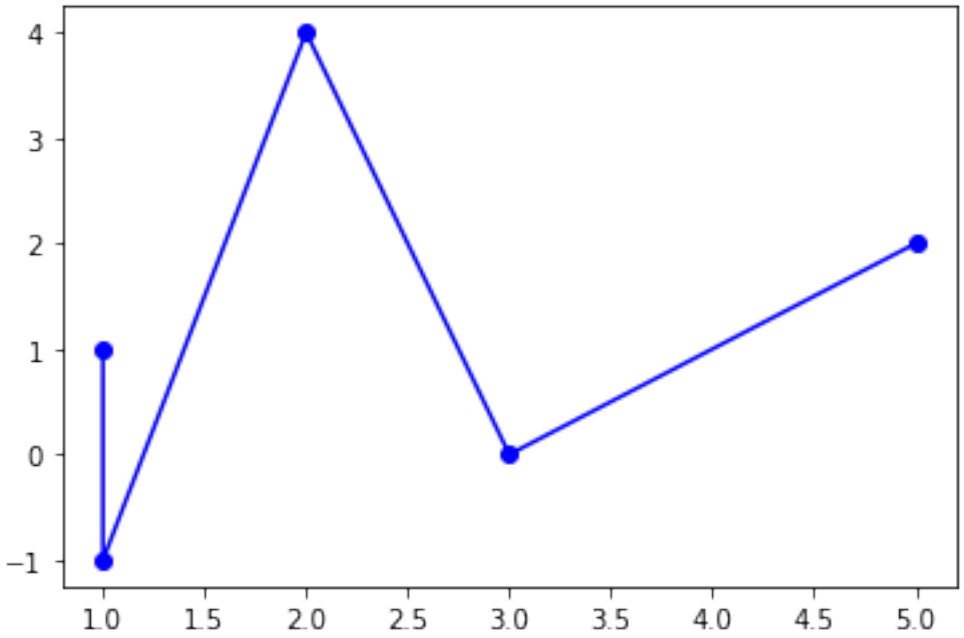
1.1 Plotting with matplotlib

Before we will plot pictures of the 3R-arm, we have to understand (at least a little bit) the way, Python draws pictures. First, we import the library that will do the job:

[13]:
```
import matplotlib.pyplot as plt
```

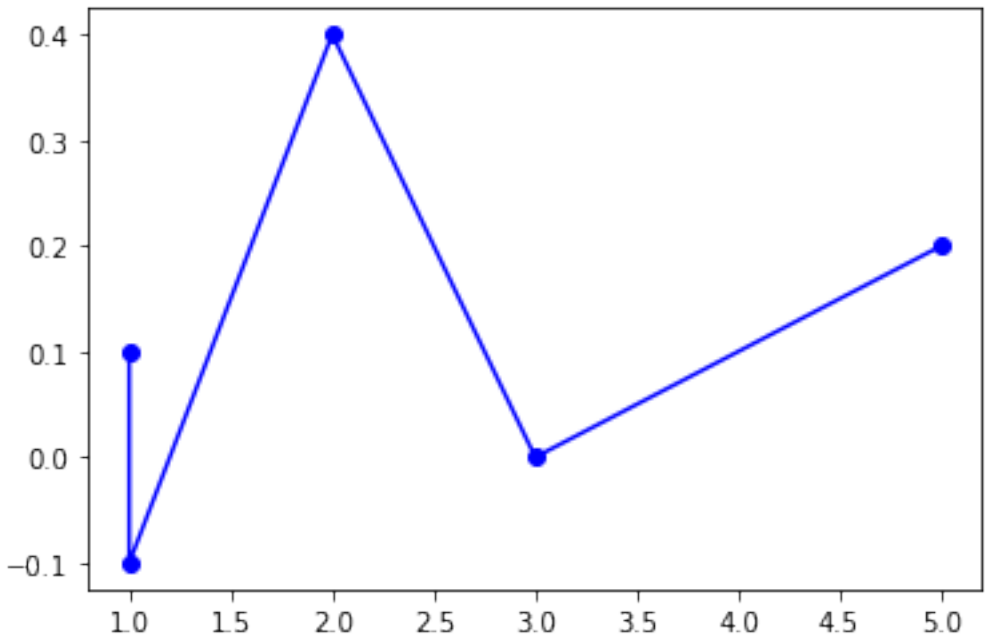
A very quick and basic method to plot figures is the following

[14]:
```
plt.plot([1,1,2,3,5],[1,-1,4,0,2],'-ob')
plt.show()
```



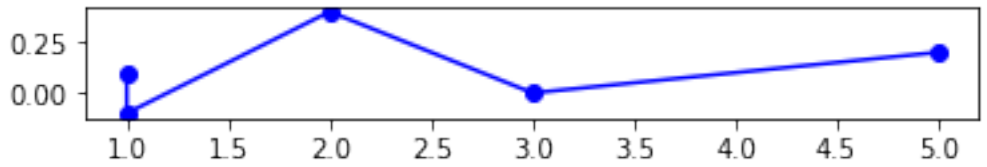
The command `plt.plot`will arrange the plot, while `plt.show()` will display the result. The **lists** `[1,1,2,3,5]` and `[1,-1,4,0,2]`, respectively, are the x- and the y-coordinates, respectively, of some points. By the option `'-ob'`, the points will be displayed as filled circles and the color will be set to blue. Moreover, the points are connected by lines - perfect for drawing a 2R-arm!

[15]:
```
plt.plot([1,1,2,3,5],[1/10,-1/10,4/10,0/10,2/10],'-ob')
plt.show()
```



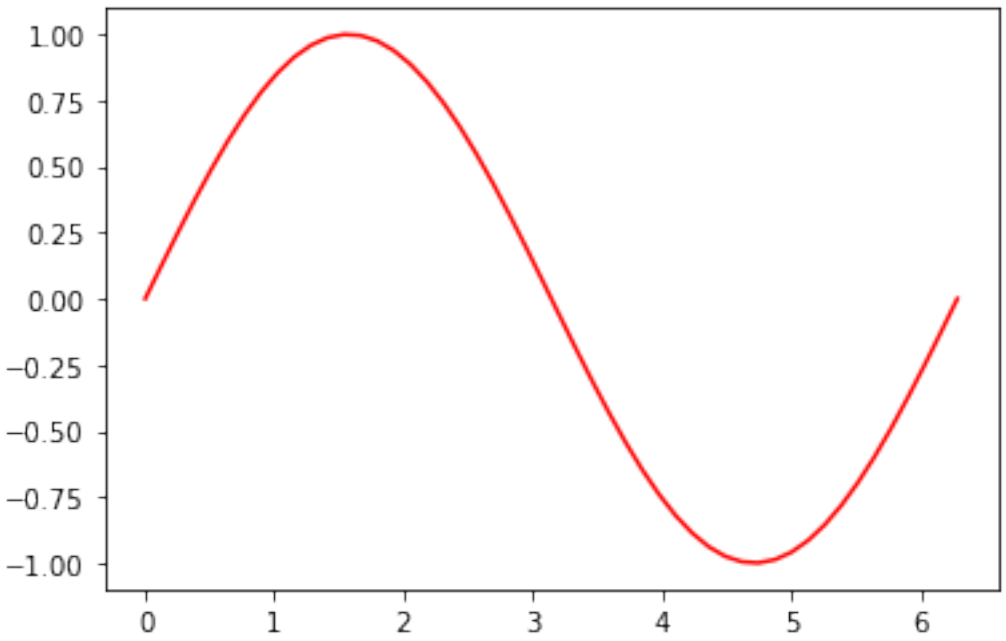
The figure above shows that Python will choose the scaling of the axis by its own, which will lead to distortions in the intended pictures of the 3R-arm. We add a simple command to avoid that behaviour:

[16]:
```
plt.axes().set_aspect('equal')
plt.plot([1,1,2,3,5],[1/10,-1/10,4/10,0/10,2/10],'-ob')
plt.show()
```



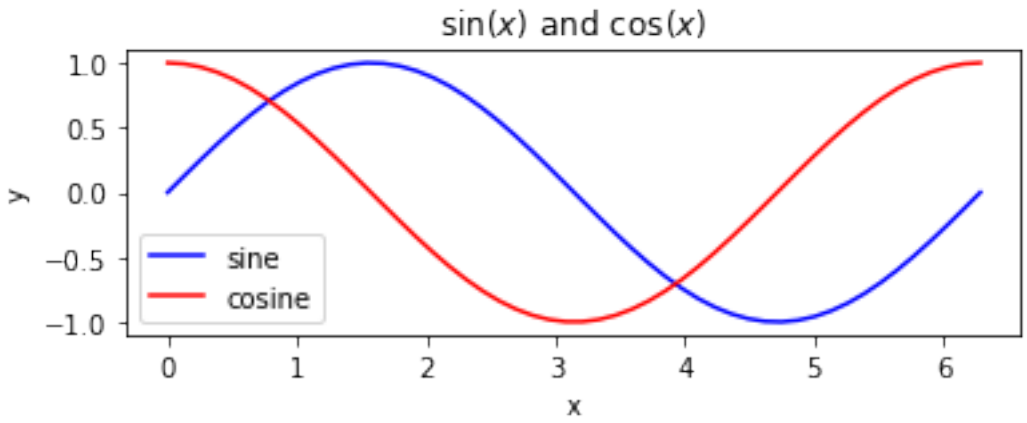
Usually, plots are used to display graphs of functions by choosing a large amount of x-values and their corresponding y-values. We demonstrate it by plotting the sine-function on the interval $[0, 2\pi]$.

[17]:
```
xxx = np.linspace(0,2*np.pi,50)     # creates a list of 50 values between 0 and␣
 ↪2*pi
yyy = np.sin(xxx)
plt.plot(xxx,yyy,'-r')
plt.show()
```



The folowing example is a bit more complicated; it shows the sine and the cosine on the intervall $[0, 2\pi]$.

[18]:
```
xxx = np.linspace(0,2*np.pi,50)                    # creates a list of 50 values␣
 ↪between 0 and 2*pi
fig, ax = plt.subplots()                           # set up a figure and "axes"; the␣
 ↪size of the figure is chosen
ax.set_aspect('equal')
ax.plot(xxx,np.sin(xxx),'-b',label='sine')         # create the sine curve
ax.plot(xxx,np.cos(xxx),'-r',label='cosine')       # create the cosine curve
ax.set_xlabel('x')                                 # label the x-axis by x
ax.set_ylabel('y')                                 # label the y-axis by y
ax.set_title('$\sin(x)$ and $\cos(x)$')            # make a title; matplotlib␣
 ↪understands LaTeX-code
ax.legend();                                       # show the legend; the semicolon␣
 ↪prevents the output of the command
```

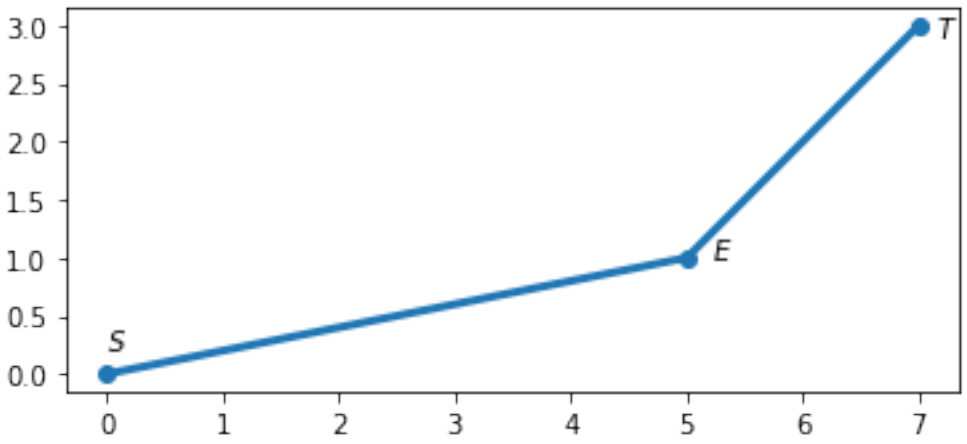


1.2 Plotting the 2R-arm

We are ready to make a picture of the 3R-arm. Supose that the points S, E, T have the coordinates

$$S : (0,0) \tag{1}$$
$$E : (5,1) \tag{2}$$
$$T : (7,3) \tag{3}$$

[19]:
```
xxx = [0,5,7]    # collect the x-coordinates
yyy = [0,1,3]    # collect the y-coordinates
fig, ax = plt.subplots()
ax.set_aspect('equal')
ax.text(0,0.2,'$S$')
ax.text(5.2,1,'$E$')
ax.text(7.15,2.9,'$T$')
ax.plot(xxx,yyy,'-o', lw=3);    # "lw" stands for "linewidth" and controls the␣
 ↪width of the lines
```



Next, we want to have a function that returns the solution of the DKP (i.e. the coordinates of S, E, W, G) for a specific choice of the joint angles. Recall that

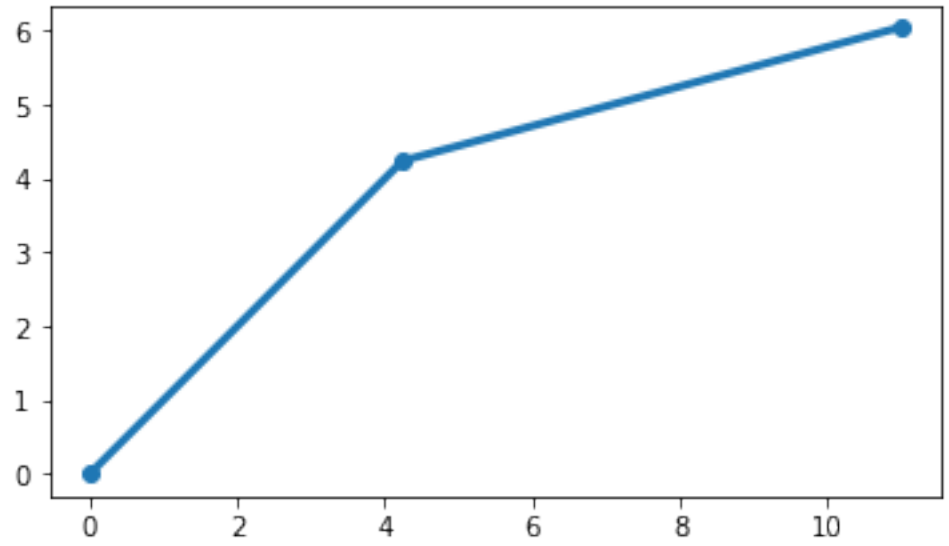
$$S = (0, 0) \tag{4}$$
$$E = (l_1 \cos(t_1), l_1 \sin(t_1)) \tag{5}$$
$$W = (l_1 \cos(t_1) + l_2 \cos(t_1 + t_2), l_1 \sin(t_1) + l_2 \sin(t_1 + t_2)) \tag{6}$$

Furthermore, notice that we need a list of the x-coordinates and a list of the y-coordinates for the plots.

[20]:
```
def DKPplot(td_1,td_2):     # input: joint angles in degree
    a_1 = deg2rad(td_1)           # conversion to radian - we already wrote a␣
 ↪function!
    a_2 = deg2rad(td_1+td_2)     # we will need the sum of the first two angles
    xxx = ([0,l_1*np.cos(a_1),
            l_1*np.cos(a_1)+l_2*np.cos(a_2)])
    yyy = ([0,l_1*np.sin(a_1),
            l_1*np.sin(a_1)+l_2*np.sin(a_2)])
    return [xxx,yyy]
```

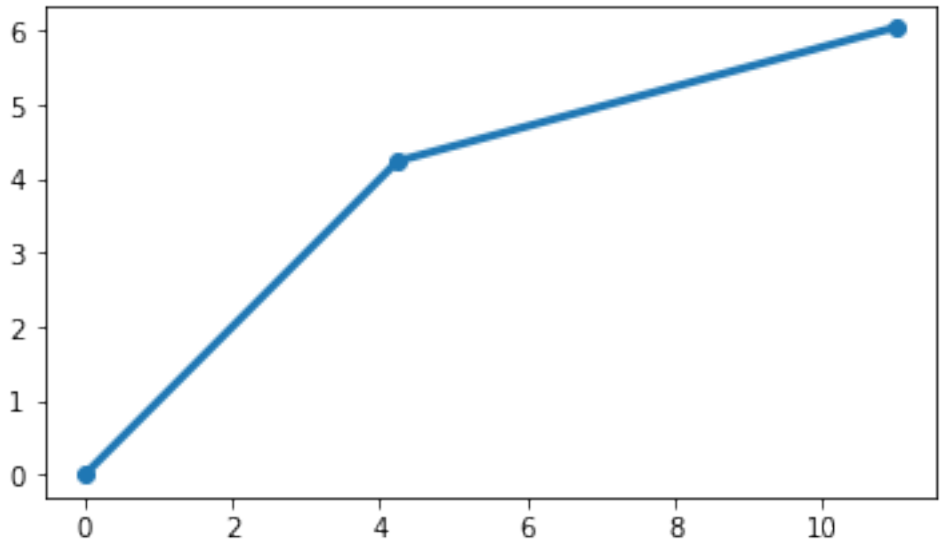
Make sure that you understand the preceding function! We test it:

[21]:
```
result = DKPplot(45,-30)     # compute the position of the points of the 3R-arm
fig, ax = plt.subplots()
ax.set_aspect('equal')
ax.plot(result[0],result[1],'-o', lw=3);     # "result[0]" is the first list in␣
 ↪"result"
                                             # "result[1]" is the second list in␣
 ↪"result"
```



Instead of accessing the two elements of the list `result = DKPplot(...)` by `result[0]` and `result[1]`, we may also use

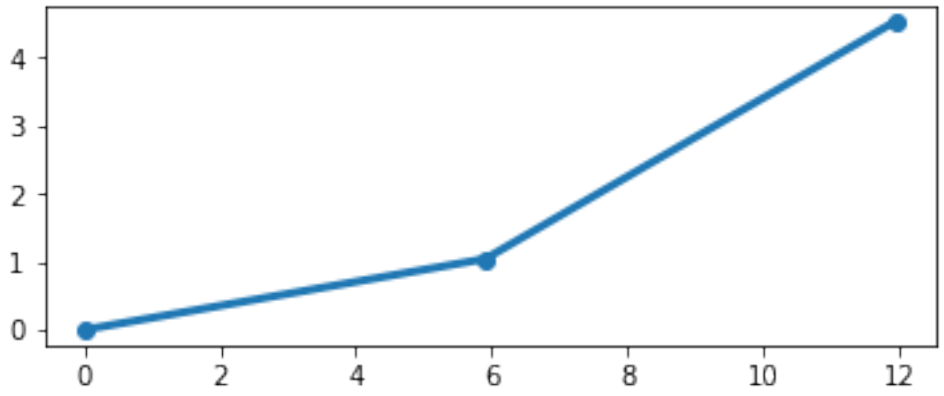
[22]:
```
resultx,resulty = DKPplot(45,-30)     # the two components of DKPplot are␣
 ↪unpacked into the variables `resultx`and `resulty`
fig, ax = plt.subplots()
ax.set_aspect('equal')
ax.plot(resultx,resulty,'-o', lw=3);     # "result[0]" is the first list in␣
 ↪"result"
                                         # "result[1]" is the second list in␣
 ↪"result"
```



For playing around it may be uncomfortable to type the preceeding lines again and again - let us make a function that will do the job for us!

[23]:
```
def DKPpic(t1,t2):
    result = DKPplot(t1,t2)
    fig, ax = plt.subplots()
    ax.set_aspect('equal')
    ax.plot(result[0],result[1],'-o', lw=3);
```

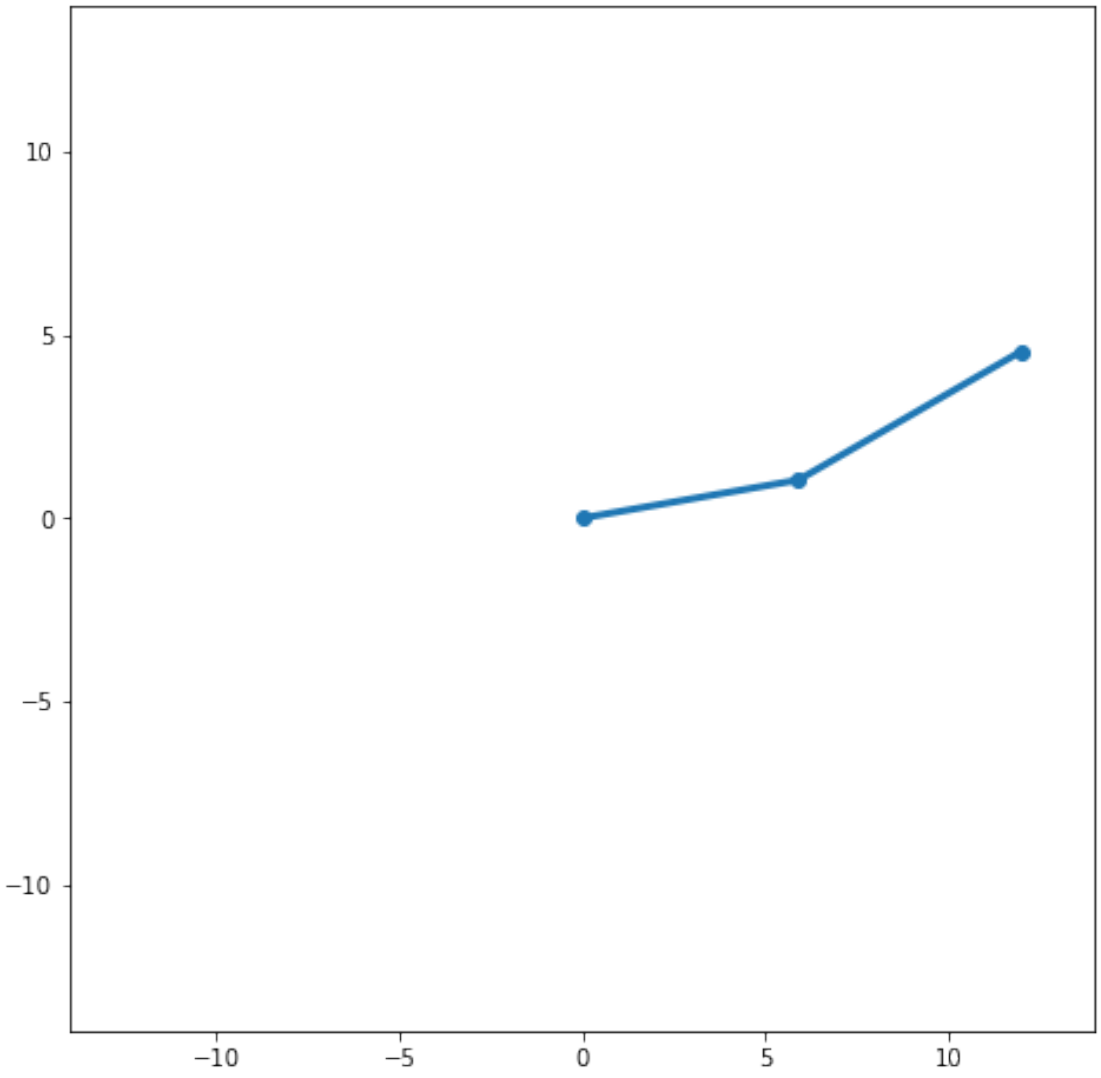
[24]:
```
DKPpic(10,20)
```



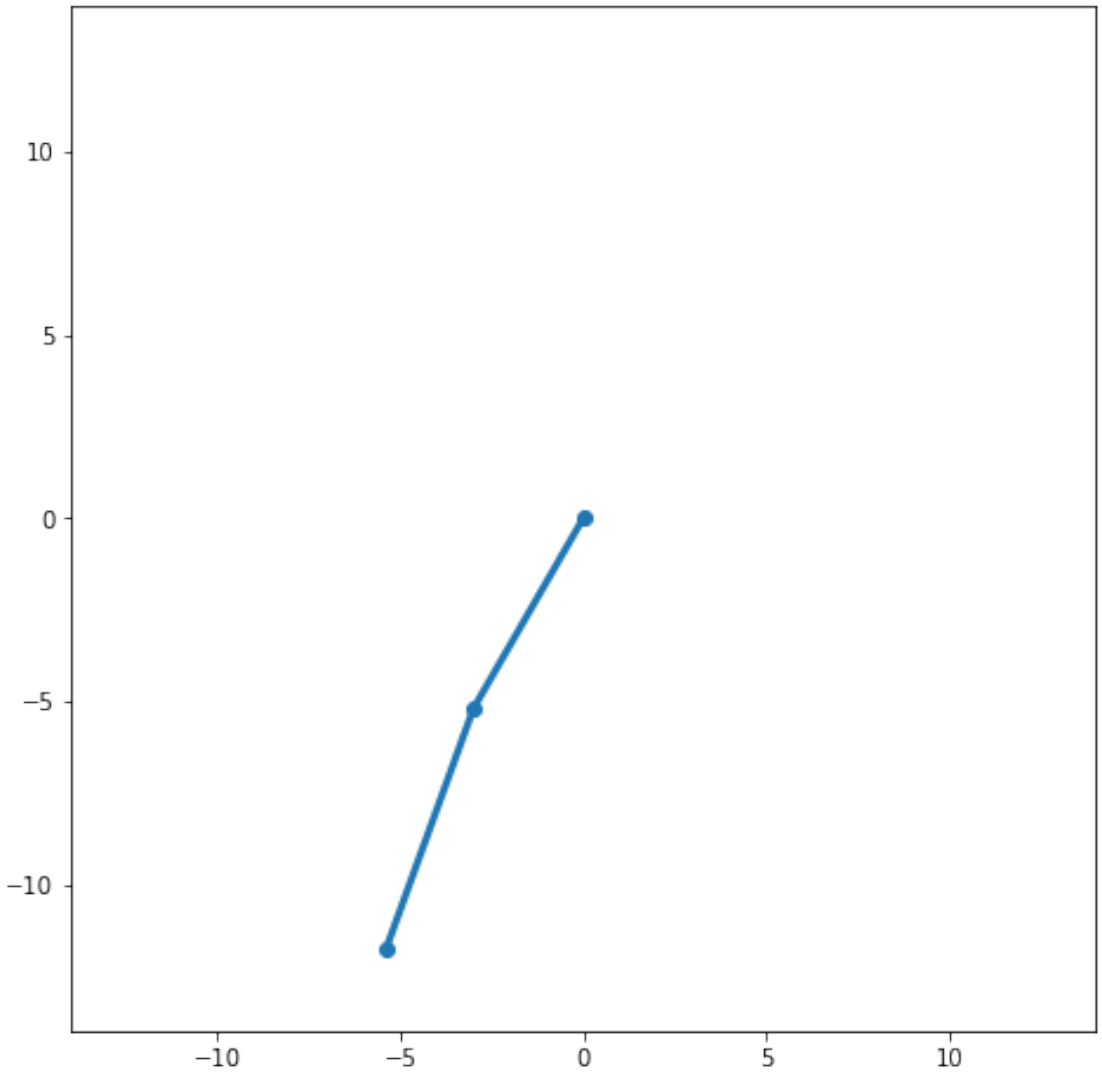
For a comparison of two different poses of the arm it would be nicer to have pictures of the same size. Thus, we modify the definition a bit.

[25]:
```
def DKPpic(t1,t2):
    result = DKPplot(t1,t2)
    fig, ax = plt.subplots(figsize=(8,8))
    ax.set_xlim(( -l_all, l_all))          # define the range of the x-axis;␣
 ↪recall the definition of "l_all".
    ax.set_ylim((-l_all, l_all))
    ax.set_aspect('equal')
    ax.plot(result[0],result[1],'-o', lw=3);

DKPpic(10,20)
```



[26]:
```
DKPpic(240,10)
```



1.3 It is time for animations!

We will need two more commands for the animation.

[27]:
```
from matplotlib import animation    # makes the animation
from IPython.display import HTML    # displays videos in Jupyter notebooks
```

First of all, we create an empty figure for the animation.

[ ]:
```
fig, ax = plt.subplots(figsize=(8,8))
ax.set_xlim(( -l_all, l_all))          # define the range of the x-axis; recall␣
 ↪the definition of "l_all".
ax.set_ylim((-l_all, l_all))
ax.set_aspect('equal')
```

```
ax.set_xlabel('x')
ax.set_ylabel('y')

# create objects that will change in the animation. These are
# initially empty, and will be given new values for each frame
# in the animation.
txt_title = ax.set_title('')
line1, = ax.plot([], [], '-o', lw=2)    # ax.plot returns a list of 2D line␣
 ↪objects,
                                        # so unpack it by "," after "line1"
```

Next, we decide the start and the stop values for the joint angles.

[29]:
```
t1_start, t1_stop = 20, 170
t2_start, t2_stop = 120, 30
```

Like videos, animations consists of single pictures, the **frames**. We use a variable to store the number of frames we want to have:

[30]:
```
num_frames = 240
```

Next, let us split the interval from `t1_start`to `t1_stop` into `num_frames` pieces.

[31]:
```
t1_list = np.linspace(t1_start,t1_stop,num_frames)
t2_list = np.linspace(t2_start,t2_stop,num_frames)
```

We may access the 5th element of the "array" `t1_list` by

[32]:
```
t2_list[4]   # Python starts counting a 0!!!!
```

[32]:
```
118.49372384937239
```

We are ready to define every frame by a function

[33]:
```
def drawframe(n):  # "n" is a counter
    t1, t2 = t1_list[n], t2_list[n]     # actual values for the joint angles
    result = DKPplot(t1,t2)
    line1.set_data(result[0], result[1])
    # txt_title.set_text('Frame = {0:4d}'.format(n))
    txt_title.set_text('$t_1=$'+str(round(t1,1))+'°, $t_2=$'+str(round(t2,1)).
 ↪format(n))
    return (line1)
```

We can define the animation; the assignment `interval=50` will display one frame every 50 milliseconds

[34]:
```
anim = animation.FuncAnimation(fig, drawframe, frames=num_frames, interval=50)
```

It is time to display the result, which is done by `HTML`

[35]:
```
HTML(anim.to_html5_video())
```

[35]: <IPython.core.display.HTML object>

1.4 Animated drawing of a line

First, we have to solve the inverse kinematic problem; we will use one solution only; there will be no test for impossible positions of the tool. Notice that we have

$$\theta_2 = 180 - \arccos \frac{l_1^2 + l_2^2 - x^2 - y^2}{2 l_1 l_2} \tag{7}$$

$$\theta_1 = \operatorname{arctan2}(y, x) - \arccos \frac{x^2 + y^2 + l_1^2 - l_2^2}{2 l_1 \sqrt{x^2 + y^2}} \tag{8}$$

[36]:
```
def IKP(x,y):
    # returns the angles $\theta_1, \theta_2, \theta_3$ (in degrees) for the␣
 ↪position $x,y,\gamma$ of the tool.
    t_2 = 180 - 180/np.pi*np.arccos((l_1**2+l_2**2-x**2-y**2)/(2*l_1*l_2))
    t_1 = 180/np.pi * (np.arctan2(y,x) - np.arccos((x**2+y**2+l_1**2-l_2**2)/
 ↪(2*l_1*np.sqrt(x**2+y**2))))
    return [t_1,t_2]
```

We want to move the tool point from $(-7, 2)$ to $(9, -7)$. To perform the described task, we notice that

$$(x, y) = (1 - t) \cdot (-7, 2) + t \cdot (9, -7); \;\; 0 \leq t \leq 1$$

is a good parametrization for the line between the two points in question. However, it is easier to utilize `np.linspace` again: we get `num_frames` values for the x- and y-values by

[37]:
```
xxx = np.linspace(-7,9,num_frames)
yyy = np.linspace(2,-7,num_frames)
```

The definition of the nth frame is similar to the one above, but we have to compute the angles in each step.

[38]:
```
def drawframe(n):  # "n" is a counter
    xt, yt = xxx[n], yyy[n]
    t1, t2 = IKP(xt,yt)     # actual values for the joint angles
    result = DKPplot(t1,t2)
    line1.set_data(result[0], result[1])
    # txt_title.set_text('Frame = {0:4d}'.format(n))
    txt_title.set_text('$x=$'+str(round(xt,1))+', $y=$'+str(round(yt,1)).
 ↪format(n))
    return (line1)
```

[39]:
```
anim = animation.FuncAnimation(fig, drawframe, frames=num_frames, interval=50)
```

[40]:
```
HTML(anim.to_html5_video())
```

[40]: <IPython.core.display.HTML object>

We want to "trace" the tool point. To this end we have to introduce a second part to the figure, called `line2`.

[ ]:
```
fig, ax = plt.subplots(figsize=(8,8))
ax.set_xlim(( -l_all, l_all))          # define the range of the x-axis; recall␣
 ↪the definition of "l_all".
ax.set_ylim((-l_all, l_all))
ax.set_aspect('equal')
ax.set_xlabel('x')
ax.set_ylabel('y')
xe, ye = [], []
arm, = ax.plot([], [], '-o', lw=3)
drawing, = ax.plot([], [], '-r', lw=1)
```

At the beginning, the trace plot is empty.

[42]:
```
xe, ye = [], []
```

`drawing`gets the coordinates of the tool point in each frame:

[43]:
```
def drawframe(n):  # "n" is a counter
    xt, yt = xxx[n], yyy[n]
    t1, t2 = IKP(xt,yt)     # actual values for the joint angles
    result = DKPplot(t1,t2)
    arm.set_data(result[0], result[1])
    xe.append(result[0][2])
    ye.append(result[1][2])
    drawing.set_data(xe,ye)
    # txt_title.set_text('Frame = {0:4d}'.format(n))
    txt_title.set_text('$x=$'+str(round(xt,1))+', $y=$'+str(round(yt,1)).
 ↪format(n))
    return (arm, drawing)
```

[44]:
```
xe, ye = [], []
anim = animation.FuncAnimation(fig, drawframe, frames=num_frames, interval=50)
```

[45]:
```
HTML(anim.to_html5_video())
```

[45]:
```
<IPython.core.display.HTML object>
```

[46]:
```
xe, ye = [], []
anim.save('linetrace.mp4',writer = 'ffmpeg', fps = 30)
```

1.5 Drawing a line and a segment of a circle

As an example for the project we will "sraw" the letter "D" by composing a line segment from $(-8,9)$ to $(-8,3)$ and a half-circle - this won't look very well, but is an easy start.

The idea for the video is as follows: 1. Wait one second (the arm should not move immediately). 2. Move from the home position to the upper vertex of the line segment. 3. Draw the line segment.

4. Move to the upper vertex of the line segment again. 5. Draw the half circle. 6. Move to the home position. 7. Wait one second and let the arm disappear 8. Wait another second.

The length of the video is set to 10 seconds, so the motions of the arm cover $10 - 3 = 7$ seconds. If the video has 25 fps (with 40 ms per frame), then there are $7 \cdot 25 = 175$ frames for the motion. The approximate distances to move are 1. From $(13, 0)$ (home position) to $(-8, 9)$: 23 cm; 51 frames 2. From $(-8, 9)$ to $(-8, 3)$: 6 cm; 27 frames 3. From $(-8, 3)$ to $(-8, 9)$: 6 cm; 13 frames 4. From $(-8, 9)$ to $(-8, 3)$ on a half circle: 9 cm; 41 frames 5. Back to home position: 18 cm; 40 frames

By using the indicated frame numbers, the velocities for the drawing moves are approximately half of the velocieties for the other moves. The frame numbers add up to 172; the remaining 3 frames are used for the part 8. of the video.

We will first define the numbers of frames for each step and the total number of frames (which should be $250 = 10 \cdot 25$):

[47]:
```
numframes = np.array([25,51,27,13,41,40,25,28])
frame_number = sum(numframes)
frame_number
```

[47]:
```
250
```

The video is split at the partial sums of `numframes`: 25, 25+51=76, 25+51+27=103 etc.:

[48]:
```
split = np.cumsum(numframes)
split
```

[48]:
```
array([ 25,  76, 103, 116, 157, 197, 222, 250])
```

First, we have to wait 25 = 'numframes[0]' frames in home position:

[49]:
```
n = numframes[0]
part0x = (l_1+l_2)*np.ones(n)
part0y = np.zeros(n)
mask0 = np.ones(n) # don't plot these points
```

We compute the required position of the tool point for each frame. Furthermore, we note whether the point is part of the drawing or not. For the first part we have to go on the line segment from $(l_1 + l_2, 0)$ to $(-8, 9)$ in 51 = `numframes[1]` frames.

[50]:
```
n = numframes[1] - 1
part1x = np.array([(1-i/n)*(l_1+l_2)+-i/n*8 for i in range(n+1)])
 ↪  # x-values
part1y = np.array([i/n*9 for i in range(n+1)]) # y-values
mask1 = np.ones(n+1)                                    # don't plot
 ↪these points
print(part1x,'\n',part1y,'\n',mask1)
```

```
[13.   12.58 12.16 11.74 11.32 10.9  10.48 10.06  9.64  9.22  8.8   8.38
  7.96  7.54  7.12  6.7   6.28  5.86  5.44  5.02  4.6   4.18  3.76  3.34
  2.92  2.5   2.08  1.66  1.24  0.82  0.4  -0.02 -0.44 -0.86 -1.28 -1.7
```

```
-2.12 -2.54 -2.96 -3.38 -3.8  -4.22 -4.64 -5.06 -5.48 -5.9  -6.32 -6.74
-7.16 -7.58 -8.  ]
[0.   0.18 0.36 0.54 0.72 0.9  1.08 1.26 1.44 1.62 1.8  1.98 2.16 2.34
2.52 2.7  2.88 3.06 3.24 3.42 3.6  3.78 3.96 4.14 4.32 4.5  4.68 4.86
5.04 5.22 5.4  5.58 5.76 5.94 6.12 6.3  6.48 6.66 6.84 7.02 7.2  7.38
7.56 7.74 7.92 8.1  8.28 8.46 8.64 8.82 9.  ]
[1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
1. 1. 1.]
```

Next, we go from $(-8,9)$ to $(-8,3)$:

[51]:
```
n = numframes[2] - 1
part2y = np.array([(1-i/n)*9+i/n*3 for i in range(n+1)]) # y-values
part2x = -8*np.ones(n+1)                  # x-values are constantly -8
mask2 = np.zeros(n+1)                     # plot these points
```

Go back:

[52]:
```
n = numframes[3] - 1
part3y = np.array([(1-i/n)*3+i/n*9 for i in range(n+1)]) # y-values
part3x = -8*np.ones(n+1)                  # x-values are constantly -8
mask3 = np.ones(n+1)                     # don't plot these points
```

For the half circle around $(-8,6)$ with radius 3, we determine the angles between $\pi/2$ and $-\pi/2$:

[53]:
```
n = numframes[4]
tt = np.linspace(np.pi/2,-np.pi/2,n)
tt
```

[53]:
```
array([ 1.57079633,  1.49225651,  1.41371669,  1.33517688,  1.25663706,
        1.17809725,  1.09955743,  1.02101761,  0.9424778 ,  0.86393798,
        0.78539816,  0.70685835,  0.62831853,  0.54977871,  0.4712389 ,
        0.39269908,  0.31415927,  0.23561945,  0.15707963,  0.07853982,
        0.        , -0.07853982, -0.15707963, -0.23561945, -0.31415927,
       -0.39269908, -0.4712389 , -0.54977871, -0.62831853, -0.70685835,
       -0.78539816, -0.86393798, -0.9424778 , -1.02101761, -1.09955743,
       -1.17809725, -1.25663706, -1.33517688, -1.41371669, -1.49225651,
       -1.57079633])
```

The point corresponding to the angle α has coordinates

$$x = -8 + 3 \cdot \cos(\alpha) \tag{9}$$

$$y = 6 + 3 \cdot \sin(\alpha) \tag{10}$$

It is quite lucky that we can compute these values for the complete array `tt`at once:

[54]:
```
part4x = -8 + 3*np.cos(tt)
part4y = 6 + 3*np.sin(tt)
mask4 = np.zeros(n)
print(part4x,'\n',part4y)
```

```
[-8.         -7.76462271 -7.5306966  -7.29966391 -7.07294902 -6.8519497
 -6.6380285  -6.43250431 -6.23664424 -6.05165586 -5.87867966 -5.7187821
 -5.57294902 -5.44207951 -5.32698043 -5.2283614  -5.14683045 -5.08289024
 -5.03693498 -5.009248   -5.         -5.009248   -5.03693498 -5.08289024
 -5.14683045 -5.2283614  -5.32698043 -5.44207951 -5.57294902 -5.7187821
 -5.87867966 -6.05165586 -6.23664424 -6.43250431 -6.6380285  -6.8519497
 -7.07294902 -7.29966391 -7.5306966  -7.76462271 -8.        ]
 [9.         8.990752   8.96306502 8.91710976 8.85316955 8.7716386
 8.67301957 8.55792049 8.42705098 8.2812179  8.12132034 7.94834414
 7.76335576 7.56749569 7.3619715  7.1480503  6.92705098 6.70033609
 6.4693034  6.23537729 6.         5.76462271 5.5306966  5.29966391
 5.07294902 4.8519497  4.6380285  4.43250431 4.23664424 4.05165586
 3.87867966 3.7187821  3.57294902 3.44207951 3.32698043 3.2283614
 3.14683045 3.08289024 3.03693498 3.009248   3.        ]
```

We have to go back to the home position; this time, we may use `np.linspace` again:

[55]:
```
n = numframes[5]
part5x = np.linspace(-8,l_1+l_2,n)
part5y = np.linspace(3,0,n)
mask5 = np.ones(n) # don't plot these points
```

Finally, we stay in home position

[56]:
```
n = numframes[6]
part6x = (l_1+l_2)*np.ones(n)
part6y = np.zeros(n)
mask6 = np.ones(n) # don't plot these points
```

It is boring, but we have to concatenate all the arrays

[57]:
```
tool_x = np.concatenate((part0x,part1x,part2x,part3x,part4x,part5x,part6x))
tool_y = np.concatenate((part0y,part1y,part2y,part3y,part4y,part5y,part6y))
mask = np.concatenate((mask0,mask1,mask2,mask3,mask4,mask5,mask6))
```

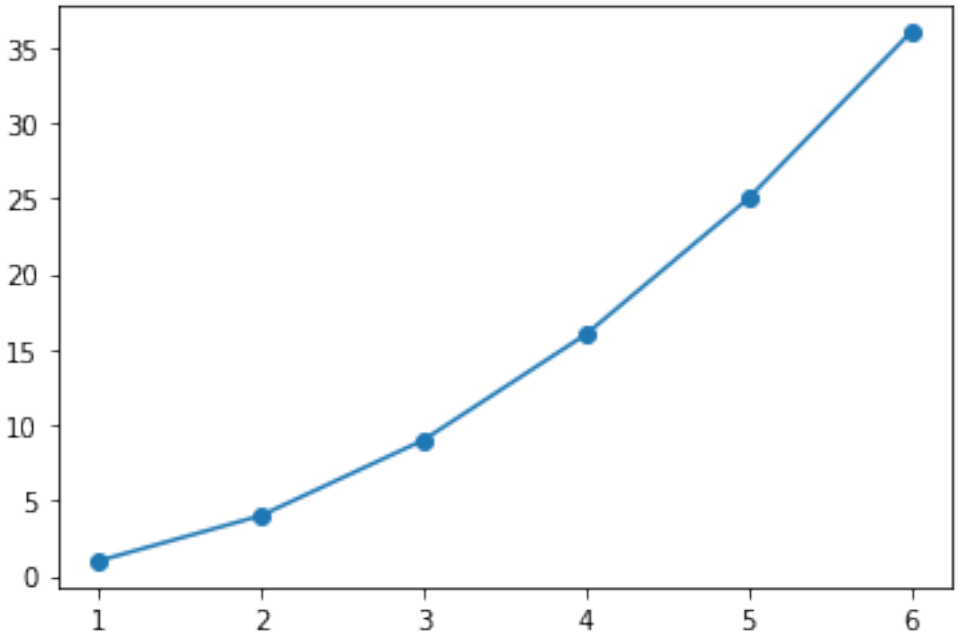
Numpy provides a tool to "mask" arrays, i.e., to mark some of the entries of an array as invalid. The syntax is

```
masked_array = np.ma.masked_array(array, mask = mask_array)
```

where `mask_array` is an array of the same dimension of `array`with 0 and 1 as entries. 0 means that the corresponding entry is valid, 1 means it is invalid.

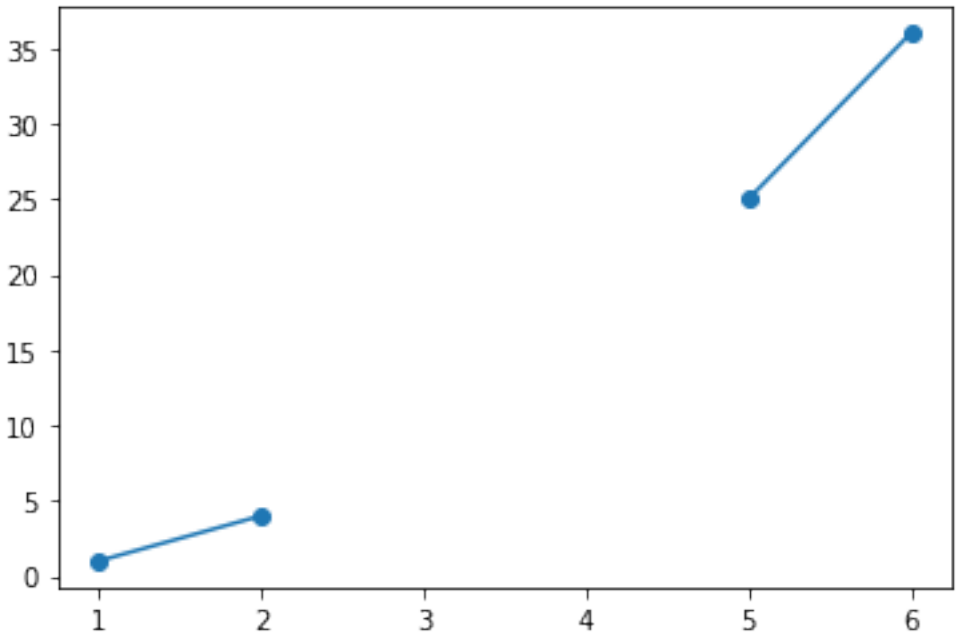
Pyplot will ignore invalid entries:

[58]:
```
test_x = np.array([1,2,3,4,5,6])
test_y = test_x**2
plt.plot(test_x,test_y,'-o');
```



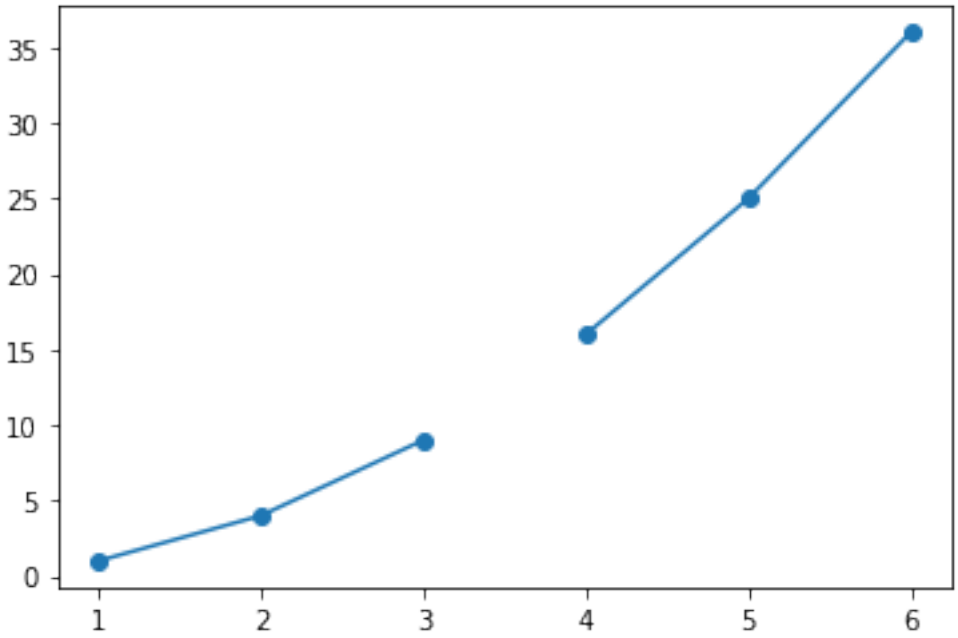
Precisely what we expected! Now we want to delete the 3rd and the 4th point:

[59]:
```
test_mask = np.array([0,0,1,1,0,0])
test_xm = np.ma.masked_array(test_x,mask=test_mask)
plt.plot(test_xm,test_xm**2,'-o');
```



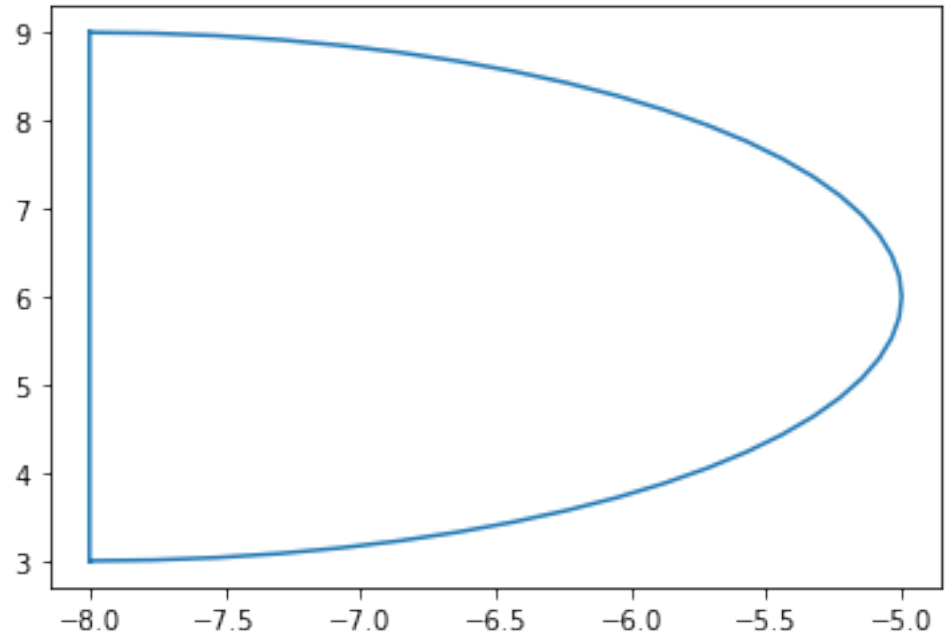
Next, we want to delete the line between the 3rd and the 4th point only. To this end we double the 3rd point:

[60]:
```
test_x = np.array([1,2,3,3,4,5,6])
test_mask = np.array([0,0,0,1,0,0,0])
test_xm = np.ma.masked_array(test_x,mask=test_mask)
plt.plot(test_xm,test_xm**2,'-o');
```



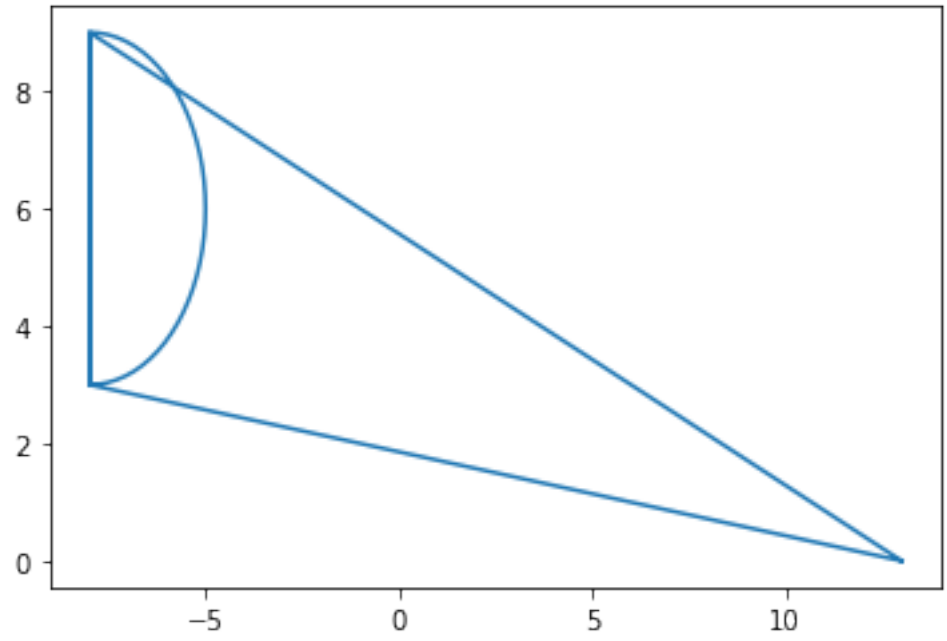
Precisely what we need, right? The preceding discussion should explain the use of the array `mask`. We shall now see if everything is right:

[67]:
```
tool_xm = np.ma.masked_array(tool_x,mask=mask)
tool_ym = np.ma.masked_array(tool_y,mask=mask)
plt.plot(tool_xm,tool_ym);
```



We plot the unmasked arrays for comparison:

[68]:
```
plt.plot(tool_x,tool_y);
```



Next, we construct the frames of the videos. First, we define the empty picture:

[ ]:
```
fig, ax = plt.subplots(figsize=(8,8))
ax.set_xlim(( -l_all, l_all))            # define the range of the x-axis;␣
 ↪recall the definition of "l_all".
ax.set_ylim((-l_all, l_all))
ax.set_aspect('equal')
ax.set_xlabel('x')
ax.set_ylabel('y')

txt_title = ax.set_title('')
arm, = ax.plot([], [], '-o', lw=1)       # the arm
stroke, = ax.plot([], [], '-r', lw=3)    # the strokes
```

Recall the the arm is visible for the first 222 frames only and then should disappear.

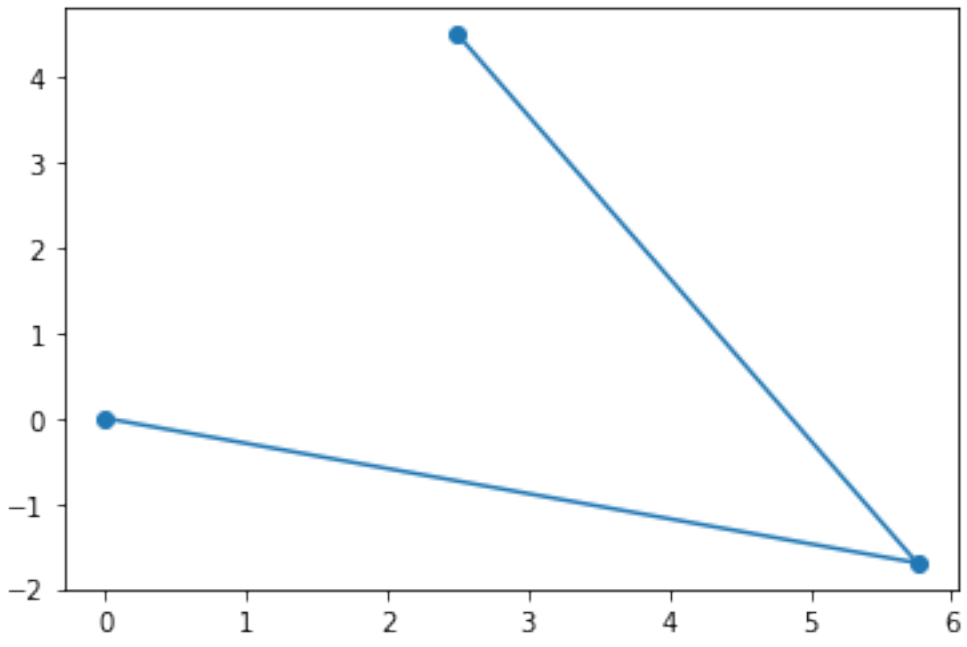
For $n \leq 222$, the position of the elbow and the tool tip has to be evaluated by applying the IKP to `tool_x[n]` and `tool_y[n]`. This task will be done by

```
theta_1, theta_2 = IKP(tool_x[n], tool_y[n])
```

```
result_x, result_y = DKPplot(theta_1,theta_2)
```

As an example:

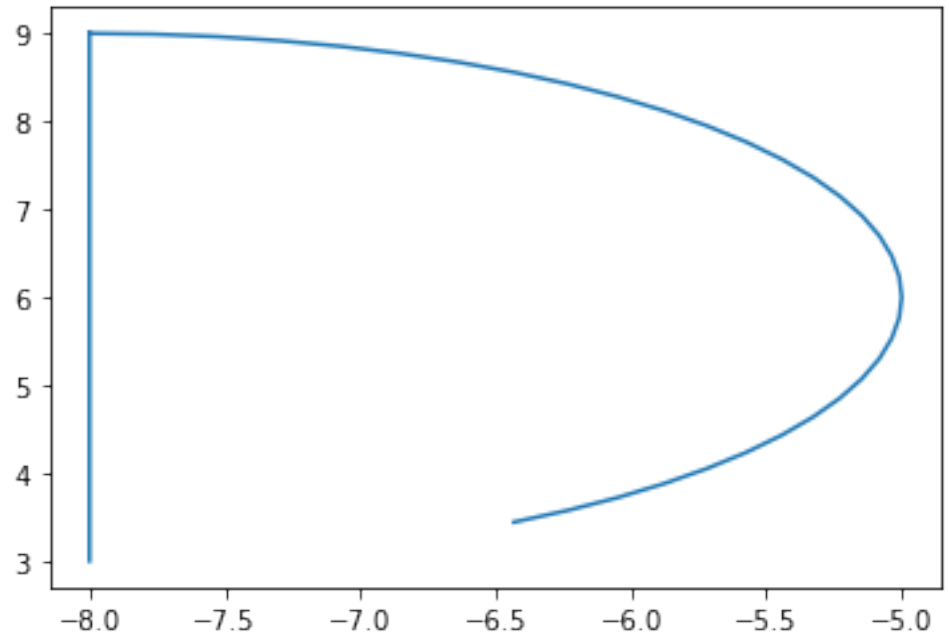
[76]:
```
theta_1, theta_2 = IKP(tool_x[50], tool_y[50])
result_x, result_y = DKPplot(theta_1,theta_2)
plt.plot(result_x,result_y,'-o');
```



Moreover, we have to display all the points drawn so far, i.e., all unmasked points of `tool_xm`,

`tool_ym` until position n. The first n entries of the array `tool_xm` are accessed using `tool_xm[:n]`. We take $n = 150$ as an example.

[88]:
```
plt.plot(tool_xm[:150],tool_ym[:150]);
```



[91]:
```
visible = 222
def drawframe(n):  # "n" is a counter
    # the arm is visible for the first 222 frames
    if n < visible:
        theta_1, theta_2 = IKP(tool_x[n], tool_y[n])
        result_x, result_y = DKPplot(theta_1,theta_2)
        arm.set_data(result_x, result_y)
    else:
        arm.set_data([],[])
    stroke.set_data(tool_xm[:min(n,visible-1)],tool_ym[:min(n,visible-1)])
    # avoid to run out of the array by taking the minimum
    txt_title.set_text('Frame = {0:4d}'.format(n))
    return (arm)
anim = animation.FuncAnimation(fig, drawframe, frames=num_frames, interval=40)
HTML(anim.to_html5_video())
```

[91]:
```
<IPython.core.display.HTML object>
```

It works! The “D” doesn’t look very nice, but this is one possibility to write some character.

Of course you can play around and try to make another character / a lerger character / a second character...

Have fun!

Finally, we want to save the video as an mp4-file, as before:

[92]:
```
anim.save('arm_draws_D.mp4',writer = 'ffmpeg', fps = 25)
```