
Minipupper Kinematics

Horst Dumcke

Jul 14, 2022

CONTENTS

1	2D Coordinate systems	3
1.1	Define a coordinate system as origin of the upper leg	3
1.2	Upper leg	3
1.3	Draw reach of upper leg	4
1.4	Coordinate system for the upper leg	5
1.5	Location of the foot	5
1.6	Make some drawings	5
1.7	Inverse Kinematics	7
1.8	Brute force solution	8
1.9	Geometric solution	8
1.10	Calculate γ	11
1.11	Calculate θ	11
1.12	Draw reach of leg	12
2	Unified Robotics Description Format	15
2.1	Reference	15
2.2	Length of legs	20
2.3	Urdfpy	21
3	2D Leg Movement	23
3.1	Forward Kinematics	23
3.2	Inverse Kinematics	26
3.3	Push Up	27
3.4	Deployment	30
4	Minipupper leg movements	33
4.1	Minipupper leg and servos	33
4.2	Special case: Parallelogram	36
4.3	Angle limits	37
5	Discussion of a paper	39
5.1	Swing Phase	40
5.2	Support Phase	42
5.3	Sigmoid function	43
5.4	Putting everything together	46
6	Gaits for Minipupper	49
6.1	Leg position for testing	53
7	3D Coordinate Systems	55
7.1	Conventions	55

7.2	Rotation in 3D	55
7.3	Transformations in 3D	57
7.4	Pose	57
7.5	Python transformation libraries	57
8	Rigid-body kinematics	61
8.1	Reference	61
8.2	Code	61
8.3	Analytic Inverse Kinematics	61
8.4	Numerical Inverse Kinematics	69
8.5	Robot Orientation	69
9	A Fresh Look at Gaits	77
9.1	Reference	77
9.2	Foot Trajectory planning	77
10	Parameters	81
10.1	Velocity	81
10.2	Direction	81
10.3	Number of points in trajectory	81
10.4	Frequency	81
10.5	Step length	82
10.6	Step height	82
10.7	Gait Scheduling	83
11	Visualization and Simulation Environments	89
11.1	Bullet Real-Time Physics Simulation	89
11.2	Open Dynamics Engine	89
11.3	Dynamic Animation and Robotics Toolkit	89
11.4	Open Robot Control Software	89
11.5	Gazebo	89
11.6	Todo	89
12	Software frameworks for the development of robotic systems	91
12.1	Rocks	91
12.2	ROS - Robot Operating System	91

About this book

This book contains my notes and Jupyter notebooks that I created while gaining a better understanding of the kinematics of a quadruped robot in general and [MangDang Minipupper](#) in particular.

We cover the mathematics for 2D and 3D movements and develop different gaits for the robot. The Jupyter notebooks contain the code that will illustrate the theoretical background. But theory and reality do not always match. We provide a simulation environment using the real-time physics simulation [pybullet](#) and an execution environment that can be installed on a Minipupper so that we can upload leg trajectories to the physical robot and have them executed. We also provide a web-based controller so that we can control the simulated or physical Minipupper from any web browser including a smartphone.

- *2D Coordinate systems*
- *Unified Robotics Description Format*
- *2D Leg Movement*
- *Minipupper leg movements*
- *Discussion of a paper*
- *Gaits for Minipupper*
- *3D Coordinate Systems*
- *Rigid-body kinematics*
- *A Fresh Look at Gaits*
- *Visualization and Simulation Environments*
- *Software frameworks for the development of robotic systems*

A word of caution

This work is a product of my learning effort, not the product of an expert in the field. If I would write it again I would certainly to things differently. But I still hope it will be of use to the reader.

How to install

Installation on a PC

The goal is to focus on portable Python code to be able to run it in any Python virtual environment. We make heavy use of [Jupyter Lab](#)

- Clone the repository
- Create a Python virtual environment
- cd into the repository
- run “./install.sh”

Use the development environment

- cd into the repository
- start Jupyter lab with “jupyter lab jupyternb”

Use the simulation environment

- run “minipupper –help” for instructions on how to use this command
- run “web-controller” and point your browser at the URL shown on the screen

Installation on Minipupper

Use a SD card that has been configured with [minipupper_base](#) Use Ubuntu 22.04 as this is the only version we have tested with so far.

- Clone this repository
- cd into the repository
- run ./install.sh
- reboot
- point your web browser at <http://xx.xx.xx.xx:8080> where xx.xx.xx.xx is the IP address of your minipupper

For development your can also:

- run “minipupper execute –help”
- run “minipupper walk” and adjust parameters for your use case

2D COORDINATE SYSTEMS

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

```
import sympy as sp
from sympy.abc import theta, gamma
from sympy.vector import CoordSys3D
```

```
import math as math
```

```
upper_leg_length = 3
lower_leg_length = 4
```

1.1 Define a coordinate system as origin of the upper leg

```
U = CoordSys3D('U')
```

1.2 Upper leg

The upper leg is connected to the origin of this coordinate system and rotated by an angle of θ . Calculate the location of the end of the upper leg

```
lu = upper_leg_length*sp.cos(theta)*U.i + upper_leg_length*sp.sin(theta)*U.j
print(lu)
```

```
(3*cos(theta))*U.i + (3*sin(theta))*U.j
```

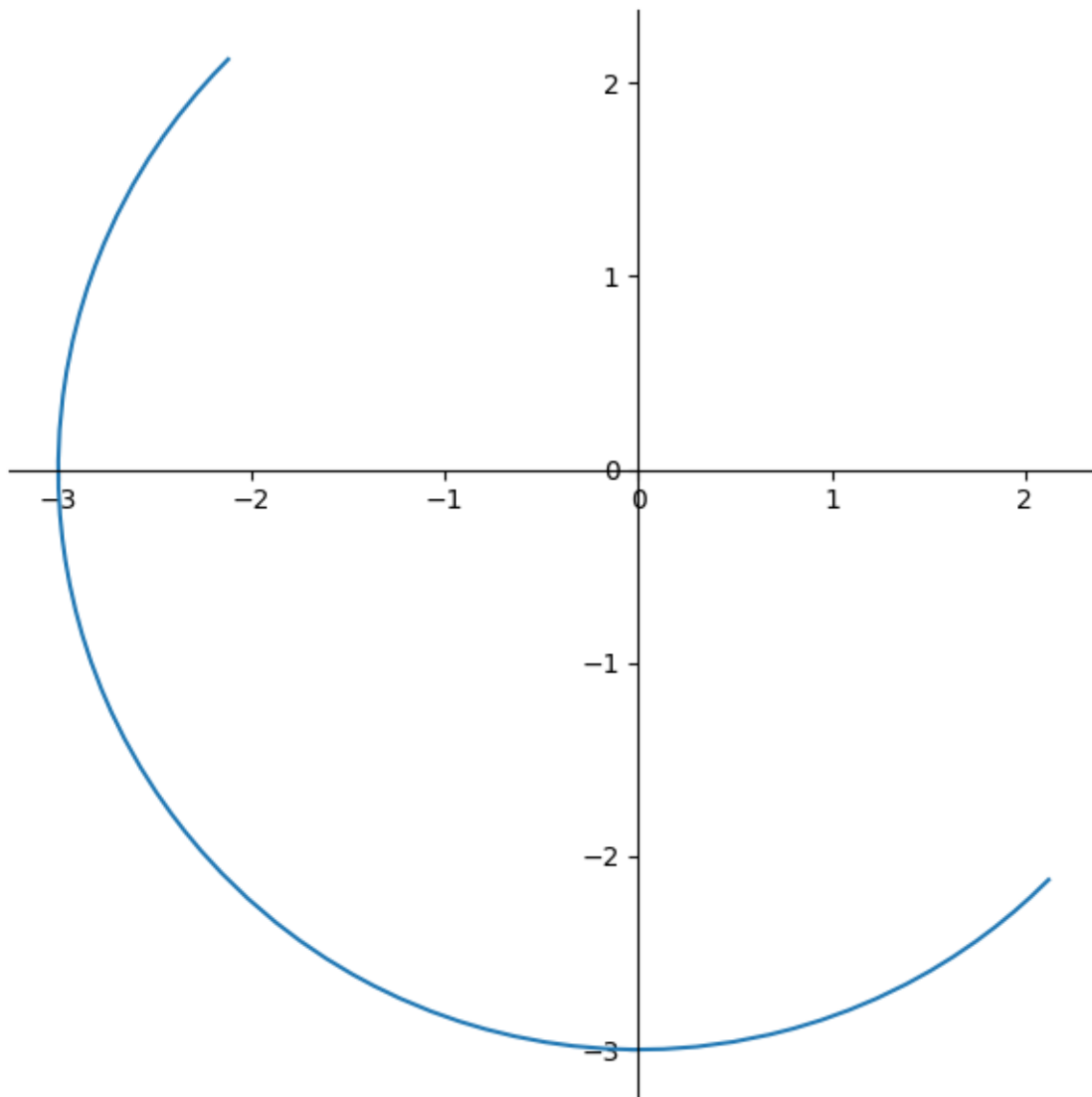
1.3 Draw reach of upper leg

We assume that neutral is -135 degree and the servo can move +/- 90 degree

```
from sympy.plotting import plot_parametric
```

```
deg135 = 3*sp.pi.evalf()/4  
deg90 = sp.pi.evalf()/2
```

```
plot_parametric((upper_leg_length*sp.cos(theta), upper_leg_length*sp.sin(theta)),  
↪(theta, -deg90-deg135, deg90-deg135), size=(6.0, 6.0))
```



```
<sympy.plotting.plot.Plot at 0x1237d6fa0>
```


1.4 Coordinate system for the upper leg

Attach a new coordinate system to the end of the upper leg and the x-axis having the same direction of the upper leg

```
L = U.orient_new_axis('L', theta, U.k, location=lu)
```

1.5 Location of the foot

Calculate the location of the end of the lower leg in this coordinate system

```
ll = lower_leg_length*sp.cos(gamma)*L.i + lower_leg_length*sp.sin(gamma)*L.j
print(ll)
```

```
(4*cos(gamma))*L.i + (4*sin(gamma))*L.j
```

1.6 Make some drawings

```
t = np.radians(-135)
g = np.radians(80)
lu_example = lu.evalf(subs={theta: t})
ll_example = ll.evalf(subs={gamma: g})
```

```
# lower leg coordinate system seen in upper leg coordinate system
Lx = sp.vector.express(L.i, U)
Ly = sp.vector.express(L.j, U)
Lx_example = Lx.evalf(subs={theta: t})
Ly_example = Ly.evalf(subs={theta: t})
```

```
points = np.zeros((2, 2))
points[0,1] = float(lu_example.components.get(U.i, 0))
points[1,1] = float(lu_example.components.get(U.j, 0))

fig, ax = plt.subplots()
plt.xlim(-4,4) #<-- set the x axis limits
plt.ylim(-4,4) #<-- set the y axis limits
# plot a black point at the origin
plt.plot(0,0,'ok')
# plot upper leg
plt.plot(points[0], points[1], '-k')
# plot angle
angles = np.linspace(0, float(t))
xs = 0.5 * np.cos(angles)
ys = 0.5 * np.sin(angles)
plt.plot(xs, ys, '-b')
plt.text(xs[int(len(angles)/2)]+0.1, ys[int(len(angles)/2)]-0.2, "$\\theta$ (%s$^{\circ}$)" % math.degrees(t))
# plot a black point at the joint between upper and lower leg
plt.plot(points[0,1], points[1,1], 'ok')
```

(continues on next page)

(continued from previous page)

```

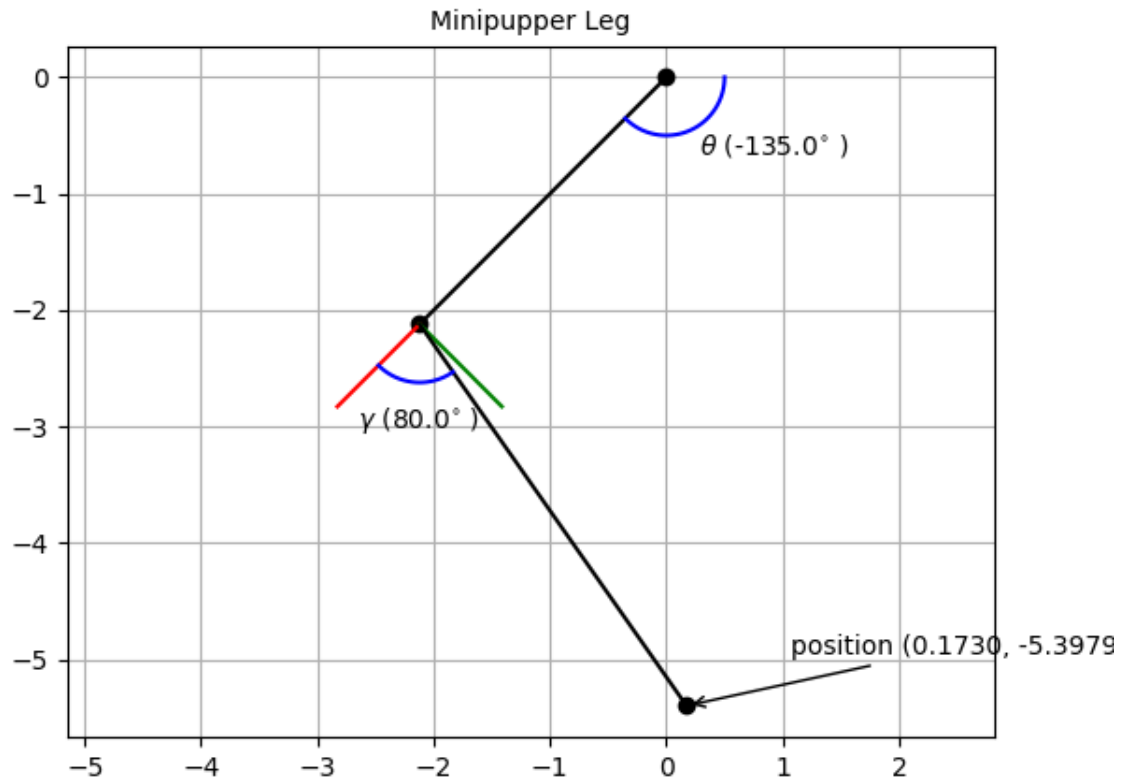
# show coordinate system for lower leg
points[:,0] = points[:,1]
points[0,1] = float((lu_example+Lx_example).components.get(U.i, 0))
points[1,1] = float((lu_example+Lx_example).components.get(U.j, 0))
plt.plot(points[0], points[1], '-r')
points[0,1] = float((lu_example+Ly_example).components.get(U.i, 0))
points[1,1] = float((lu_example+Ly_example).components.get(U.j, 0))
plt.plot(points[0], points[1], '-g')
# plot lower leg
points[0,1] = float((lu_example+sp.vector.express(ll_example, U).evalf(subs={theta: t}
↵)).components.get(U.i, 0))
points[1,1] = float((lu_example+sp.vector.express(ll_example, U).evalf(subs={theta: t}
↵)).components.get(U.j, 0))
plt.plot(points[0], points[1], '-k')
# plot angle
angles = np.linspace(float(t), float(t)+float(g))
xs = points[0,0] + 0.5 * np.cos(angles)
ys = points[1,0] + 0.5 * np.sin(angles)
plt.plot(xs, ys, '-b')
plt.text(xs[int(len(angles)/2)]-0.5, ys[int(len(angles)/2)]-0.4, "$\gamma$ (%s$^{\circ}$
↵circ)$" % math.degrees(g))
# plot a black point at the end of the lower leg
plt.plot(points[0,1],points[1,1],'ok')
ax.annotate(
    "position (%.4f, %.4f)" % (points[0,1],points[1,1]),
    xy=(points[0,1],points[1,1]), xycoords='data',
    xytext=(40, 20), textcoords='offset points',
    arrowprops=dict(arrowstyle="→"))

plt.axis('equal')  #<-- set the axes to the same scale

plt.grid(visible=True, which='major')
plt.title('Minipupper Leg', fontsize=10)

```

```
Text(0.5, 1.0, 'Minipupper Leg')
```



1.7 Inverse Kinematics

Given a point in the base coordinate system calculate theta and gamma so that the end point of the lower leg is positioned at that point

There are points with multiple solutions and there are points that can not be reached. Calculate the region that is reachable

```
P = lu + sp.vector.express(11, U)
print(P)
```

```
(-4*sin(gamma)*sin(theta) + 4*cos(gamma)*cos(theta) + 3*cos(theta))*U.i +
↪ (4*sin(gamma)*cos(theta) + 4*sin(theta)*cos(gamma) + 3*sin(theta))*U.j
```

1.8 Brute force solution

We can derive a system of two nonlinear equations and use a solver to find the solutions. This is quite computational intense and you have to uncomment the line to run it

```
P1 = P.components[U.i]
P2 = P.components[U.j]
print(P1)
print(P2)
```

```
-4*sin(gamma)*sin(theta) + 4*cos(gamma)*cos(theta) + 3*cos(theta)
4*sin(gamma)*cos(theta) + 4*sin(theta)*cos(gamma) + 3*sin(theta)
```

```
solution = []
#solution = sp.solvers.solve((P1-points[0,1], P2-points[1,1]), (theta, gamma))
```

```
for i in range(len(solution)):
    print("theta: %s, gamma: %s" % (math.degrees(solution[i][0]), math.
    degrees(solution[i][1])))
```

1.9 Geometric solution

We will use trigonometric functions to solve the inverse kinematic problem

We introduce a virtual leg l_v that spans from the origin of our coordinate system to the point P with coordinates (x, y).

For simplicity we denote in the following calculations l_u , l_l , l_v as the length of the vectors l_u , l_l , l_v .

We will use the law of cosines

The length of the virtual leg is

$$l_v = \sqrt{x^2 + y^2}$$

and the law of cosines provides

$$l_v^2 = l_u^2 + l_l^2 - 2l_u l_l \cos(180 - \gamma)$$

With $\cos(180 - \gamma) = -\cos(\gamma)$ and the above given length of l_v we obtain

$$\cos(\gamma) = \frac{x^2 + y^2 - l_u^2 - l_l^2}{2l_u l_l}$$

Thus

$$\gamma = \arccos\left(\frac{x^2 + y^2 - l_u^2 - l_l^2}{2l_u l_l}\right)$$

Further we know that

$$x = l_u \cos(\theta) + l_l \cos(\theta + \gamma)$$

$$y = l_u \sin(\theta) + l_l \sin(\theta + \gamma)$$

With $\cos(a + b) = \cos(a) \cos(b) - \sin(a) \sin(b)$ and $\sin(a + b) = \cos(a) \sin(b) + \cos(b) \sin(a)$ we obtain

$$x = l_u \cos(\theta) + l_l \cos(\theta) \cos(\gamma) - l_l \sin(\theta) \sin(\gamma)$$

$$y = l_u \sin(\theta) + l_l \cos(\theta) \sin(\gamma) + l_l \cos(\gamma) \sin(\theta)$$

We now have a system of two linear equations with two variables $\sin(\theta)$ and $\cos(\theta)$

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} l_u + l_l \cos(\gamma) & -l_l \sin(\gamma) \\ l_l \sin(\gamma) & l_u + l_l \cos(\gamma) \end{bmatrix} = \begin{bmatrix} C \\ S \end{bmatrix}$$

where $C = \cos(\theta)$ and $S = \sin(\theta)$. Knowing that

$$\tan(\theta) = \frac{\sin(\theta)}{\cos(\theta)}$$

We calculate

$$\theta = \arctan\left(\frac{S}{C}\right)$$

1.9.1 References

The following sources helped me to refine my understanding

Lecture from Howie Choset, Carnegie Mellon

Paper from Dhaivat Dholakiya et al

```
points = np.zeros((2, 2))
points[0,1] = float(lu_example.components.get(U.i, 0))
points[1,1] = float(lu_example.components.get(U.j, 0))

fig, ax = plt.subplots()
plt.xlim(-4,4) #<-- set the x axis limits
plt.ylim(-4,4) #<-- set the y axis limits
# plot a black point at the origin
plt.plot(0,0,'ok')
# plot upper leg
plt.plot(points[0], points[1], '-k')
plt.text(points[0,1]/2+0.3, points[1,1]/2, 'lu')
# plot angle
angles = np.linspace(0, float(t))
xs = 0.5 * np.cos(angles)
ys = 0.5 * np.sin(angles)
plt.plot(xs, ys, '-b')
plt.text(xs[int(len(angles)/2)]+0.1, ys[int(len(angles)/2)]-0.2, "$\\theta$ (%s^{\r{circ}})$" % math.degrees(t))
# plot a black point at the joint between upper and lower leg
plt.plot(points[0,1], points[1,1], 'ok')
# show coordinate system for lower leg
points[:,0] = points[:,1]
points[0,1] = float((lu_example+Lx_example).components.get(U.i, 0))
points[1,1] = float((lu_example+Lx_example).components.get(U.j, 0))
plt.plot(points[0], points[1], '-r')
points[0,1] = float((lu_example+Ly_example).components.get(U.i, 0))
points[1,1] = float((lu_example+Ly_example).components.get(U.j, 0))
plt.plot(points[0], points[1], '-g')
```

(continues on next page)

(continued from previous page)

```

# plot lower leg
points[0,1] = float((lu_example+sp.vector.express(ll_example, U).evalf(subs={theta: t}
↵)).components.get(U.i, 0))
points[1,1] = float((lu_example+sp.vector.express(ll_example, U).evalf(subs={theta: t}
↵)).components.get(U.j, 0))
plt.plot(points[0], points[1], '-k')
plt.text(points[0,0]+points[1,0]/2+0.3, points[0,1]+points[1,1]/2, 'll')
# plot angle
angles = np.linspace(float(t), float(t)+float(g))
xs = points[0,0] + 0.5 * np.cos(angles)
ys = points[1,0] + 0.5 * np.sin(angles)
plt.plot(xs, ys, '-b')
plt.text(xs[int(len(angles)/2)]-0.5, ys[int(len(angles)/2)]-0.4, "$\gamma$ (%s$^\circ$
↵circ)$" % math.degrees(g))
# plot a black point at the end of the lower leg
plt.plot(points[0,1],points[1,1],'ok')
ax.annotate(
    "position (%.4f, %.4f)" % (points[0,1],points[1,1]),
    xy=(points[0,1],points[1,1]), xycoords='data',
    xytext=(40, 20), textcoords='offset points',
    arrowprops=dict(arrowstyle="→"))

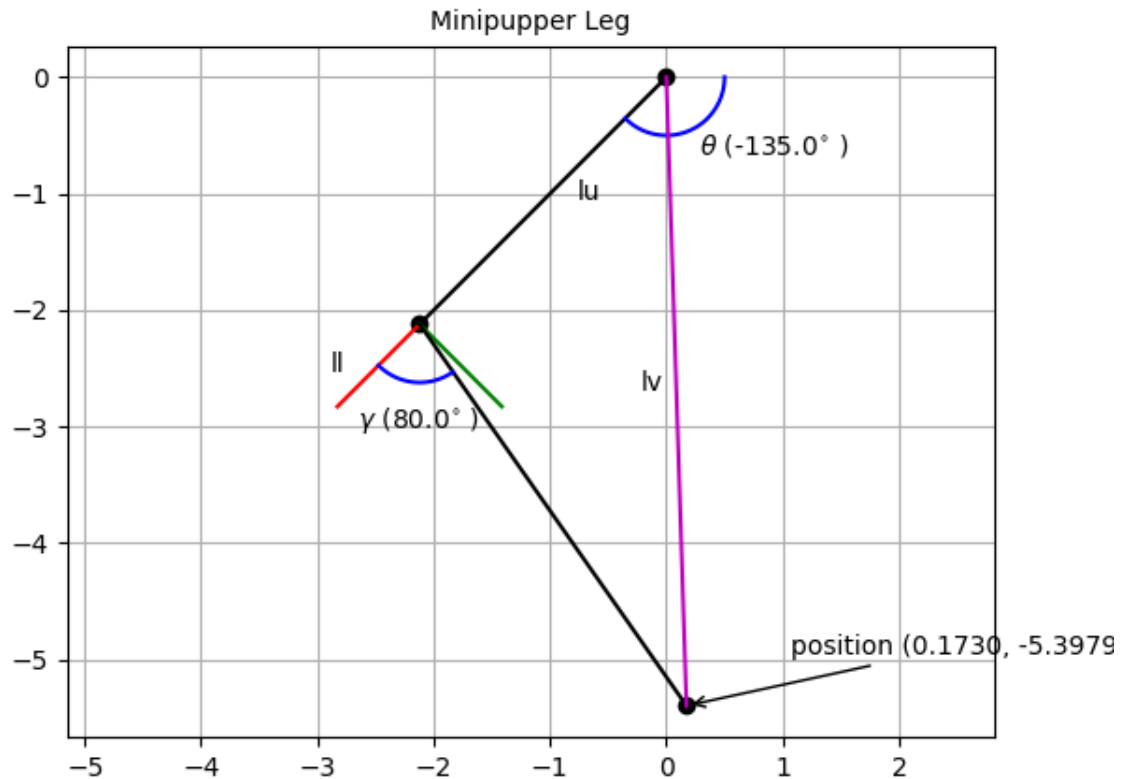
# plot a vector from the origin to the foot
new_points = np.zeros((2, 2))
new_points[0,1] = points[0,1]
new_points[1,1] = points[1,1]
plt.plot(new_points[0], new_points[1], '-m')
plt.text(new_points[0,1]/2-0.3, new_points[1,1]/2, 'lv')

plt.axis('equal') #<-- set the axes to the same scale

plt.grid(visible=True, which='major')
plt.title('Minipupper Leg', fontsize=10)

```

```
Text(0.5, 1.0, 'Minipupper Leg')
```



1.10 Calculate γ

$$\gamma = \arccos\left(\frac{x^2 + y^2 - l_u^2 - l_l^2}{2l_u l_l}\right)$$

```
gam = np.arccos((points[0,1]**2+points[1,1]**2-upper_leg_length**2-lower_leg_
length**2)/(2*upper_leg_length*lower_leg_length))
print(math.degrees(gam))
```

```
80.000000000000001
```

1.11 Calculate θ

Solving

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} l_u + l_l \cos(\gamma) & -l_l \sin(\gamma) \\ l_l \sin(\gamma) & l_u + l_l \cos(\gamma) \end{bmatrix} = \begin{bmatrix} C \\ S \end{bmatrix}$$

or

$$B = \begin{bmatrix} x \\ y \end{bmatrix}$$

$$X = \begin{bmatrix} X_1 \\ X_2 \end{bmatrix} = \begin{bmatrix} C \\ S \end{bmatrix}$$

$$B = AX$$

$$X = A^{-1}B$$

and calculating θ

$$\theta = \arctan\left(\frac{X_2}{X_1}\right)$$

```
B = np.array([points[0,1], points[1,1]])
A11 = upper_leg_length + lower_leg_length * np.cos(gam)
A12 = -lower_leg_length * np.sin(gam)
A21 = lower_leg_length * np.sin(gam)
A22 = upper_leg_length + lower_leg_length * np.cos(gam)

A = np.array([[A11, A12], [A21, A22]])
```

```
X = np.linalg.inv(A).dot(B)
```

```
thet = np.arctan(X[1]/X[0])
print(math.degrees(thet))
print("%s in the 3rd Quadrant corresponds to %s in the first Quadrant" % (math.
degrees(np.pi -thet), math.degrees(thet)))
```

```
44.999999999999986
135.0 in the 3rd Quadrant corresponds to 44.999999999999986 in the first Quadrant
```

1.12 Draw reach of leg

We plot a wide range of servo angles without concern if a physical robot will allow these positions. We observe that some positions cause the end of the upper leg touching the ground and the foot of the leg sticks into the air. If we think about gravity we can see that some positions will put a lot of lateral stress on parts of the leg

```
def draw_leg(ax, l1, l2, t, g):
    points = np.zeros((2, 3))
    points[0,1] = l1*np.cos(t)
    points[1,1] = l1*np.sin(t)
    points[0,2] = l2*np.cos(g+t) + points[0,1]
    points[1,2] = l2*np.sin(g+t) + points[1,1]
    ax.set_aspect('equal')
    ax.set(xlim=(-5, 5), ylim=(-5, 5))
    ax.get_xaxis().set_ticks([])
    ax.get_yaxis().set_ticks([])
    ax.plot(points[0], points[1])
```

```
ncols=10
nrows=10
l1_angles = np.linspace(-5*np.pi/4, np.pi/4, num=ncols)
l2_angles = np.linspace(0.0, np.pi, num=nrows)
```

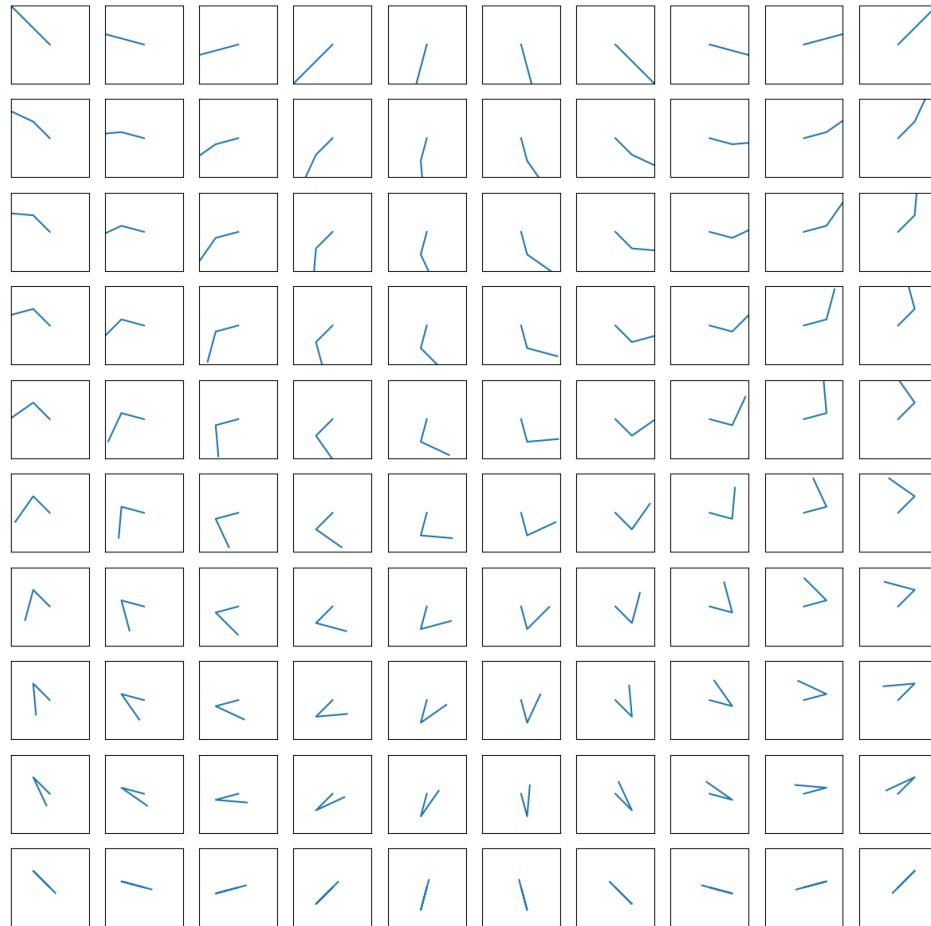
(continues on next page)

(continued from previous page)

```

fig, axs = plt.subplots(ncols=ncols, nrows=nrows, figsize=(15,15))
for row in range(nrows):
    for col in range(ncols):
        draw_leg(axs[row, col], upper_leg_length, lower_leg_length, l1_angles[col],
        ↪l2_angles[row])

```



```

col=5
row=4
print("%s %s" % (np.degrees(l1_angles[col-1]), np.degrees(l2_angles[row-1])))

```

```

-105.00000000000001  59.99999999999999

```


UNIFIED ROBOTICS DESCRIPTION FORMAT

URDF is an XML specification to model robots. We will have a closer look at the urdf file for minipupper

2.1 Reference

The following sources helped me to refine my understanding

Understanding URDF using MATLABR - Peter Corke

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

```
from xml.dom import minidom
```

```
with open('./simple.urdf', 'r') as f:
    xmldoc = minidom.parse(f)
```

```
print(xmldoc.toprettyxml())
```

```
<?xml version="1.0" ?>
<!-- copied from Understanding URDF using MATLABR - Peter Corke -->
<robot name="planar2">

    <link name="base_link"/>

    <link name="link1"/>

    <link name="link2"/>

    <link name="end"/>

    <joint name="q1" type="continuous">
```

(continues on next page)

(continued from previous page)

```
<parent link="base_link"/>

<child link="link1"/>

<axis xyz="0 0 1"/>

</joint>

<joint name="q2" type="continuous">

  <parent link="link1"/>

  <child link="link2"/>

  <origin xyz="1 0 0"/>

  <axis xyz="0 0 1"/>

</joint>

<joint name="notajoint" type="fixed">

  <parent link="link2"/>

  <child link="end"/>

  <origin xyz="1 0 0"/>

</joint>

</robot>
```

```
def parseDom(node):
    if node.nodeType != minidom.Node.ELEMENT_NODE:
        return
    if node.nodeName == 'transmission':
        # ignore transmission and avoid overwritten links
        return
    if len(node.getAttribute("name")) > 0:
```

(continues on next page)

(continued from previous page)

```

urdf[node.getAttribute("name")] = node
if node.nodeName == 'link':
    links.append(node.getAttribute("name"))
if node.nodeName == 'joint':
    joints.append(node.getAttribute("name"))
for child in node.childNodes:
    parseDom(child)

```

```

urdf = {}
links = []
joints = []
parseDom(xmlDoc.documentElement)

```

```

for i in range(len(links)):
    print(links[i])

```

```

base_link
link1
link2
end

```

```

def getJointParent(node):
    for n in urdf[node].childNodes:
        if n.nodeType != minidom.Node.ELEMENT_NODE:
            continue
        if n.nodeName == 'parent':
            return n.getAttribute('link')

```

```

def getJointChild(node):
    for n in urdf[node].childNodes:
        if n.nodeType != minidom.Node.ELEMENT_NODE:
            continue
        if n.nodeName == 'child':
            return n.getAttribute('link')

```

```

for i in range(len(joints)):
    print("%s: parent %s child %s" % (joints[i], getJointParent(joints[i]),
    ↵getJointChild(joints[i])))

```

```

q1: parent base_link child link1
q2: parent link1 child link2
notajoint: parent link2 child end

```

```

with open('../mini-pupper_description/urdf/mini-pupper.urdf', 'r') as f:
    xmlDoc = minidom.parse(f)

```

```

urdf = {}
links = []
joints = []
parseDom(xmlDoc.documentElement)

```

```
def getJointOrigin(node):
    for n in urdf[node].childNodes:
        if n.nodeType != minidom.Node.ELEMENT_NODE:
            continue
        if n.nodeName == 'origin':
            return n.getAttribute('xyz')
```

```
joints_origin = {}
for i in range(len(joints)):
    if 'debug' in joints[i]:
        continue
    if 'inertia' in joints[i]:
        continue
    origin = getJointOrigin(joints[i])
    if origin == None:
        continue
    joints_origin[joints[i]] = [float(x) for x in origin.split(' ')]
    print("%s: parent %s child %s origin %s" % (joints[i], getJointParent(joints[i]),
    ↪getJointChild(joints[i]), origin))
```

```
lf_hip_joint: parent base_link child lf_hip_link origin 0.06014 0.0235 0.0171
lf_upper_leg_joint: parent lf_hip_link child lf_upper_leg_link origin 0 0.0197 0
lf_lower_leg_joint: parent lf_upper_leg_link child lf_lower_leg_link origin 0 0.
↪00475 -0.05
lf_foot_joint: parent lf_lower_leg_link child lf_foot_link origin 0 0 -0.056
lh_hip_joint: parent base_link child lh_hip_link origin -0.05886 0.0235 0.0171
lh_upper_leg_joint: parent lh_hip_link child lh_upper_leg_link origin 0 0.0197 0
lh_lower_leg_joint: parent lh_upper_leg_link child lh_lower_leg_link origin 0 0.
↪00475 -0.05
lh_foot_joint: parent lh_lower_leg_link child lh_foot_link origin 0 0 -0.056
rf_hip_joint: parent base_link child rf_hip_link origin 0.06014 -0.0235 0.0171
rf_upper_leg_joint: parent rf_hip_link child rf_upper_leg_link origin 0 -0.0197 0
rf_lower_leg_joint: parent rf_upper_leg_link child rf_lower_leg_link origin 0 -0.
↪00475 -0.05
rf_foot_joint: parent rf_lower_leg_link child rf_foot_link origin 0 0 -0.056
rh_hip_joint: parent base_link child rh_hip_link origin -0.05886 -0.0235 0.0171
rh_upper_leg_joint: parent rh_hip_link child rh_upper_leg_link origin 0 -0.0197 0
rh_lower_leg_joint: parent rh_upper_leg_link child rh_lower_leg_link origin 0 -0.
↪00475 -0.05
rh_foot_joint: parent rh_lower_leg_link child rh_foot_link origin 0 0 -0.056
```

```
joints_origin
```

```
{'lf_hip_joint': [0.06014, 0.0235, 0.0171],
 'lf_upper_leg_joint': [0.0, 0.0197, 0.0],
 'lf_lower_leg_joint': [0.0, 0.00475, -0.05],
 'lf_foot_joint': [0.0, 0.0, -0.056],
 'lh_hip_joint': [-0.05886, 0.0235, 0.0171],
 'lh_upper_leg_joint': [0.0, 0.0197, 0.0],
 'lh_lower_leg_joint': [0.0, 0.00475, -0.05],
 'lh_foot_joint': [0.0, 0.0, -0.056],
 'rf_hip_joint': [0.06014, -0.0235, 0.0171],
 'rf_upper_leg_joint': [0.0, -0.0197, 0.0],
 'rf_lower_leg_joint': [0.0, -0.00475, -0.05],
```

(continues on next page)

(continued from previous page)

```

'rf_foot_joint': [0.0, 0.0, -0.056],
'rh_hip_joint': [-0.05886, -0.0235, 0.0171],
'rh_upper_leg_joint': [0.0, -0.0197, 0.0],
'rh_lower_leg_joint': [0.0, -0.00475, -0.05],
'rh_foot_joint': [0.0, 0.0, -0.056]}

```

```

locations = ['lf', 'lh', 'rf', 'rh']
# robot origin
xs = [0.0]
ys = [0.0]
zs = [0.0]
# hip
for l in locations:
    loc = "%s_hip_joint" % l
    xs.append(joints_origin[loc][0])
    ys.append(joints_origin[loc][1])
    zs.append(joints_origin[loc][2])
# upper leg
for l in locations:
    loc = "%s_upper_leg_joint" % l
    loc_parent = "%s_hip_joint" % l
    xs.append(joints_origin[loc][0] + joints_origin[loc_parent][0])
    ys.append(joints_origin[loc][1] + joints_origin[loc_parent][1])
    zs.append(joints_origin[loc][2] + joints_origin[loc_parent][2])
# lower leg
for l in locations:
    loc = "%s_lower_leg_joint" % l
    loc_parent = "%s_upper_leg_joint" % l
    loc_parent1 = "%s_hip_joint" % l
    xs.append(joints_origin[loc][0] + joints_origin[loc_parent][0] + joints_
    ↪origin[loc_parent1][0])
    ys.append(joints_origin[loc][1] + joints_origin[loc_parent][1] + joints_
    ↪origin[loc_parent1][1])
    zs.append(joints_origin[loc][2] + joints_origin[loc_parent][2] + joints_
    ↪origin[loc_parent1][2])
# foot
for l in locations:
    loc = "%s_foot_joint" % l
    loc_parent = "%s_lower_leg_joint" % l
    loc_parent1 = "%s_upper_leg_joint" % l
    loc_parent2 = "%s_hip_joint" % l
    xs.append(joints_origin[loc][0] + joints_origin[loc_parent][0] + joints_
    ↪origin[loc_parent1][0] + joints_origin[loc_parent2][0])
    ys.append(joints_origin[loc][1] + joints_origin[loc_parent][1] + joints_
    ↪origin[loc_parent1][1] + joints_origin[loc_parent2][1])
    zs.append(joints_origin[loc][2] + joints_origin[loc_parent][2] + joints_
    ↪origin[loc_parent1][2] + joints_origin[loc_parent2][2])

```

```

import matplotlib.pyplot as plt
import numpy as np

fig = plt.figure()
ax = fig.add_subplot(projection='3d')

ax.scatter(xs, ys, zs)

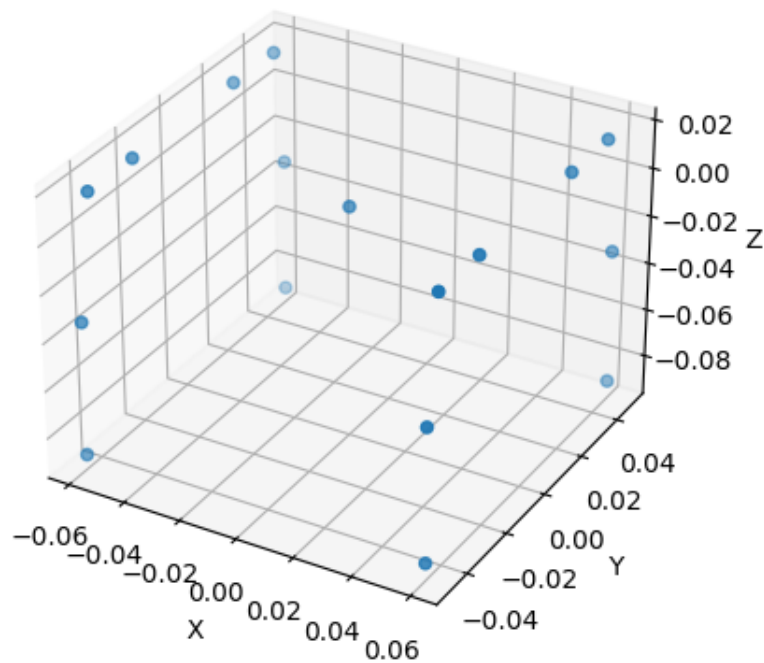
```

(continues on next page)

(continued from previous page)

```
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
```

```
Text(0.5, 0, 'Z')
```



2.2 Length of legs

Calculate the length of the upper and lower leg based on information from the URDF file

We will use this information for our inverse kinematics

```
print("Lenght of upper leg: %s" % np.linalg.norm(np.array(joints_origin['lh_lower_leg_
↪joint'])))
print("Lenght of lower leg: %s" % np.linalg.norm(np.array(joints_origin['lh_foot_joint
↪'])))
```

```
Lenght of upper leg: 0.05022511821787979
Lenght of lower leg: 0.056
```


2.3 Urdfpy

There is more work to do to properly parse a URDF file. We will use a library

Urdfpy Documentation

urdfpy currently requires networkx==2.2 but this version is broken on Python 3.9

pip uninstall networkx

pip install networkx

and ignore the warnings

If you want to display ur5

pip uninstall pyrender

pip install git+https://github.com/mmatl/pyrender.git

```
from urdfpy import URDF

# uncomment the urdf file you want to work on. Leave None if you want to skip this_
↪step
robot=None
# we use a modified version because the original file throws an error
#robot = URDF.load('../mini-pupper_description/urdf/mini-pupper_edited.urdf')
#robot = URDF.load('../simple_visual.urdf')
#robot = URDF.load('../ur5/ur5.urdf')
```

```
if robot is not None:
    for link in robot.links:
        print(link.name)
```

```
if robot is not None:
    for joint in robot.joints:
        print('{} connects {} to {}'.format(
            joint.name, joint.parent, joint.child
        ))
```

```
if robot is not None:
    for joint in robot.actuated_joints:
        print(joint.name)
```

```
if robot is not None:
    if robot.name == 'mini-pupper':
        cfg={
            'lf_hip_joint': 0.0,
            'lh_hip_joint': 0.0,
            'rf_hip_joint': 0.0,
            'rh_hip_joint': 0.0,
            'lf_upper_leg_joint': np.pi/4,
            'lh_upper_leg_joint': np.pi/4,
            'rf_upper_leg_joint': np.pi/4,
            'rh_upper_leg_joint': np.pi/4,
            'lf_lower_leg_joint': -np.pi/2,
```

(continues on next page)

(continued from previous page)

```
        'lh_lower_leg_joint': -np.pi/2,
        'rf_lower_leg_joint': -np.pi/2,
        'rh_lower_leg_joint': -np.pi/2
    }
    if robot.name == 'planar2':
        cfg={'q1': 0.0, 'q2': math.pi/4}
    if robot.name == 'ur5':
        cfg={
            'shoulder_pan_joint': 0.0,
            'shoulder_lift_joint': 2.0,
            'elbow_joint': 2.0,
            'wrist_1_joint': 0.0,
            'wrist_2_joint': 0.0,
            'wrist_3_joint': 0.0,
        }
```

```
import math
# Attention, the following will open a new Window but does not terminate properly.
# ↪ (Mac OS)
if robot is not None:
    robot.show(cfg=cfg)
```

2D LEG MOVEMENT

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

3.1 Forward Kinematics

We can express 2D Transformations as a 3-dimensional homogeneous transformation matrix

$$\begin{bmatrix} \cos(\theta) & -\sin(\theta) & x \\ \sin(\theta) & \cos(\theta) & y \\ 0 & 0 & 1 \end{bmatrix}$$

Where θ is the rotation angle and x,y are the components of the translation vector

In the following we use Spatial Maths for Python from Peter Corke as our Python module

3.1.1 Reference

The following sources helped me to refine my understanding

Robotics Programming Study Guide - Tim Bower

Describing rotation and translation in 2D - Peter Corke

```
from spatialmath import *
```

```
# set parameters according 1-kinematics_2d
theta = np.radians(-135)
gamma = np.radians(80)
lu = 3
ll = 4

# set parameters according http://faculty.salina.k-state.edu/tim/robot_prog/Arm_
↳ robots/forwardKin.html#simple-2-d-example
#theta = np.pi*40/180
#gamma = -np.pi*15/180
#lu = 5
#ll = 2
```

```
# create 2d rotation Matrix for theta
R = SO2(theta)
print(R)
```

```
-0.7071    0.7071
-0.7071   -0.7071
```

```
# create a 2d homogeneous translation matrix
# calculate the translation vector
T = SE2([lu*np.cos(theta), lu*np.sin(theta)])
print(T)
```

```
1      0    -2.121
0      1    -2.121
0      0      1
```

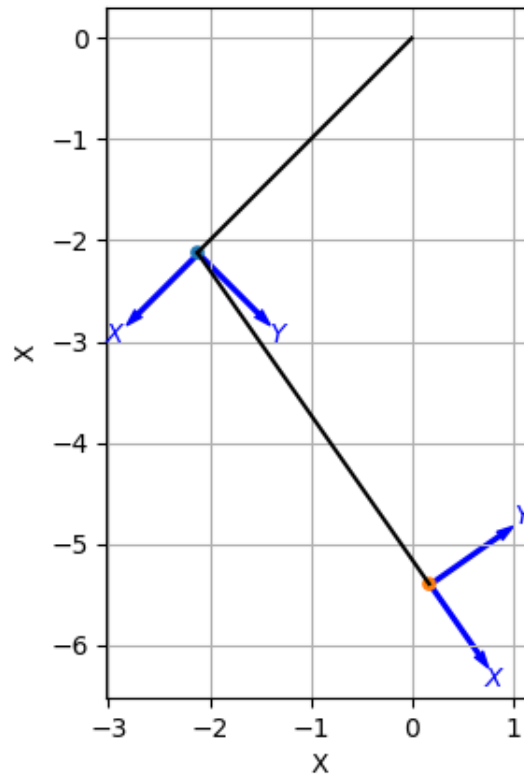
```
# combining the two
T1 = SE2([lu*np.cos(theta), lu*np.sin(theta)]) * SE2(theta)
print(T1)
```

```
-0.7071    0.7071   -2.121
-0.7071   -0.7071   -2.121
0          0          1
```

```
# Transforming from origin to end of upper leg to end of lower leg
T2 = SE2([l1*np.cos(gamma), l1*np.sin(gamma)]) * SE2(gamma)
T3 = T1*T2
print(T3)
```

```
points = np.zeros((2, 2))
fig, ax = plt.subplots()
# plot upper leg
points[0,1] = lu*np.cos(theta)
points[1,1] = lu*np.sin(theta)
plt.plot(points[0], points[1], '-k')
T1.plot()
# plot lower leg
points[0,0] = points[0,1]
points[1,0] = points[1,1]
points[0,1] = lu*np.cos(theta) + l1*np.cos(theta+gamma)
points[1,1] = lu*np.sin(theta) + l1*np.sin(theta+gamma)
plt.plot(points[0], points[1], '-k')
T3.plot()
plt.grid(True)
```

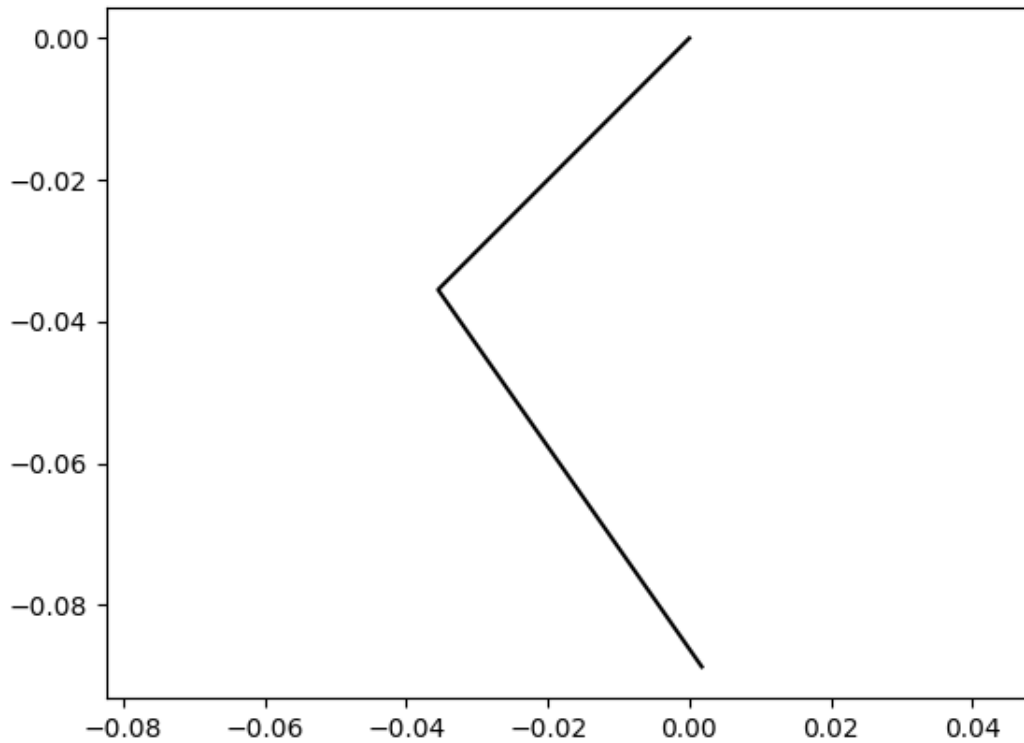
```
0.5736    0.8192    0.173
-0.8192    0.5736   -5.398
0          0          1
```



```
# for Minipupper write function for forward kinematic using only np
def transformationMatrix(angle, l):
    c = np.cos(angle)
    s = np.sin(angle)
    return np.array([[c, -s, l*c], [s, c, l*s], [0, 0, 1]])
def mpForwardKin(angles):
    lu = 0.05022511821787979 # from urdf
    ll = 0.065 # measured, includes the rubber foot
    T1 = transformationMatrix(angles[0], lu)
    T2 = transformationMatrix(angles[1], ll)
    T3 = T1.dot(T2)
    P1 = T1.dot(np.array([0,0,1]))[0:2]
    P2 = T3.dot(np.array([0,0,1]))[0:2]
    return np.array([[0, P1[0], P2[0]], [0, P1[1], P2[1]]])
```

```
points = mpForwardKin([theta, gamma])
fig, ax = plt.subplots()
plt.plot(points[0], points[1], '-k')
plt.axis('equal')

plt.grid(visible=False, which='major')
```



3.2 Inverse Kinematics

We use the calculations from 1-kinematics_2d

```
def mpInverseKin(point):
    lu = 0.05022511821787979 # from urdf
    ll = 0.065 # measured, includes the rubber foot
    gam = np.arccos((point[0]**2+point[1]**2-lu**2-ll**2)/(2*lu*ll))
    B = np.array(point)
    A11 = lu + ll * np.cos(gam)
    A12 = -ll * np.sin(gam)
    A21 = ll * np.sin(gam)
    A22 = lu + ll * np.cos(gam)

    A = np.array([[A11, A12], [A21, A22]])
    X = np.linalg.inv(A).dot(B)
    thet = np.arctan(X[1]/X[0])
    # correction for 2nd and 3rd quadrant
    # we assume -np.pi/4 <= thet <= -3*np.pi/4
    if thet > -np.pi/4:
        thet = thet - np.pi
    return [thet, gam]
```

```
np.testing.assert_almost_equal([theta, gamma], mpInverseKin(mpForwardKin([theta,
gamma]))[:,2]), decimal=7, err_msg='', verbose=True)
```

```
for i in range(1,20):
    theta = -np.pi/4 - i/20 * np.pi
    gamma = i/20 * np.pi
    np.testing.assert_almost_equal([theta, gamma], mpInverseKin(mpForwardKin([theta,
gamma]))[:,2]), decimal=10, err_msg='', verbose=True)
```

3.3 Push Up

We start with a neutral position and then draw a line up and down the y axes. On this line we define a number of points that must be reached, each point in a configurable time interval that is constant for all points.

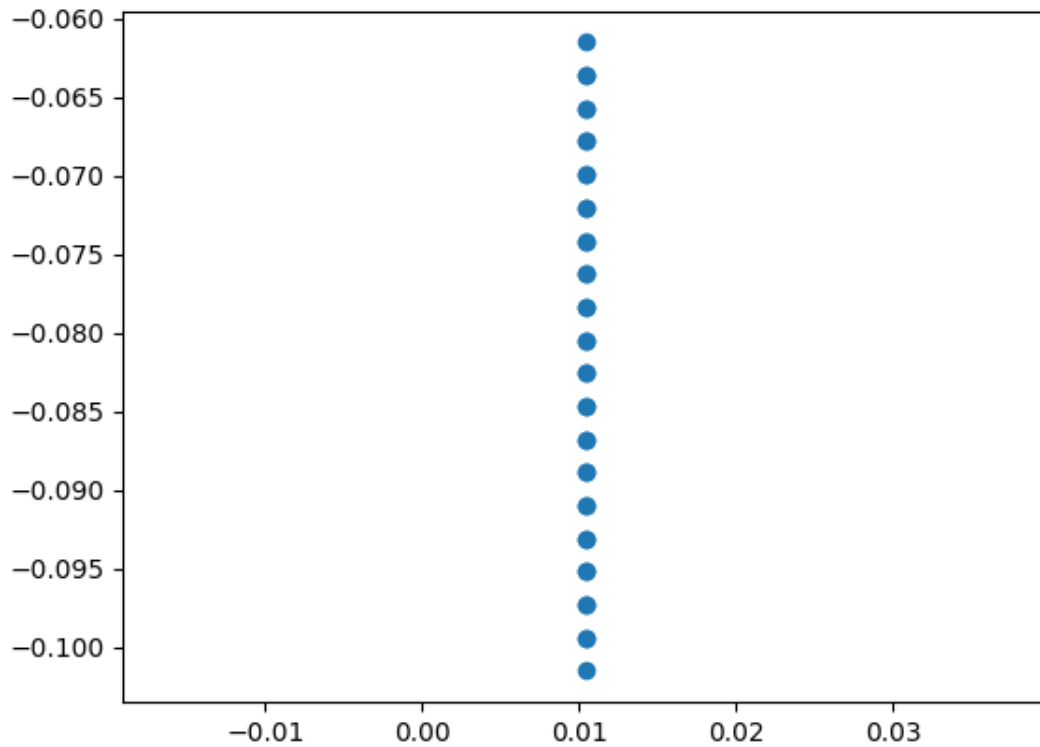
We will take care that we can repeat this leg_servo_positions to build loops

```
theta = -3*np.pi/4
gamma = np.pi/2
midpoint = mpForwardKin([theta, gamma])[:,2]
#
delta = 0.02 # how much up, down
N = 20 # number of points
line = np.linspace(midpoint[1]-delta, midpoint[1]+delta, num=N)
```

```
# create points there to position the foot. We construct a loop and make sure the end
points are only hit once
points = np.full((2, 2*N-2), midpoint[0])
points[1][0:N] = line
points[1][N:] = np.flip(line)[1:N-1]

#plot
fig, ax = plt.subplots()
plt.scatter(points[0], points[1])
plt.axis('equal') #<-- set the axes to the same scale

plt.grid(visible=False, which='major')
```



```
leg_servo_positions = []
for j in range(2*N-2):
    leg_servo_positions.append(mpInverseKin(points[:,j]))
```

```
import matplotlib.animation as animation
from collections import deque

fig, ax = plt.subplots()
plt.xlim(-0.15, 0.15)
plt.ylim(-0.15, 0)
plt.axis('equal')
plt.grid(visible=False, which='major')

line, = ax.plot([], [], 'o-', lw=2)
trace, = ax.plot([], [], '.-', lw=1, ms=2)
step_template = 'step = %.1f'
step_text = ax.text(0.05, 0.9, '', transform=ax.transAxes)
history_len = 50
history_x, history_y = deque(maxlen=history_len), deque(maxlen=history_len)

def animate(i):
    p = mpForwardKin(leg_servo_positions[i])

    if i == 0:
```

(continues on next page)

(continued from previous page)

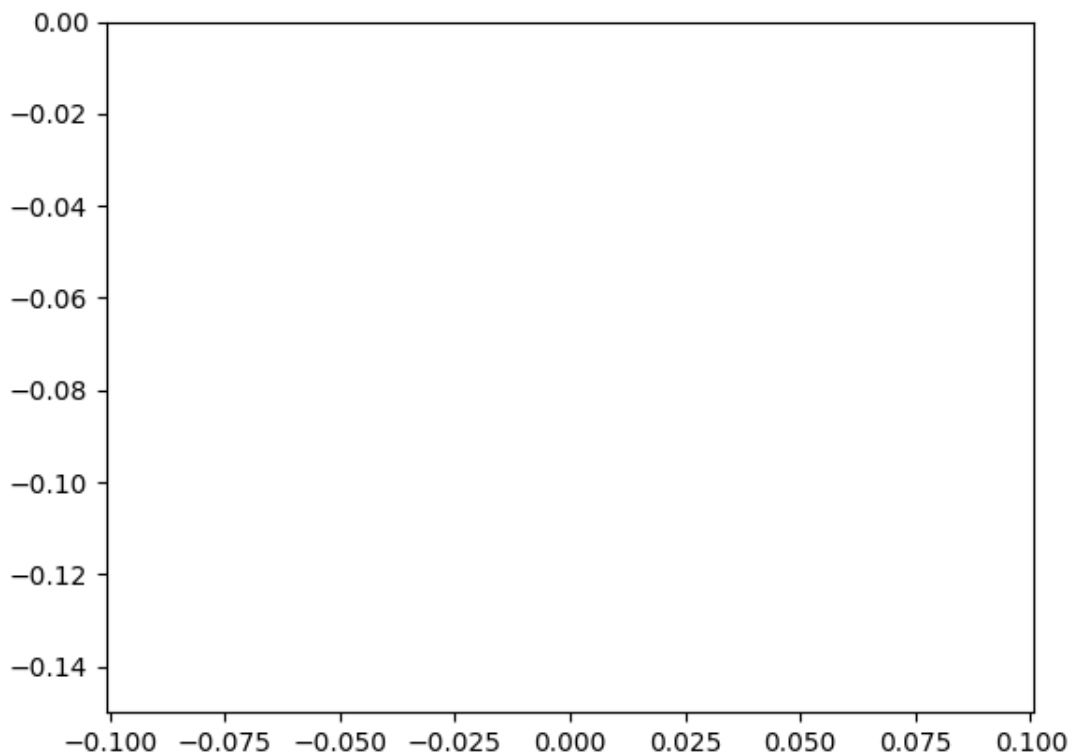
```
history_x.clear()
history_y.clear()

history_x.appendleft(p[0][2])
history_y.appendleft(p[1][2])

line.set_data(p[0], p[1])
trace.set_data(history_x, history_y)
step_text.set_text(step_template % (i))
#step_text.set_text("%s" % (p[0]))
return line, trace, step_text

ani = animation.FuncAnimation(fig, animate, len(leg_servo_positions), interval=200,
                              blit=True)

plt.show()
```



3.4 Deployment

We can deploy leg trajectories to a simulated or physical minipupper by writing angles to a file for each servo.

The controller in the simulated or physical environments reads all servo command files and finds the file with the most steps in it. This is the cycle length. You can specify the duration of one cycle by a parameter.

```
# write to disk

fh_hip = open("hip_servo", "w")
fh_upper_leg = open("upper_leg_servo", "w")
fh_lower_leg = open("lower_leg_servo", "w")
# hip will not move
fh_hip.write("%.10f\n" % 0.0)
for j in range(2*N-2):
    fh_upper_leg.write("%.10f\n" % leg_servo_positions[j][0])
    fh_lower_leg.write("%.10f\n" % leg_servo_positions[j][1])
fh_hip.close()
fh_upper_leg.close()
fh_lower_leg.close()

# Check

cycle_duration = 1 # duration in seconds

hip = []
with open("hip_servo", "r") as f:
    for line in f:
        hip.append(float(line.rstrip()))
upper_leg = []
with open("upper_leg_servo", "r") as f:
    for line in f:
        upper_leg.append(float(line.rstrip()))
lower_leg = []
with open("lower_leg_servo", "r") as f:
    for line in f:
        lower_leg.append(float(line.rstrip()))

max_steps = max(len(hip), len(upper_leg), len(lower_leg))

print("Duration per cycle: %s [msec]\nDuration per step: %s [msec]\nNumber of steps:
↳ %s"
      % (cycle_duration*1000, cycle_duration*1000/max_steps, max_steps))
```

```
Duration per cycle: 1000 [msec]
Duration per step: 26.31578947368421 [msec]
Number of steps: 38
```

```
# copy files to controller
import shutil

joints = ['hip', 'upper_leg', 'lower_leg']
legs = ['lf', 'rf', 'lh', 'rh']

for joint in joints:
```

(continues on next page)

(continued from previous page)

```
for leg in legs:
    shutil.copy("./%s_servo" % joint, "../controller/servos/%s_%s" % (leg, joint))
```

```
# example how to control only front legs
```

```
for joint in joints:
    for leg in legs[2:]:
        with open("../controller/servos/%s_%s" % (leg, joint), "w") as f:
            angle = 0.0
            if 'upper' in joint:
                angle = -np.pi/4
            if 'lower' in joint:
                angle = np.pi/2
            f.write("%.10f\n" % angle)
```

```
# Save foot movements
```

```
for joint in joints:
    for leg in legs:
        shutil.copy("./%s_servo" % joint, "../controller/servos/pushup/%s_%s" % (leg,
↵joint))
```


MINIPUPPER LEG MOVEMENTS

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

```
# Just a helper module, details are covered later
from gaits import GaitController
```

4.1 Minipupper leg and servos

We made the assumption that the lower leg joint moves around the endpoint of the upper leg and this is how we calculate the angles.

However the lower leg of the minipupper moves as in the figure below. The lower leg servo is connected to the hip cage and drives the lower leg through linkage.

```
upper_leg_length = 0.05022511821787979 # from urdf
lower_leg_length = 0.065 # measured, includes the rubber foot
lower_leg_extension_length = 0.015 # measured
servo_arm_length = 0.0125 # measured
servo_link_length = 0.0175 # measured
servo_delta_x = 0.0105 # measured
servo_delta_y = 0.014 # measured
gc = GaitController(1, 1, 1, 1, 1, 1)
```

```
theta = -3*np.pi/4
gamma = np.pi/2

# see below
zmax = -2.2142974355881813
#gamma = zmax - theta-1e-10 # correct rounding errors
#theta = np.round(zmax - gamma, 15)+1e-10 # correct rounding errors

leg = gc.mpForwardKin([theta, gamma])
servos = np.zeros((2,2))
servos[:, 0] = leg[:, 0]
servos[0, 1] = servos[0, 1] + servo_delta_x
servos[1, 1] = servos[1, 1] + servo_delta_y

# Black magic just for the figure
```

(continues on next page)

(continued from previous page)

```

linkage = np.zeros((2,6))
T1 = gc._transformationMatrix(theta, gc.lu)
T2 = gc._transformationMatrix(gamma+np.pi, lower_leg_extension_length)
T3 = T1.dot(T2)
P = T3.dot(np.array([0, 0, 1]))[0:2]
linkage[:, 0] = leg[:, 1]
linkage[:, 1] = P
T1 = gc._transformationMatrix(theta, 0)
T2 = gc._transformationMatrix(gamma+np.pi, lower_leg_extension_length)
T3 = T1.dot(T2)
P = T3.dot(np.array([0, 0, 1]))[0:2]
linkage[:, 2] = P
T2 = gc._transformationMatrix(gamma, servo_arm_length)
T3 = T1.dot(T2)
P = T3.dot(np.array([0, 0, 1]))[0:2]
linkage[:, 3] = P
linkage[:, 5] = servos[:, 1]

```

```

# copied from http://paulbourke.net/geometry/circlesphere/circle_intersection.py
# adapted to numpy

```

```

class Circle(object):
    """ An OOP implementation of a circle as an object """

    def __init__(self, point, radius):
        self.xpos = point[0]
        self.ypos = point[1]
        self.radius = radius

    def circle_intersect(self, circle2):
        """
        Intersection points of two circles using the construction of triangles
        as proposed by Paul Bourke, 1997.
        http://paulbourke.net/geometry/circlesphere/
        """
        X1, Y1 = self.xpos, self.ypos
        X2, Y2 = circle2.xpos, circle2.ypos
        R1, R2 = self.radius, circle2.radius

        Dx = X2-X1
        Dy = Y2-Y1
        D = np.sqrt(Dx**2 + Dy**2)
        # Distance between circle centres
        if D > R1 + R2:
            return "The circles do not intersect"
        elif D < np.fabs(R2 - R1):
            return "No Intersect - One circle is contained within the other"
        elif D == 0 and R1 == R2:
            return "No Intersect - The circles are equal and coincident"
        else:
            if D == R1 + R2 or D == R1 - R2:
                CASE = "The circles intersect at a single point"
            else:
                CASE = "The circles intersect at two points"
            chorddistance = (R1**2 - R2**2 + D**2)/(2*D)

```

(continues on next page)

(continued from previous page)

```

# distance from 1st circle's centre to the chord between intersects
halfchordlength = np.sqrt(np.round(R1**2 - chorddistance**2, 15))
chordmidpointx = X1 + (chorddistance*Dx)/D
chordmidpointy = Y1 + (chorddistance*Dy)/D
I1 = (chordmidpointx + (halfchordlength*Dy)/D,
      chordmidpointy - (halfchordlength*Dx)/D)
theta1 = np.degrees(np.arctan2(I1[1]-Y1, I1[0]-X1))
I2 = (chordmidpointx - (halfchordlength*Dy)/D,
      chordmidpointy + (halfchordlength*Dx)/D)
theta2 = np.degrees(np.arctan2(I2[1]-Y1, I2[0]-X1))
if theta2 > theta1:
    I1, I2 = I2, I1
return (I1, I2, CASE)

```

```

c1 = Circle(linkage[:, 3], servo_link_length)
c2 = Circle(servos[:, 1], servo_arm_length)
ret = c1.circle_intersect(c2)
if len(ret) == 28:
    linkage[:, 5] = [0.0, 0.0]
    raise ValueError("Invalid angles: %s" % ret)
linkage[:, 4] = ret[1]

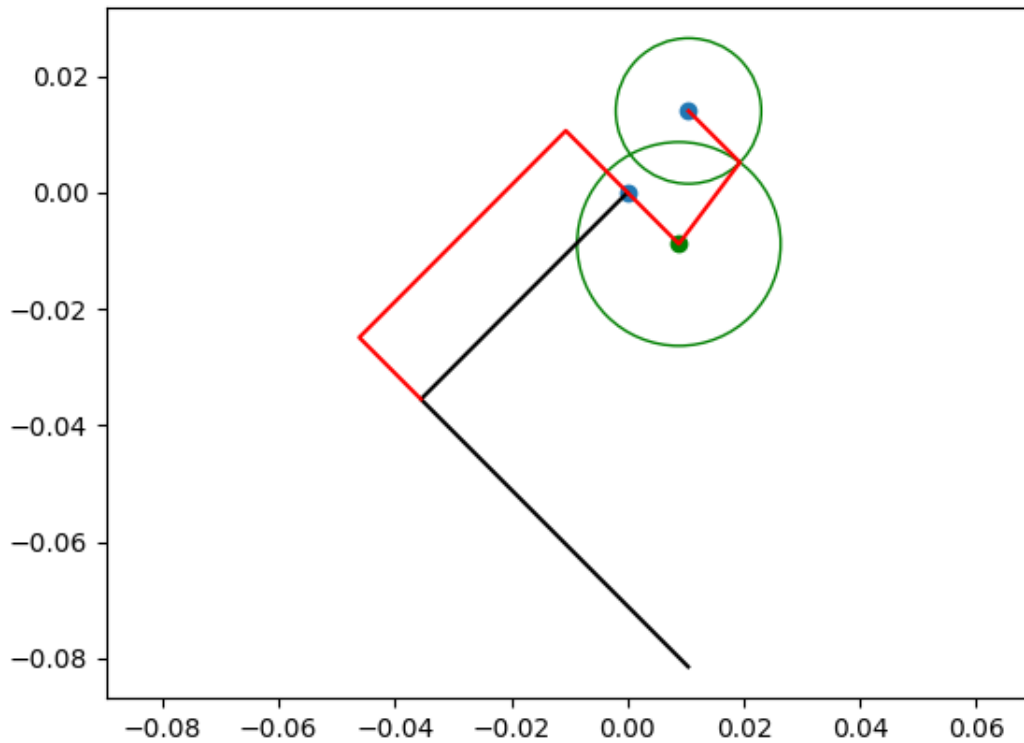
```

```

#plot
fig, ax = plt.subplots()
plt.plot(leg[0], leg[1], '-k')
plt.scatter(servos[0], servos[1])
plt.plot(linkage[0], linkage[1], '-r')
plt.scatter(linkage[0, -3], linkage[1, -3], color='green')
circle1 = plt.Circle(linkage[:, 3], servo_link_length, color='g', fill=False)
circle2 = plt.Circle(servos[:, 1], servo_arm_length, color='g', fill=False)
ax.add_patch(circle1)
ax.add_patch(circle2)
plt.axis('equal') #<-- set the axes to the same scale

plt.grid(visible=False, which='major')

```



```
# angle of servo arm (see below)
alpha = -np.arccos((linkage[0, 4] - servos[0, 1])/servo_arm_length)
print("%.5f" % (np.degrees(alpha), np.degrees(theta + gamma)))
```

```
-45.00000 -45.00000
```

4.2 Special case: Parallelogram

The above considerations are generic and by changing the length of the different part you can see the impact. At first I have taken a wrong measurement which lead me down this path.

However there is a special case where the length of servo_link is exactly the distance between the two servo motors. This is when the following condition is met:

$$servo_link_length^2 = servo_delta_x^2 + servo_delta_y^2$$

In this case we have a parallelogram between the lower leg servo and the linkage that revolves around the upper leg servo. There is a second parallelogram formed by the linkage spanning the part revolving around the upper leg servo and the lower leg. An element of that part is adjustable to ensure the property is met.

In this special case the lower leg servo arm is always parallel to the lower leg. We set $\zeta = \theta + \gamma$ as the angle of the lower leg servo arm. We can see that we have a dependency between the position of the upper and lower leg. Changing the position of the upper leg by maintaining the relative position of the lower leg requires also to adapt the lower leg. This is different from quadrupeds where the lower leg actuator is actually attached to the upper leg

4.3 Angle limits

The lower leg servo arm is limited in rotation. First the servo motor itself has a limited rotation of 180° . This is further limited by mechanics. We will have a collision if the lower leg servo arm comes too close to the upper leg servo. We will set a limit of -90° . We also do not want the end of the upper leg to touch ground before the lower leg can get to the ground. We therefore limit the lower leg servo arm to 0° . If the upper leg is set to -90° we can not move the lower leg less than 45° .

We further limit the upper leg movement to a range between -90° and -180° and the lower leg movement to a range between 0° and 135° .

In summary and $\zeta = \theta + \gamma$ as the angle of the lower leg servo arm:

$$-90^\circ \leq \zeta \leq 0^\circ$$

$$-180^\circ \leq \theta - 90^\circ$$

$$45^\circ \leq \gamma \leq 135^\circ$$

```
joints = ['hip', 'upper_leg', 'lower_leg']
legs = ['lf', 'rf', 'lh', 'rh']

angles = np.linspace(-np.pi/2, -np.pi, 20)

for joint in joints:
    for leg in legs:
        with open("../controller/servos/leg-test/%s_%s" % (leg, joint), "w") as f:
            angle = 0.0
            if 'upper' in joint:
                angle = -3*np.pi/4
            if 'lower' in joint:
                angle = np.pi/2
            f.write("%.10f\n" % angle)

with open("../controller/servos/leg-test/rf_upper_leg", "w") as f:
    for i in range(len(angles)):
        f.write("%.10f\n" % angles[i])
with open("../controller/servos/leg-test/rf_lower_leg", "w") as f:
    for i in range(len(angles)):
        if angles[i] > -3*np.pi/4:
            f.write("%.10f\n" % (np.pi/2))
        else:
            f.write("%.10f\n" % (3*np.pi/4))
```

```
for joint in joints:
    for leg in legs:
        with open("../controller/servos/lift-leg/%s_%s" % (leg, joint), "w") as f:
            angle = 0.0
            if 'upper' in joint:
                angle = -3*np.pi/4
            if 'lower' in joint:
                angle = np.pi/2
            f.write("%.10f\n" % angle)

with open("../controller/servos/lift-leg/rf_upper_leg", "w") as f:
```

(continues on next page)

(continued from previous page)

```
angle = -np.pi
f.write("%.10f\n" % angle)

with open("../controller/servos/lift-leg/rf_lower_leg", "w") as f:
    angle = 3*np.pi/4
    f.write("%.10f\n" % angle)
```

DISCUSSION OF A PAPER

We will study foot trajectories as described in Foot Trajectory Planning Method with Adjustable Parameters for Complex Environment

We will break down the foot trajectory into two phases, a swing phase and a stance phase. we will use 3rd-order Bezier curves to describe the curves of each phase and to smoothen the impact of the transition between the two phases we will use a Sigmoid function.

A Bézier curve is a mapping from $s \in [0, 1]$ to convex combinations of points P_0, P_1, \dots, P_n in some vector space:

$$B(s) = \sum_{i=0}^n \binom{n}{i} s^i (1-s)^{n-i} P_i$$

with $n = 3$:

$$B(s) = \sum_{i=0}^3 \binom{3}{i} s^i (1-s)^{3-i} P_i$$

$$B(s) = (1-s)^3 P_0 + 3s(1-s)^2 P_1 + 3s^2(1-s) P_2 + s^3 P_3$$

To find the maximum we have to solve the differential equation

$$\frac{dB}{ds} = 0$$

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

```
import sympy as sp
s, P0, P1, P2, P3 = sp.symbols('s P_0 P_1 P_2 P_3')
```

```
B = (1-s)**3*P0+3*s*(1-s)**2*P1+3*s**2*(1-s)*P2+s**3*P3
B
```

$$P_0(1-s)^3 + 3P_1s(1-s)^2 + 3P_2s^2 \cdot (1-s) + P_3s^3$$

```
H, L = sp.symbols('H L')
```

```
B1 = B.subs(P0, sp.Matrix([[ -L/2, 0]]).transpose())
B2 = B1.subs(P1, sp.Matrix([[ 0, H]]).transpose())
B3 = B2.subs(P2, sp.Matrix([[ L/2, H]]).transpose())
B4 = B3.subs(P3, sp.Matrix([[ L/2, 0]]).transpose())
B4
```

$$\begin{bmatrix} \frac{Ls^3}{2} + \frac{3Ls^2 \cdot (1-s)}{2} - \frac{L(1-s)^3}{2} \\ 3Hs^2 \cdot (1-s) + 3Hs(1-s)^2 \end{bmatrix}$$

```
sp.solve(sp.diff(B4[1], s), s)
```

```
[1/2]
```

```
B4.subs(s, 1/2)
```

$$\begin{bmatrix} 0.1875L \\ 0.75H \end{bmatrix}$$

5.1 Swing Phase

The below figure shows the trajectory of the swing phase. We use from numpy the bezier.curve module to do the calculations. Red dots in the figure are the support points.

```
import bezier

step_length = 1.0
point_height = 1.0

points = np.asarray([
    [-step_length/2, 0.0, step_length/2, step_length/2],
    [0.0, point_height, point_height, 0.0]
])
curve = bezier.Curve(points, degree=3)
```

```
import matplotlib.patches as patches

fig, ax = plt.subplots()

plt.scatter(points[0], points[1], color='red')
curve.plot(50, ax=ax)
# plot step_length
l_points = np.zeros((2, 2))
l_points[0,0] = -step_length/2
l_points[1,0] = step_length/2
#plt.plot(l_points[0], l_points[1], '-k')
p1 = patches.FancyArrowPatch(l_points[0], l_points[1], arrowstyle='<->', mutation_
    scale=20)
ax.add_artist(p1)
```

(continues on next page)

(continued from previous page)

```

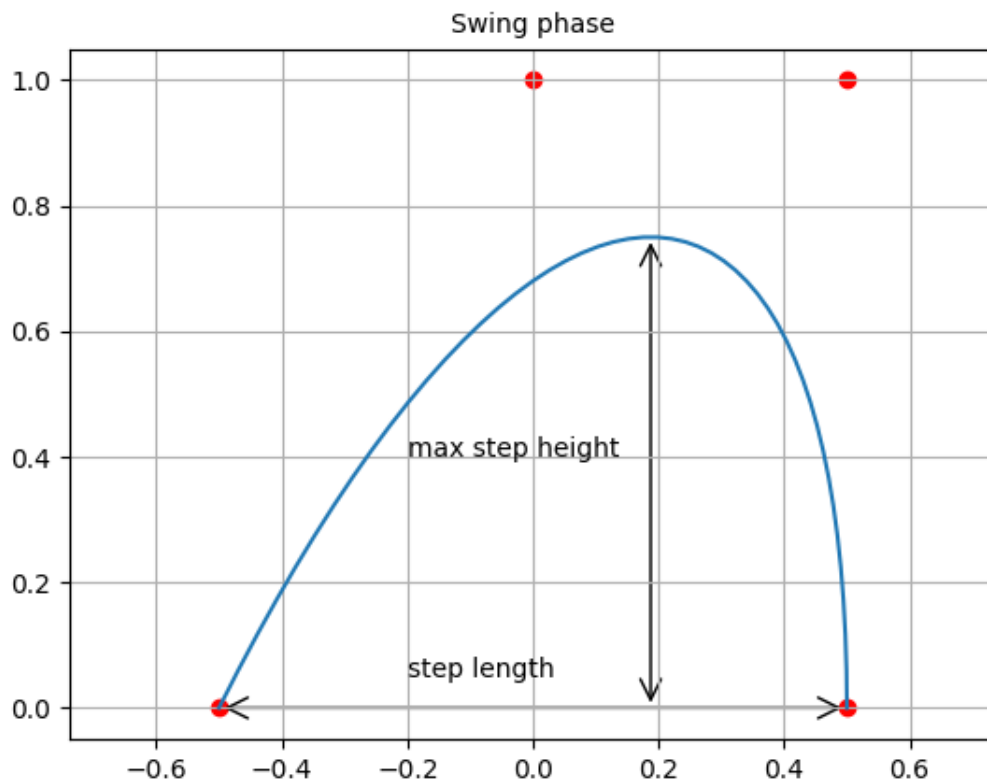
plt.text(-0.2, 0.05, "step length")
# plot step height
h_points = np.zeros((2, 2))
h_points[0,0] = 0.1875*step_length
h_points[1,0] = 0.1875*step_length
h_points[1,1] = 0.75*point_height
p2 = patches.FancyArrowPatch(h_points[0], h_points[1], arrowstyle='<->', mutation_
    scale=20)
ax.add_artist(p2)
plt.text(-0.2, 0.4, "max step height")

plt.axis('equal') #<-- set the axes to the same scale

plt.grid(visible=True, which='major')
plt.title('Swing phase', fontsize=10)

```

```
Text(0.5, 1.0, 'Swing phase')
```



5.2 Support Phase

We will use a straight line for the stance phase

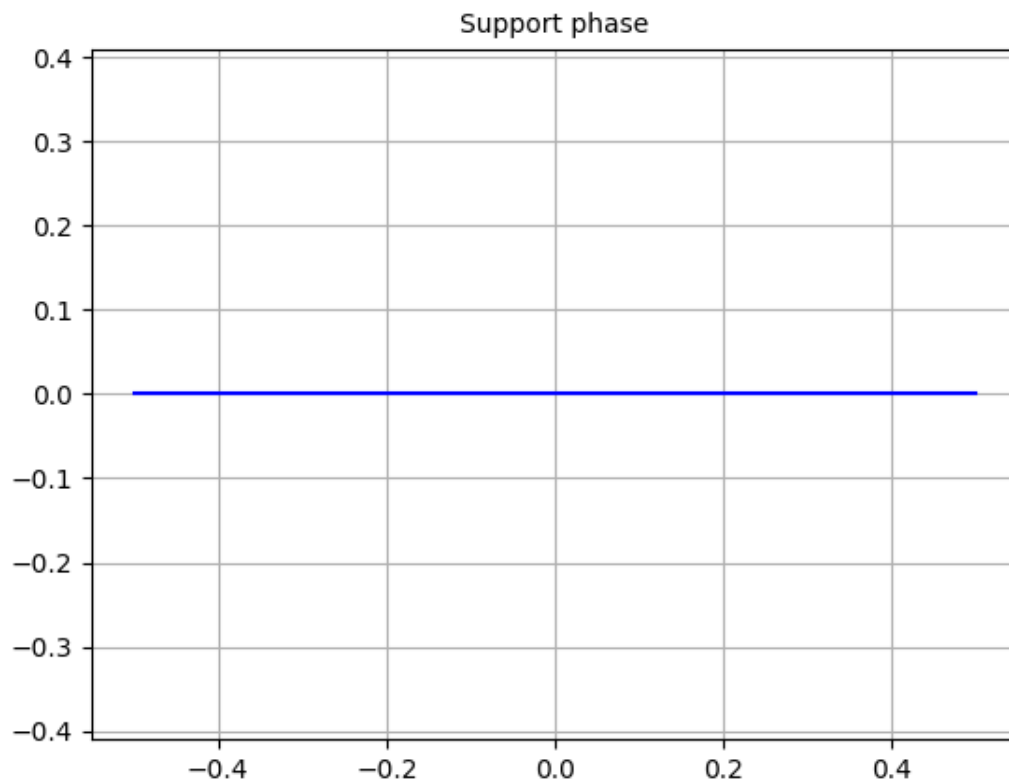
```
fig, ax = plt.subplots()

l_points = np.zeros((2, 2))
l_points[0,0] = -step_length/2
l_points[0,1] = step_length/2
plt.plot(l_points[0], l_points[1], '-b')

plt.axis('equal')  #<-- set the axes to the same scale

plt.grid(visible=True, which='major')
plt.title('Support phase', fontsize=10)
```

```
Text(0.5, 1.0, 'Support phase')
```



5.3 Sigmoid function

The sigmoid function is defined as

$$S(x) = \frac{1}{1 + e^{-x}}$$

We introduce a truncation parameter ϵ ($\epsilon > 0$) and map the time $t \in [0, 1]$ to the input $s \in [0, 1]$ of our Bézier curve

```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
```

```
num_points = 50
epsilon = 4
points = np.zeros((2, num_points))

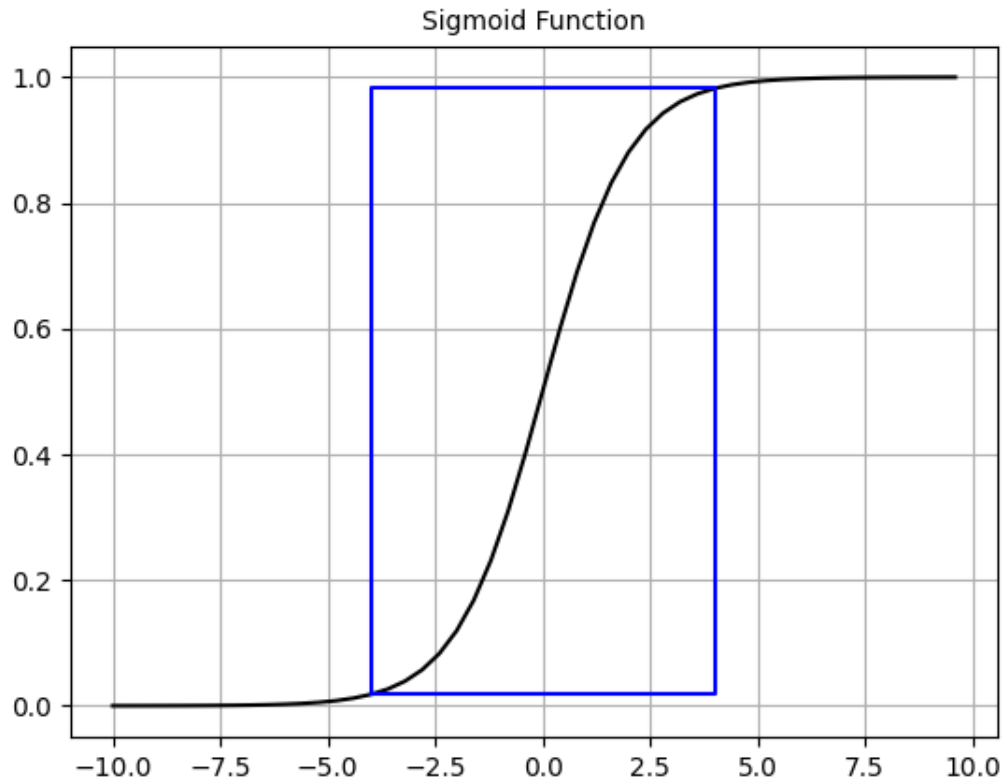
points[0] = np.linspace(-10, 10, num=num_points, endpoint=False)
points[1] = np.array(list(map(sigmoid, points[0])))

e_points = np.zeros((2, 5))
e_points[0,0] = -epsilon
e_points[1,0] = sigmoid(-epsilon)
e_points[0,1] = -epsilon
e_points[1,1] = sigmoid(epsilon)
e_points[0,2] = epsilon
e_points[1,2] = sigmoid(epsilon)
e_points[0,3] = epsilon
e_points[1,3] = sigmoid(-epsilon)
e_points[0,4] = e_points[0,0]
e_points[1,4] = e_points[1,0]

fig, ax = plt.subplots()
plt.plot(points[0], points[1], '-k')
plt.plot(e_points[0], e_points[1], '-b')

plt.grid(visible=True, which='major')
plt.title('Sigmoid Function', fontsize=10)
```

```
Text(0.5, 1.0, 'Sigmoid Function')
```

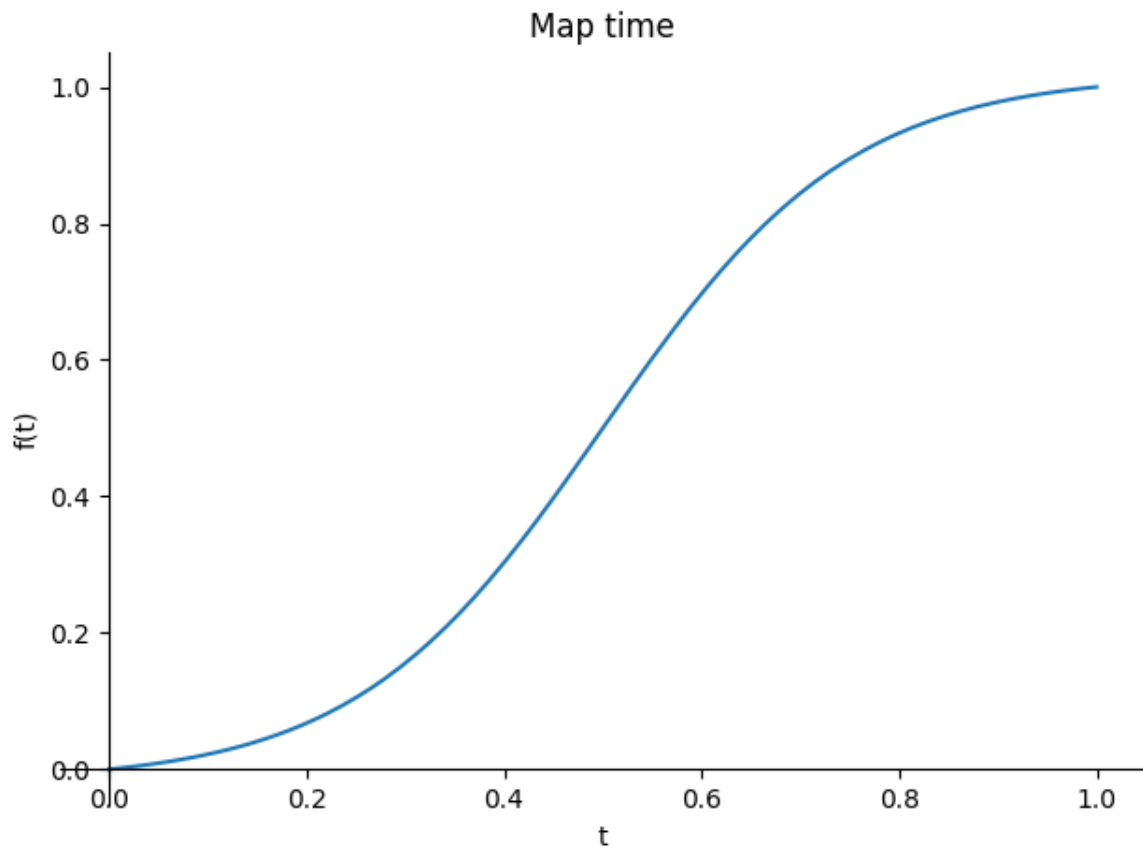


```
x, t, epsilon = sp.symbols('x t epsilon')
```

```
m = ((sp.exp(epsilon) + 1)/(sp.exp(epsilon) - 1))*(1/(sp.exp(-2*epsilon*(t-1/2))+1) - 1/(sp.exp(epsilon) + 1))
m
```

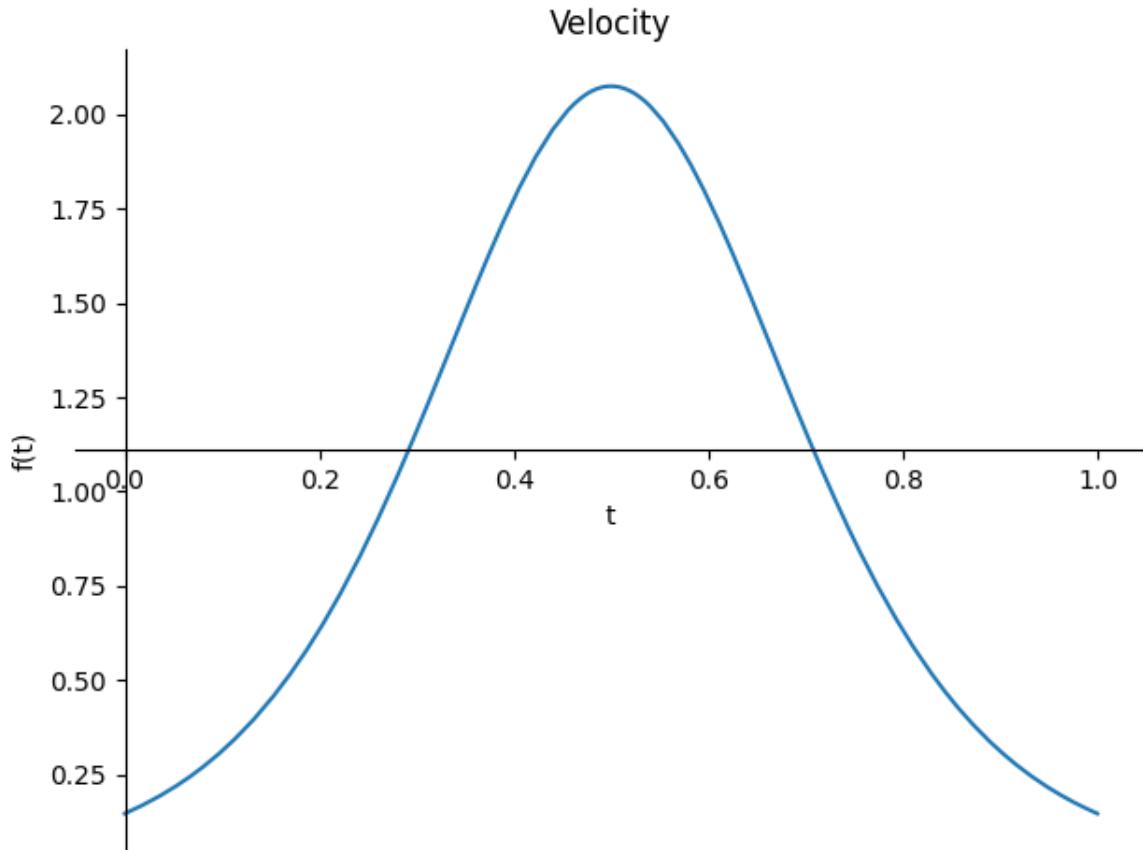
$$\frac{\left(-\frac{1}{e^{\epsilon}+1} + \frac{1}{1+e^{-2\epsilon(t-0.5)}}\right)(e^{\epsilon}+1)}{e^{\epsilon}-1}$$

```
from sympy.plotting import plot
plot(m.subs(epsilon, 4), (t, 0, 1), title='Map time')
```

```
<sympy.plotting.plot.Plot at 0x118d5af40>
```

```
plot(sp.diff(m.subs(epsilon, 4), t), (t, 0, 1), title='Velocity')
```



```
<sympy.plotting.plot.Plot at 0x118d4ca30>
```

5.4 Putting everything together

We start with the stance phase and select a number of points from the interval $[0, 1]$ using our sigmoid function. Then we select the same number of points for the swing phase but we do not include the endpoints since they are already included in the stance phase

```
def adapt_velocity(eps, times):
    return [(np.exp(eps) + 1)/(np.exp(eps) - 1) * (1/(np.exp(-2*eps*(t-1/2))+1) - 1/
    ↪ (np.exp(eps) + 1)) for t in times]
```

```
def stance_phase(l, eps, times):
    return l * np.array(adapt_velocity(eps, np.linspace(0, 1, num=N))) - 1/2
```

```
def swing_phase(l, h, eps, times):
    points = np.asarray([
        [-1/2, 0.0, 1/2, 1/2],
        [0.0, h/0.75, h/0.75, 0.0]
    ])
    curve = bezier.Curve(points, degree=3)
    return curve.evaluate_multi(times)
```

```

N = 20 # number of points
step_length = 1.0
step_height = 0.75
eps = 4

v_adj = np.array(adapt_velocity(4, np.linspace(0, 1, num=N)))

l_points = np.zeros((2, N))
l_points[0] = step_length * v_adj - step_length/2

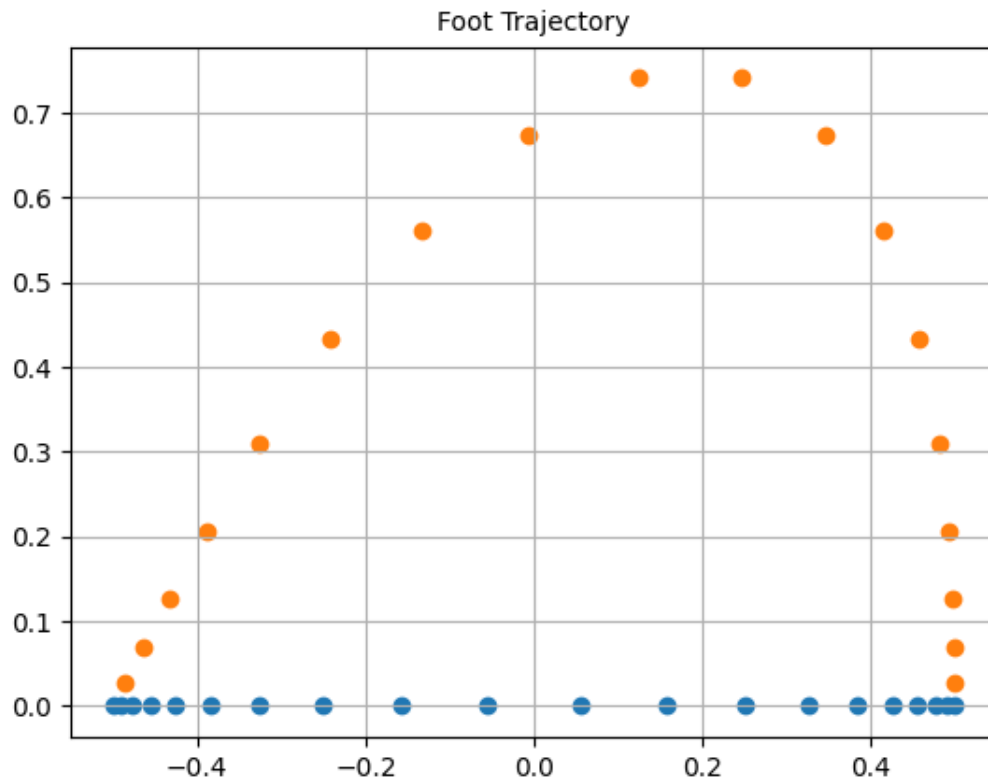
# remove first and last element from v_adj because it is already covered by the
# stance phase
s_points = np.array(swing_phase(step_length, step_height, eps, v_adj[1:-1]))

fig, ax = plt.subplots()
plt.scatter(l_points[0], l_points[1])
plt.scatter(s_points[0], s_points[1])

plt.grid(visible=True, which='major')
plt.title('Foot Trajectory', fontsize=10)

```

```
Text(0.5, 1.0, 'Foot Trajectory')
```



GAITS FOR MINIPUPPER

We moved the calculation of the foot trajectory to a Python class

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

```
from gaits import GaitController
```

```
# set values for minipupper
step_length = 0.05
step_height = 0.03
velocity_adapt = 4
number_of_points = 20
theta = -3*np.pi/4 # upper leg angle for origin of bezier curve
gamma = np.pi/2
```

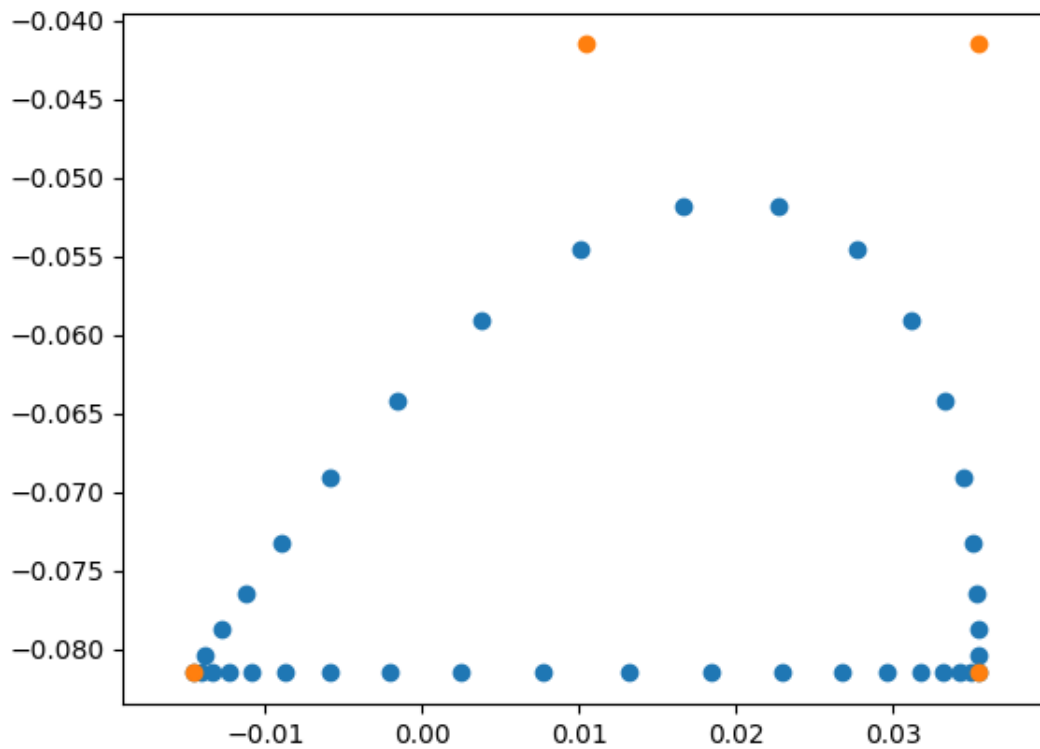
```
gc = GaitController(step_length, step_height, velocity_adapt, number_of_points, theta,
    ↪ gamma)
trajectory = gc.get_trajectory()
```

```
gc.v_adj
```

```
array([0.          , 0.00950315, 0.02364571, 0.04444672, 0.07451808,
        0.11692519, 0.17468076, 0.24971679, 0.34144914, 0.44560524,
        0.55439476, 0.65855086, 0.75028321, 0.82531924, 0.88307481,
        0.92548192, 0.95555328, 0.97635429, 0.99049685, 1.          ])
```

```
fig, ax = plt.subplots()
plt.xlim(-0.15, 0.15)
plt.ylim(-0.15, 0)
plt.axis('equal')
plt.grid(visible=False)
plt.scatter(trajectory[0], trajectory[1])
bezier_points = gc.get_bezier_points()
plt.scatter(bezier_points[0], bezier_points[1])
```

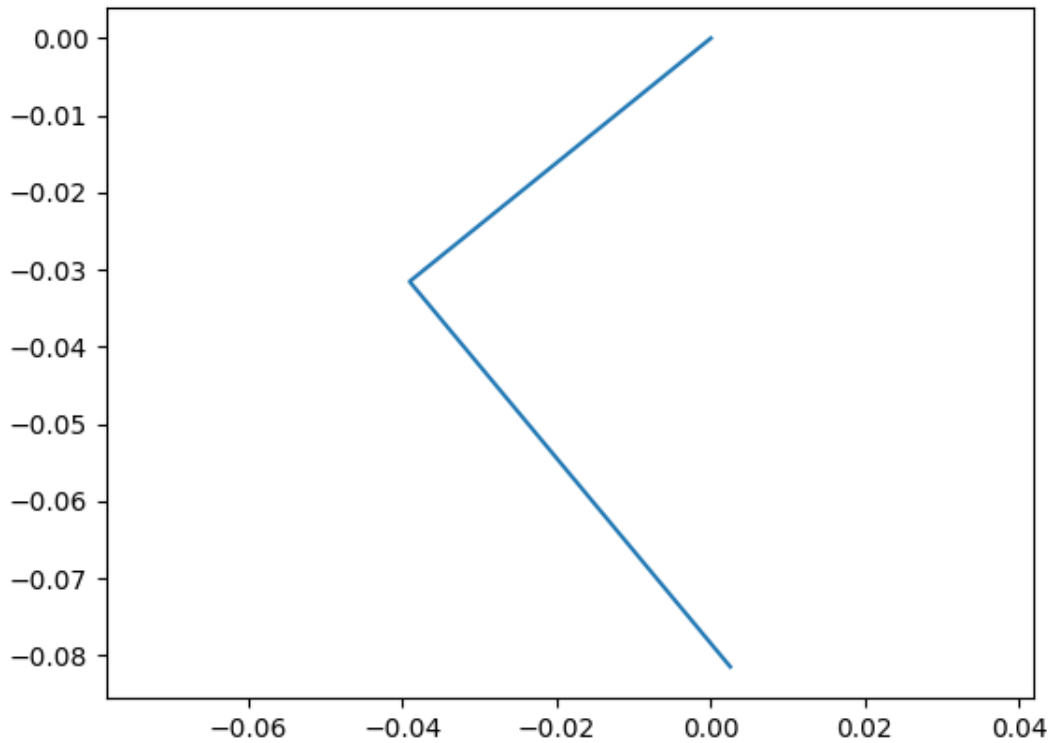
```
<matplotlib.collections.PathCollection at 0x11745d1f0>
```



```
leg_servo_positions = gc.get_leg_servo_positions()
```

```
p = gc.mpForwardKin(leg_servo_positions[11])
fig, ax = plt.subplots()
plt.xlim(-0.15, 0.15)
plt.ylim(-0.15, 0)
plt.axis('equal')
plt.grid(visible=False)
plt.plot(p[0], p[1])
```

```
[<matplotlib.lines.Line2D at 0x117551d30>]
```



```
import matplotlib.animation as animation
from collections import deque

fig, ax = plt.subplots()
plt.xlim(-0.15, 0.15)
plt.ylim(-0.15, 0)
plt.axis('equal')
plt.grid(visible=False)

line, = ax.plot([], [], 'o-', lw=2)
trace, = ax.plot([], [], '.-', lw=1, ms=2)
step_template = 'step = %.1f'
step_text = ax.text(0.05, 0.9, '', transform=ax.transAxes)
history_len = 50
history_x, history_y = deque(maxlen=history_len), deque(maxlen=history_len)

def animate(i):
    p = gc.mpForwardKin(leg_servo_positions[i])

    if i == 0:
        history_x.clear()
        history_y.clear()

    history_x.appendleft(p[0][2])
    history_y.appendleft(p[1][2])
```

(continues on next page)

(continued from previous page)

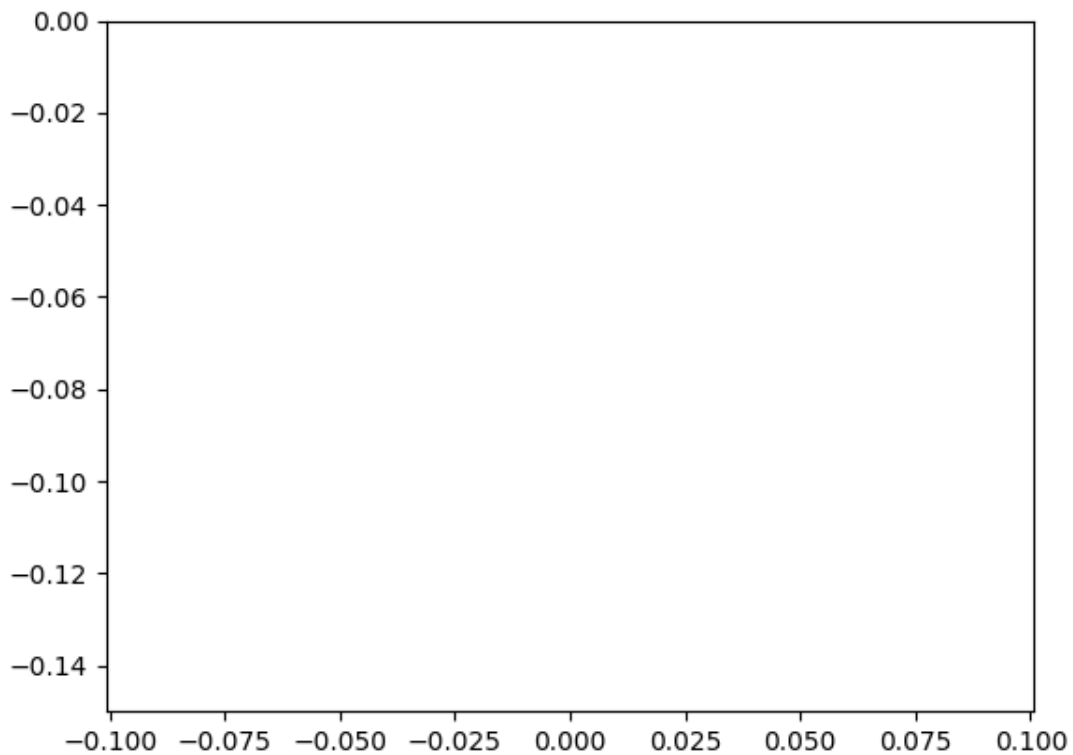
```

line.set_data(p[0], p[1])
trace.set_data(history_x, history_y)
step_text.set_text(step_template % (i))
#step_text.set_text("%s" % (p[0]))
return line, trace, step_text

ani = animation.FuncAnimation(fig, animate, len(leg_servo_positions), interval=200,
                               blit=True)

plt.show()

```



```

joints = ['hip', 'upper_leg', 'lower_leg']
legs = ['lf', 'rf', 'lh', 'rh']

for joint in joints:
    for leg in legs:
        fn = "%s_%s" % (leg, joint)
        if 'lf' in fn or 'rh' in fn:
            lp = leg_servo_positions
        else:
            lp = np.roll(leg_servo_positions, number_of_points, axis=0)
        with open("../controller/servos/walk/%s" % fn, "w") as f:

```

(continues on next page)

(continued from previous page)

```

if 'hip' in fn:
    f.write("%.10f\n" % 0.0)
elif 'upper' in fn:
    for j in range(len(lp)):
        f.write("%.10f\n" % lp[j][0])
else:
    for j in range(len(lp)):
        f.write("%.10f\n" % lp[j][1])

```

6.1 Leg position for testing

```

# neutral position
lp = [[-3*np.pi/4, np.pi/2]]

for joint in joints:
    for leg in legs:
        fn = "%s_%s" % (leg, joint)
        with open("../controller/servos/neutral/%s" % fn, "w") as f:
            if 'hip' in fn:
                f.write("%.10f\n" % 0.0)
            elif 'upper' in fn:
                for j in range(len(lp)):
                    f.write("%.10f\n" % lp[j][0])
            else:
                for j in range(len(lp)):
                    f.write("%.10f\n" % lp[j][1])

```

```

# streched position (max allowed on minipupper)
lp = [[-np.pi/2, np.pi/4]]

for joint in joints:
    for leg in legs:
        fn = "%s_%s" % (leg, joint)
        with open("../controller/servos/streched_legs/%s" % fn, "w") as f:
            if 'hip' in fn:
                f.write("%.10f\n" % 0.0)
            elif 'upper' in fn:
                for j in range(len(lp)):
                    f.write("%.10f\n" % lp[j][0])
            else:
                for j in range(len(lp)):
                    f.write("%.10f\n" % lp[j][1])

```

```

# sitting position (max allowed on minipupper)
lp = [[-np.pi/2, np.pi/2]]

for joint in joints:
    for leg in legs:
        fn = "%s_%s" % (leg, joint)
        with open("../controller/servos/sitting/%s" % fn, "w") as f:
            if 'hip' in fn:

```

(continues on next page)

(continued from previous page)

```
f.write("%.10f\n" % 0.0)
elif 'upper' in fn:
    for j in range(len(lp)):
        f.write("%.10f\n" % lp[j][0])
else:
    for j in range(len(lp)):
        f.write("%.10f\n" % lp[j][1])

# retracted position (max allowed on minipupper)
lp = [[-np.pi, 3*np.pi/4]]

for joint in joints:
    for leg in legs:
        fn = "%s_%s" % (leg, joint)
        with open("../controller/servos/retracted_legs/%s" % fn, "w") as f:
            if 'hip' in fn:
                f.write("%.10f\n" % 0.0)
            elif 'upper' in fn:
                for j in range(len(lp)):
                    f.write("%.10f\n" % lp[j][0])
            else:
                for j in range(len(lp)):
                    f.write("%.10f\n" % lp[j][1])
```

3D COORDINATE SYSTEMS

7.1 Conventions

In 2D coordinate system it is fairly standard to for +x to point to the right and +y to point up. In 3D coordinate systems we need a 3rd axis that is orthogonal to the other two but it is not standardized how to orient these axis. We will use a right-handed coordinate system. The name derives from the right-hand rule. If the index finger of the right hand is pointed forward, the middle finger bent inward at a right angle to it, and the thumb placed at a right angle to both, the three fingers indicate the relative orientation of the x-, y-, and z-axes in a right-handed system Wikipedia

In plots we will color the x-axis in red, the y-axis in green and the z-axis in blue.

7.2 Rotation in 3D

7.2.1 Euler angles

The Euler angles are three angles to describe the orientation of a rigid body with respect to a fixed coordinate system.

A general rotation of an object on 3D space is broken down to three rotations around the axis of a coordinate system.

Rotation about z axis:

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

Rotation about x axis:

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

Rotation about y axis:

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

7.2.2 Gimbal lock

There is a corner case when using Euler angles that leads to a gimbal lock

7.2.3 Quaternion

To avoid the issue of gimbal lock we can also use quaternions to describe a rotation in 3D space.

Quaternions are generally represented in the form

$$a + b\mathbf{i} + c\mathbf{j} + d\mathbf{k}$$

where a , b , c , and d are real numbers and \mathbf{i} , \mathbf{j} , and \mathbf{k} are the basic quaternions. The following properties hold:

$$\mathbf{i}^2 = \mathbf{j}^2 = \mathbf{k}^2 = \mathbf{ijk} = -1$$

7.2.4 Axis-angle

With the Axis-angle representation we describe a rotation in 3D space by a unit vector \mathbf{e} indicating the direction of an axis of rotation, and an angle θ describing the magnitude of the rotation about the axis

7.2.5 Rotation matrix

The rotation matrix R specifies the rotation.

With Euler angles we have the rotation matrix:

$$R(\theta_x, \theta_y, \theta_z) = R_z(\theta_z) R_y(\theta_y) R_x(\theta_x)$$

$$R(\theta_x, \theta_y, \theta_z) = \begin{bmatrix} c_y c_z & s_x s_y c_z - c_x s_z & c_x s_y s_z - s_x c_z & 0 \\ c_y s_z & s_x s_y s_z + c_x c_z & c_x s_y c_z - s_x s_z & 0 \\ -s_y & s_x c_y & c_x c_y & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$c_i = \cos(\theta_i)$$

$$s_i = \sin(\theta_i)$$

7.2.6 TODO

rotation matrix with quaternions

rotation matrix with axis-angle

7.2.7 Active or passive rotation

This part is still TODO

7.3 Transformations in 3D

7.3.1 Position and quaternion

7.3.2 Dual quaternion

7.3.3 Transformation matrix

7.3.4 Graph of Transformations

7.4 Pose

This part is still TODO

7.5 Python transformation libraries

In ROS 1, the TF library provided the helpful `transformations.py` module for doing various rotation-based conversions.

However, ROS 2 only supports TF2

Quoting from TF (Tully Foote) himself on ROS Answers,

`tf.transformations` is a fork of `transformations`. This package has been deprecated “`Transformations.py` is no longer actively developed and has a few known issues and numerical instabilities.”

The recommended alternative is a package available via pip called `transforms3d`.

A more recent project is `pytransform3d`

The SciPy framework provides `scipy spatial`

We will use `pytransform3d` for our studies

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

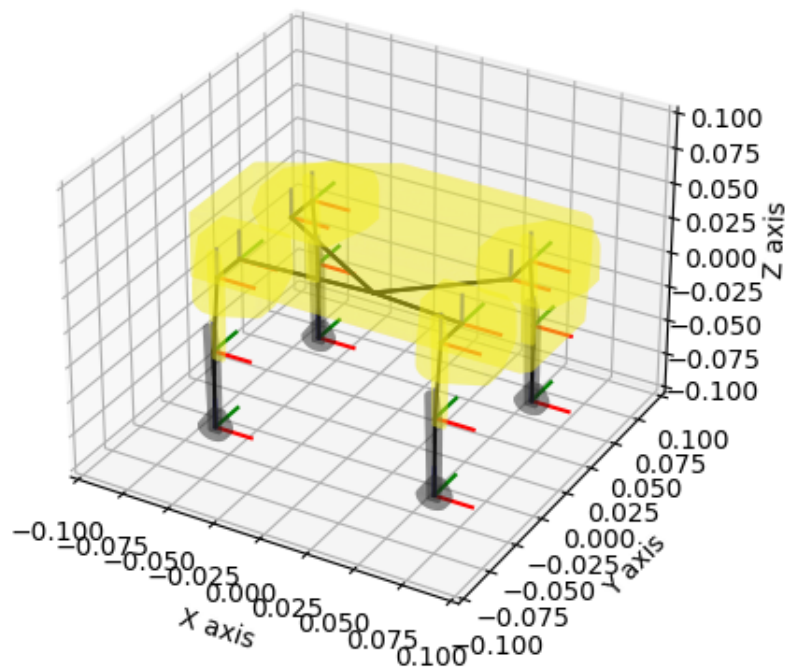
```
import os
from pytransform3d.urdf import UrdfTransformManager
```

```
tm = UrdfTransformManager()
with open('../mini-pupper_description/urdf/mini-pupper.urdf', 'r') as f:
    tm.load_urdf(f.read(), package_dir='../')
```

```
whitelist = []
legs = ['rf', 'lf', 'rh', 'lh']
for l in legs:
    whitelist.append("%s_hip_link" % l)
    whitelist.append("%s_upper_leg_link" % l)
    whitelist.append("%s_lower_leg_link" % l)
    whitelist.append("%s_foot_link" % l)
```

```
fig = plt.figure()
ax = fig.add_subplot(projection='3d')
for joint in tm._joints.keys():
    tm.set_joint(joint, 0.0)
    if "1_" in joint:
        tm.set_joint(joint, np.pi/4)
    if "2_" in joint:
        tm.set_joint(joint, -np.pi/2)
ax = tm.plot_frames_in('mini-pupper', whitelist=whitelist, s=0.02, show_name=False,
    ax=ax)
ax = tm.plot_connections_in("mini-pupper", ax=ax)
tm.plot_visuals("mini-pupper", ax=ax)
ax.set_xlim((-0.1, 0.1))
ax.set_ylim((-0.1, 0.1))
ax.set_zlim((-0.1, 0.1))
ax.set_xlabel('X axis')
ax.set_ylabel('Y axis')
ax.set_zlabel('Z axis')
```

```
Text(0.5, 0, 'Z axis')
```



```
tm.transforms[('rf_upper_leg_link', 'rf_hip_link')]
```

```
array([[ 1.    ,  0.    ,  0.    ,  0.    ],
       [ 0.    ,  1.    ,  0.    , -0.0197],
       [ 0.    ,  0.    ,  1.    ,  0.    ],
       [ 0.    ,  0.    ,  0.    ,  1.    ]])
```

```
tm.get_transform('rf_hip_link', 'mini-pupper')
```

```
array([[ 1.    ,  0.    ,  0.    ,  0.06014],
       [ 0.    ,  1.    ,  0.    , -0.0235 ],
       [ 0.    ,  0.    ,  1.    ,  0.0171 ],
       [ 0.    ,  0.    ,  0.    ,  1.    ]])
```

```
tm.get_transform('rf_foot_link', 'mini-pupper')
```

```
array([[ 1.    ,  0.    ,  0.    ,  0.06014],
       [ 0.    ,  1.    ,  0.    , -0.04795],
       [ 0.    ,  0.    ,  1.    , -0.0889 ],
       [ 0.    ,  0.    ,  0.    ,  1.    ]])
```

```
tm.transforms.keys()
```

```
dict_keys([('base_link', 'mini-pupper'), ('visual:base_link/0', 'base_link'), (
    'collision:base_link/0', 'base_link'), ('inertial_frame:base_inertia', 'base_
    inertia'), ('visual:lf_hip_link/0', 'lf_hip_link'), ('collision:lf_hip_link/0',
    'lf_hip_link'), ('inertial_frame:lf_hip_link', 'lf_hip_link'), ('visual:lf_upper_
    leg_link/0', 'lf_upper_leg_link'), ('collision:lf_upper_leg_link/0', 'lf_upper_
    leg_link'), ('inertial_frame:lf_upper_leg_link', 'lf_upper_leg_link'), (
    'visual:lf_lower_leg_link/0', 'lf_lower_leg_link'), ('collision:lf_lower_leg_
    link/0', 'lf_lower_leg_link'), ('inertial_frame:lf_lower_leg_link', 'lf_lower_
    leg_link'), ('visual:lf_foot_link/0', 'lf_foot_link'), ('collision:lf_foot_link/0
    ', 'lf_foot_link'), ('inertial_frame:lf_foot_link', 'lf_foot_link'), ('visual:lh_
    hip_link/0', 'lh_hip_link'), ('collision:lh_hip_link/0', 'lh_hip_link'), (
    'inertial_frame:lh_hip_link', 'lh_hip_link'), ('visual:lh_upper_leg_link/0', 'lh_
    upper_leg_link'), ('collision:lh_upper_leg_link/0', 'lh_upper_leg_link'), (
    'inertial_frame:lh_upper_leg_link', 'lh_upper_leg_link'), ('visual:lh_lower_leg_
    link/0', 'lh_lower_leg_link'), ('collision:lh_lower_leg_link/0', 'lh_lower_leg_
    link'), ('inertial_frame:lh_lower_leg_link', 'lh_lower_leg_link'), ('visual:lh_
    foot_link/0', 'lh_foot_link'), ('collision:lh_foot_link/0', 'lh_foot_link'), (
    'inertial_frame:lh_foot_link', 'lh_foot_link'), ('visual:rf_hip_link/0', 'rf_hip_
    link'), ('collision:rf_hip_link/0', 'rf_hip_link'), ('inertial_frame:rf_hip_link
    ', 'rf_hip_link'), ('visual:rf_upper_leg_link/0', 'rf_upper_leg_link'), (
    'collision:rf_upper_leg_link/0', 'rf_upper_leg_link'), ('inertial_frame:rf_upper_
    leg_link', 'rf_upper_leg_link'), ('visual:rf_lower_leg_link/0', 'rf_lower_leg_
    link'), ('collision:rf_lower_leg_link/0', 'rf_lower_leg_link'), ('inertial_
    frame:rf_lower_leg_link', 'rf_lower_leg_link'), ('visual:rf_foot_link/0', 'rf_
    foot_link'), ('collision:rf_foot_link/0', 'rf_foot_link'), ('inertial_frame:rf_
    foot_link', 'rf_foot_link'), ('visual:rh_hip_link/0', 'rh_hip_link'), (
    'collision:rh_hip_link/0', 'rh_hip_link'), ('inertial_frame:rh_hip_link', 'rh_
    hip_link'), ('visual:rh_upper_leg_link/0', 'rh_upper_leg_link'), ('collision:rh_
    upper_leg_link/0', 'rh_upper_leg_link'), ('inertial_frame:rh_upper_leg_link',
    'rh_upper_leg_link'), ('visual:rh_lower_leg_link/0', 'rh_lower_leg_link'), (
    'collision:rh_lower_leg_link/0', 'rh_lower_leg_link'), ('inertial_frame:rh_lower_
    leg_link', 'rh_lower_leg_link'), ('visual:rh_foot_link/0', 'rh_foot_link'), (
    'collision:rh_foot_link/0', 'rh_foot_link'), ('inertial_frame:rh_foot_link/0',
    'rh_foot_link'), ('base_inertia', 'base_link'), ('lf_hip_debug_link', 'base_
    link'), ('lf_hip_link', 'base_link'), ('lf_upper_leg_link', 'lf_hip_link'), ('lf_lower_
    leg_link', 'lf_hip_link'), ('lf_foot_link', 'lf_lower_leg_link'), ('lh_hip_
    debug_link', 'base_link'), ('lh_hip_link', 'base_link'), ('lh_upper_leg_link',
    'lh_hip_link'), ('lh_lower_leg_link', 'lh_upper_leg_link'), ('lh_foot_link', 'lh_
    lower_leg_link'), ('rf_hip_debug_link', 'base_link'), ('rf_hip_link', 'base_
    link'), ('rf_upper_leg_link', 'rf_hip_link'), ('rf_lower_leg_link', 'rf_upper_
    leg_link'), ('rf_foot_link', 'rf_lower_leg_link')])
```

7.5. Python transformation libraries

(continues on next page)

(continued from previous page)

```
tm.write_png('transformations_graph.png')
```


RIGID-BODY KINEMATICS

8.1 Reference

The following sources helped me to refine my understanding

Modern Robotics: Mechanics, Planning, and Control

8.2 Code

<https://github.com/NxRLab/ModernRobotics>

https://github.com/junhyeokahn/tf_rbd1

<https://github.com/stack-of-tasks/pinocchio>

8.3 Analytic Inverse Kinematics

To find an analytical solution to the inverse kinematics problem in 3D we decompose the problem into two steps, first find the orientation and then use the solution we developed for the 2D case.

This is specific to the kind of robots like the minipupper.

The hip joints revolve around the x axis and the leg joints revolve around the y axis. We need to find the plane that goes through the end of the hip joint the the projection of P on the y,z plane. The figure below illustrate the situation.

We can calculate the angel α as

$$\alpha = \arctan2(P_z, P_y)$$

and the angle β as

$$\beta = \arctan2(d, \sqrt{r^2 - d^2})$$

and finally the angle ϕ as

$$\phi = \frac{\pi}{2} + \alpha - \beta$$

To find the upper and lower leg angle we can no use the inverse kinematics in 2D as we have done previously

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

```
d = 1.3 # lenght of hip link
l = 4 # lenght of leg in projection
#d = 0.0197
#l = 0.07749896773506083
phi = np.radians(20)
points = np.zeros((2, 3))
points[0,1] = d*np.cos(phi)
points[1,1] = d*np.sin(phi)
points[0,2] = points[0,1] + l*np.cos(np.pi/2-phi)
points[1,2] = points[1,1] - l*np.sin(np.pi/2-phi)
points1 = np.zeros((2, 3))
points1[0,1] = l*np.cos(np.pi/2-phi)
points1[1,1] = -l*np.sin(np.pi/2-phi)
points1[0,2] = points1[0,1] + d*np.cos(phi)
points1[1,2] = points1[1,1] + d*np.sin(phi)
points2 = np.zeros((2, 2))
points2[0,1] = points[0,2]
points2[1,1] = points[1,2]
```

```
angles = np.linspace(0, float(phi))
xs = 0.5 * np.cos(angles)
ys = 0.5 * np.sin(angles)
```

```
alpha = np.arctan2(points[1,2], points[0,2])
r = np.sqrt(points[0,2]**2 + points[1,2]**2)
beta = np.arctan2(d, np.sqrt(r**2-d**2))
```

```
print("P = [%.6f, %.6f]" % (points[0,2], points[1,2]))
print("calculated value of phi: %.2f" % np.degrees(np.pi/2+alpha-beta))
```

```
P = [2.589681, -3.314144]
calculated value of phi: 20.00
```

```
angles1 = np.linspace(float(alpha), 0)
xs1 = 0.5 * np.cos(angles1)
ys1 = 0.5 * np.sin(angles1)
```

```
angles2 = np.linspace(phi-np.pi/2, float(alpha))
xs2 = 0.5 * np.cos(angles2)
ys2 = 0.5 * np.sin(angles2)
```

```
fig, ax = plt.subplots()
plt.xlim(-4,4) #<-- set the x axis limits
plt.ylim(-4,4) #<-- set the y axis limits
# plot hip link and leg
plt.plot(points[0], points[1], '-k')
plt.plot(points1[0], points1[1], '-r')
plt.plot(points2[0], points2[1], '--b')
# plot angle phi
plt.plot(xs, ys, '-b')
plt.text(xs[int(len(angles)/2)]+0.07, ys[int(len(angles)/2)], "$\\phi$ (%s$^{\circ}$)" % np.degrees(phi))
```

(continues on next page)

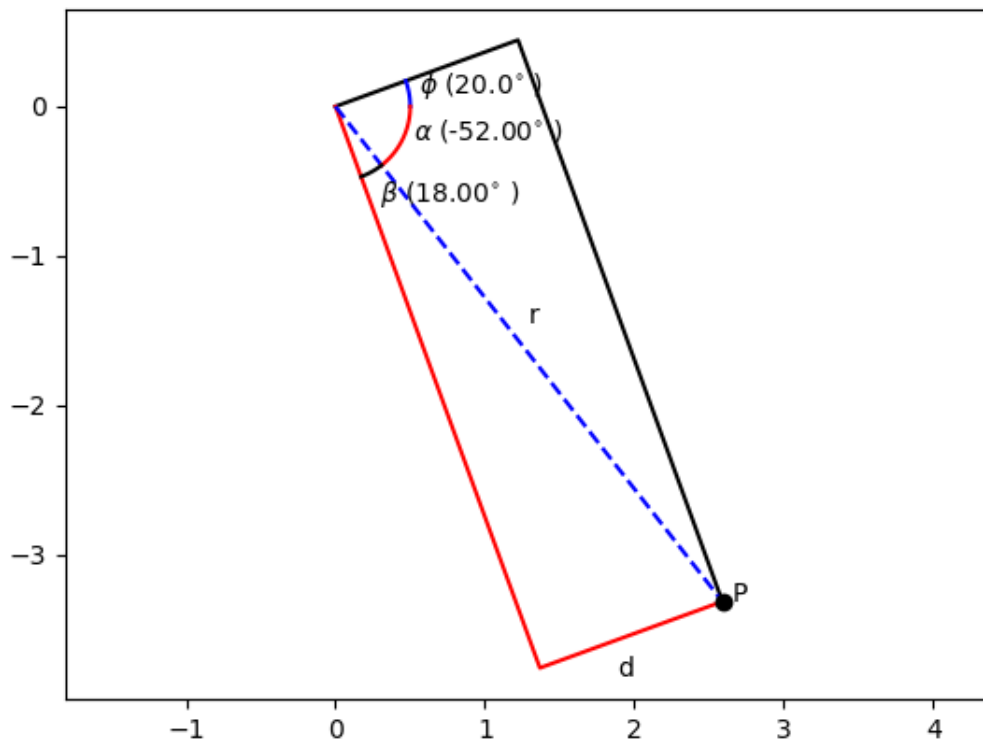
(continued from previous page)

```

# plot angle alpha
plt.plot(xs1, ys1, '-r')
plt.text(xs1[int(len(angles1)/2)]+0.07, ys1[int(len(angles1)/2)], "$\\alpha$ (%.2f$^{\circ}$)" % np.degrees(alpha))
# plot angle beta
plt.plot(xs2, ys2, '-k')
plt.text(xs2[int(len(angles2)/2)]+0.05, ys2[int(len(angles2)/2)]-0.2, "$\\beta$ (%.2f$^{\circ}$)" % np.degrees(beta))
plt.axis('equal')
# plot a black point at foot
plt.plot(points[0, 2], points[1, 2], 'ok')
plt.text(points[0, 2]+0.07, points[1, 2], "P")
plt.text(points[0, 2]-0.7, points[1, 2]-0.5, "d")
plt.text(points2[0, 1]/2, points2[1, 1]/2+0.2, "r")

plt.grid(visible=False, which='major')

```



Let's apply this to minipupper using the URDF file and the transformation manager from pytransform3d

```

import os
from pytransform3d.urdf import UrdfTransformManager

```

```

tm = UrdfTransformManager()

```

(continues on next page)

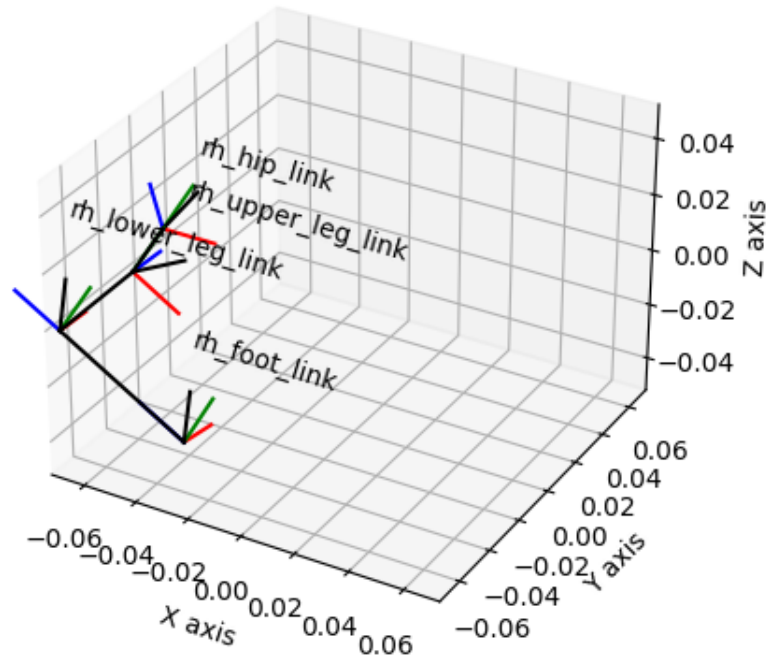
(continued from previous page)

```
with open('../mini-pupper_description/urdf/mini-pupper.urdf', "r") as f:
    tm.load_urdf(f.read(), package_dir='../')
```

```
whitelist = []
leg = 'rh'
whitelist.append("%s_hip_link" % leg)
whitelist.append("%s_upper_leg_link" % leg)
whitelist.append("%s_lower_leg_link" % leg)
whitelist.append("%s_foot_link" % leg)
```

```
fig = plt.figure()
ax = fig.add_subplot(projection='3d')
tm.set_joint('%s_hip_joint' % leg, np.radians(20))
tm.set_joint('%s_upper_leg_joint' % leg, np.pi/4)
tm.set_joint('%s_lower_leg_joint' % leg, -np.pi/2)
ax = tm.plot_frames_in('mini-pupper', whitelist=whitelist, s=0.02, show_name=True,
    ↪ax=ax)
ax = tm.plot_connections_in("mini-pupper", whitelist=whitelist, ax=ax)
ax.set_xlim((-0.07, 0.07))
ax.set_ylim((-0.07, 0.07))
ax.set_zlim((-0.05, 0.05))
ax.set_xlabel('X axis')
ax.set_ylabel('Y axis')
ax.set_zlabel('Z axis')
```

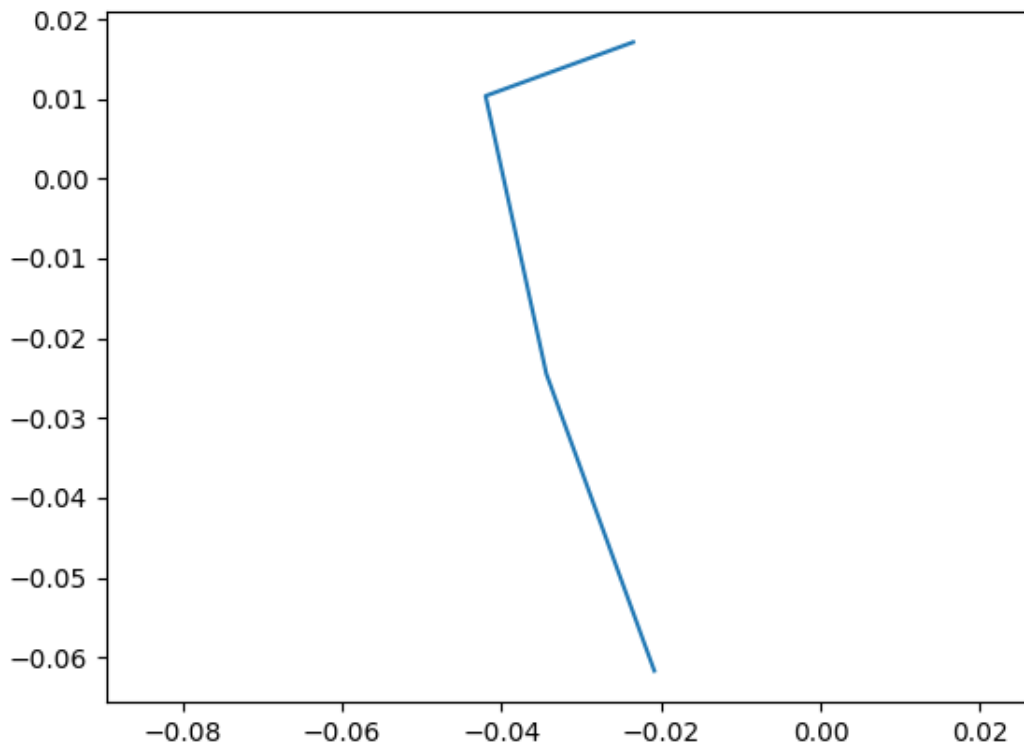
```
Text(0.5, 0, 'Z axis')
```



```
points = np.zeros((3, 4))
points[:,0] = tm.get_transform('%s_hip_link' % leg, 'base_link')[:3, 3:4].reshape(3)
points[:,1] = tm.get_transform('%s_upper_leg_link' % leg, 'base_link')[:3, 3:4].
    ↳reshape(3)
points[:,2] = tm.get_transform('%s_lower_leg_link' % leg, 'base_link')[:3, 3:4].
    ↳reshape(3)
points[:,3] = tm.get_transform('%s_foot_link' % leg, 'base_link')[:3, 3:4].reshape(3)
```

```
fig = plt.figure()
ax = fig.add_subplot()
ax.plot(points[1], points[2])
plt.axis('equal')

plt.grid(visible=False, which='major')
```



8.3.1 Observation

The upper and lower leg are not aligned in the plane as I would expect. Let's try to fix the urdf file

```
tm = UrdfTransformManager()
with open('../mini-pupper_description/urdf/mini-pupper_fixed.urdf', "r") as f:
    tm.load_urdf(f.read(), package_dir='../')
if 'l' in leg:
    orientation = 1
else:
    orientation = -1
tm.set_joint('%s_hip_joint' % leg, np.radians(orientation * 20))
tm.set_joint('%s_upper_leg_joint' % leg, np.pi/4)
tm.set_joint('%s_lower_leg_joint' % leg, -np.pi/4)
```

```
# points in base_link frame
points = np.zeros((3, 4))
points[:,0] = tm.get_transform('%s_hip_link' % leg, 'base_link')[:3, 3:4].reshape(3)
points[:,1] = tm.get_transform('%s_upper_leg_link' % leg, 'base_link')[:3, 3:4].
    ↳ reshape(3)
points[:,2] = tm.get_transform('%s_lower_leg_link' % leg, 'base_link')[:3, 3:4].
    ↳ reshape(3)
points[:,3] = tm.get_transform('%s_foot_link' % leg, 'base_link')[:3, 3:4].reshape(3)
```

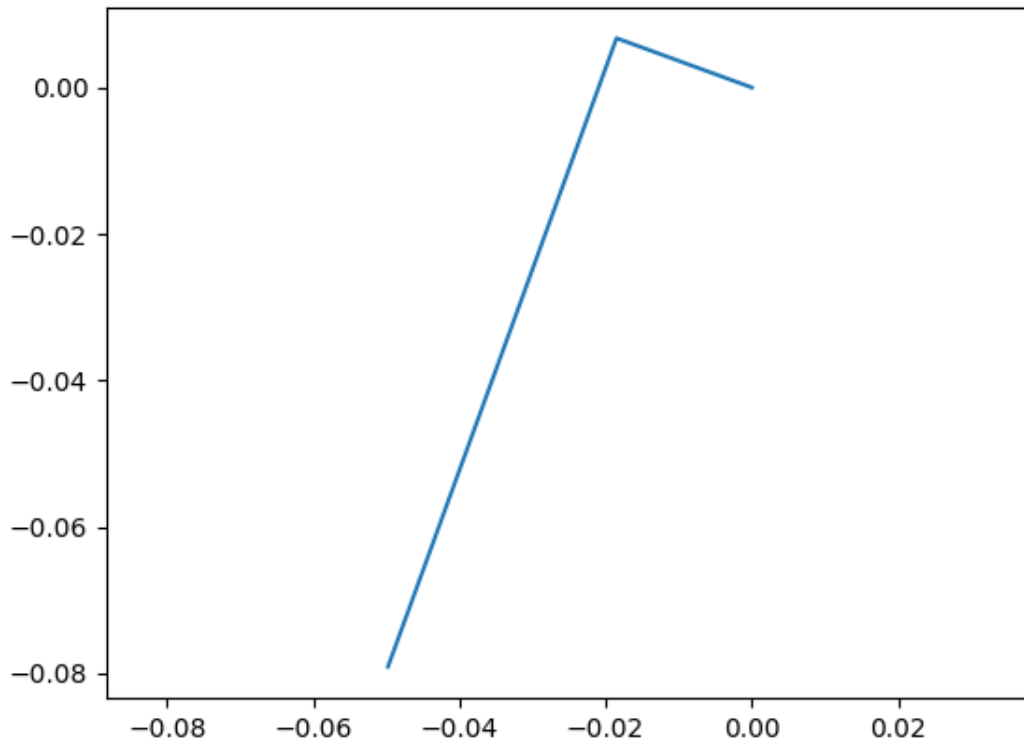
```
# points relative to hip
points1 = np.zeros((3, 4))
points1[:,0] = points[:,0] - points[:,0]
points1[:,1] = points[:,1] - points[:,0]
points1[:,2] = points[:,2] - points[:,0]
points1[:,3] = points[:,3] - points[:,0]
```

```
d = np.linalg.norm(points1[1:,1])
r = np.linalg.norm(points1[1:,3])
print(d)
print(r)
alpha = np.arctan2(points1[2,3], points1[1,3])
beta = np.arctan2(d, np.sqrt(r**2-d**2))
if 'r' in leg:
    beta = -beta
print(np.degrees(alpha))
print(np.degrees(beta))
print("calculated value of phi: %.2f" % np.degrees(np.pi/2+alpha-beta))
```

```
0.0197
0.093455272588788
-122.16900542679402
-12.169005426794019
calculated value of phi: -20.00
```

```
fig = plt.figure()
ax = fig.add_subplot()
ax.plot(points1[1], points1[2])
plt.axis('equal')

plt.grid(visible=False, which='major')
```



```
from gaits3D import GaitController3D
gc = GaitController3D()
```

```
angles = gc.mpInverseKin(leg, points[:,3])
```

```
np.degrees(angles[0])
```

```
-20.000000000000004
```

```
np.degrees(angles[1])
```

```
44.99999999941885
```

```
np.degrees(angles[2])
```

```
-44.99999999890996
```


8.4 Numerical Inverse Kinematics

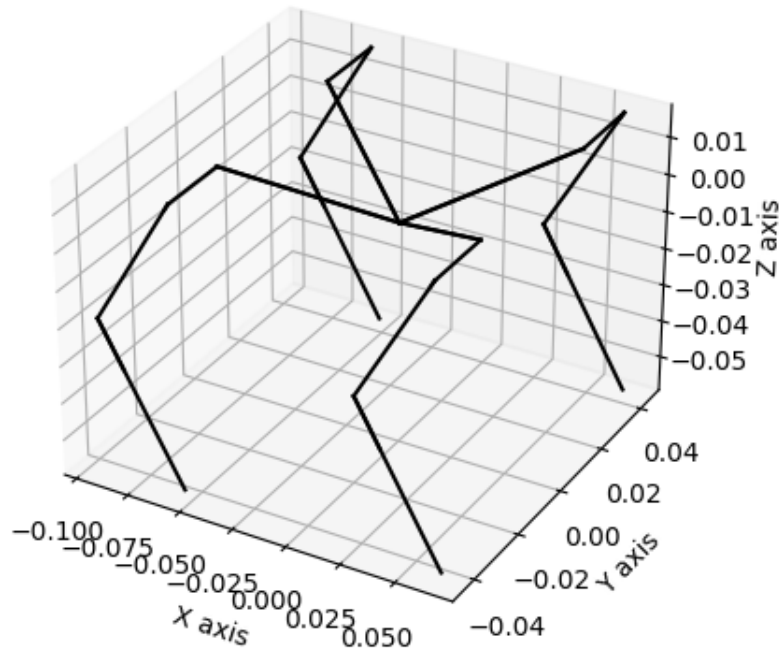
8.5 Robot Orientation

```
tm = UrdfTransformManager()
with open('../mini-pupper_description/urdf/mini-pupper_fixed.urdf', "r") as f:
    tm.load_urdf(f.read(), package_dir='../')
```

```
# Neutral position
legs = ['rf', 'lf', 'rh', 'lh']
servos = ['hip', 'upper_leg', 'lower_leg']
for leg in legs:
    for servo in servos:
        if 'hip' in servo:
            angle = 0.0
        if 'upper' in servo:
            angle = np.pi/4
        if 'lower' in servo:
            angle = -np.pi/2
    tm.set_joint('%s_%s_joint' % (leg, servo), angle)
```

```
fig = plt.figure()
ax = fig.add_subplot(projection='3d')
ax = tm.plot_connections_in("mini-pupper", ax=ax)
ax.set_xlabel('X axis')
ax.set_ylabel('Y axis')
ax.set_zlabel('Z axis')
```

```
Text(0.5, 0, 'Z axis')
```



```
wireframe = np.zeros((4, 3, 5))
legs = ['rf', 'lf', 'rh', 'lh']
for i in range(len(legs)):
    leg = legs[i]
    wireframe[i,:,1] = tm.get_transform('%s_hip_link' % leg, 'base_link')[:3, 3:4].
    ↪reshape(3)
    wireframe[i,:,2] = tm.get_transform('%s_upper_leg_link' % leg, 'base_link')[:3, ↪
    ↪3:4].reshape(3)
    wireframe[i,:,3] = tm.get_transform('%s_lower_leg_link' % leg, 'base_link')[:3, ↪
    ↪3:4].reshape(3)
    wireframe[i,:,4] = tm.get_transform('%s_foot_link' % leg, 'base_link')[:3, 3:4].
    ↪reshape(3)
```

```
# ref: https://stackoverflow.com/questions/6802577/rotation-of-3d-vector
# using the Euler-Rodrigues formula https://en.wikipedia.org/wiki/Euler-Rodrigues\_
↪formula
def rotation_matrix(axis, theta):
    """
    Return the rotation matrix associated with counterclockwise rotation about
    the given axis by theta radians.
    """
    axis = np.asarray(axis)
    axis = axis / np.sqrt(np.dot(axis, axis))
    a = np.cos(theta / 2.0)
    b, c, d = -axis * np.sin(theta / 2.0)
```

(continues on next page)

(continued from previous page)

```

aa, bb, cc, dd = a * a, b * b, c * c, d * d
bc, ad, ac, ab, bd, cd = b * c, a * d, a * c, a * b, b * d, c * d
return np.array([[aa + bb - cc - dd, 2 * (bc + ad), 2 * (bd - ac)],
                 [2 * (bc - ad), aa + cc - bb - dd, 2 * (cd + ab)],
                 [2 * (bd + ac), 2 * (cd - ab), aa + dd - bb - cc]])

```

8.5.1 Height

Move up or down by subtracting or adding height to the z coordinate of the foot positions

8.5.2 Pitch

Rotation around the Y axis by angle α

```

axis = [0, 1, 0]
alpha = np.radians(-10)
wireframe_rot_rf = np.dot(rotation_matrix(axis, alpha), wireframe[0])
wireframe_rot_rh = np.dot(rotation_matrix(axis, alpha), wireframe[2])

```

```

fig = plt.figure()
ax = fig.add_subplot()
ax.plot(wireframe[0][0], wireframe[0][2], color='k')
ax.plot(wireframe[2][0], wireframe[2][2], color='k')
ax.plot(wireframe[0][0][4], wireframe[0][2][4], 'ko')
ax.plot(wireframe[2][0][4], wireframe[2][2][4], 'ko')

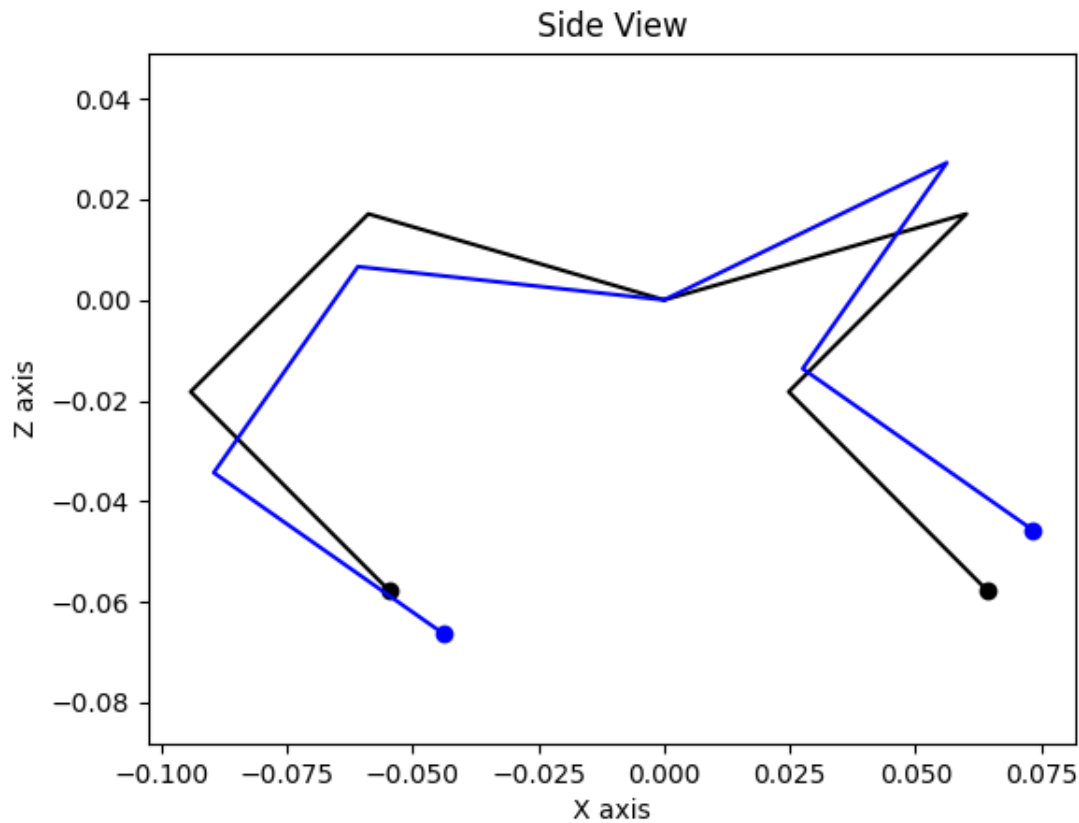
ax.plot(wireframe_rot_rf[0], wireframe_rot_rf[2], color='b')
ax.plot(wireframe_rot_rh[0], wireframe_rot_rh[2], color='b')
ax.plot(wireframe_rot_rf[0][4], wireframe_rot_rf[2][4], 'bo')
ax.plot(wireframe_rot_rh[0][4], wireframe_rot_rh[2][4], 'bo')

plt.axis('equal')
ax.set_xlabel('X axis')
ax.set_ylabel('Z axis')

plt.grid(visible=False, which='major')
plt.title('Side View')

```

```
Text(0.5, 1.0, 'Side View')
```



8.5.3 Roll

Rotation around the X axis by angle α

```
axis = [1, 0, 0]
alpha = np.radians(10)
wireframe_rot_rf = np.dot(rotation_matrix(axis, alpha), wireframe[0])
wireframe_rot_lf = np.dot(rotation_matrix(axis, alpha), wireframe[1])
```

```
print(np.transpose(wireframe[0])[4])
print(np.transpose(wireframe_rot_rf)[4])
print(gc.mpInverseKin('rf', np.transpose(wireframe_rot_rf)[4]))
```

```
[ 0.06438264 -0.0432      -0.05785332]
[ 0.06438264 -0.03249757 -0.064476   ]
[0.13255113886842565, 0.7164885251500537, -1.440155432715426]
```

```
fig = plt.figure()
ax = fig.add_subplot()
ax.plot(wireframe[0][1], wireframe[0][2], color='k')
ax.plot(wireframe[1][1], wireframe[1][2], color='k')
ax.plot(wireframe[0][1][4], wireframe[0][2][4], 'ko')
```

(continues on next page)

(continued from previous page)

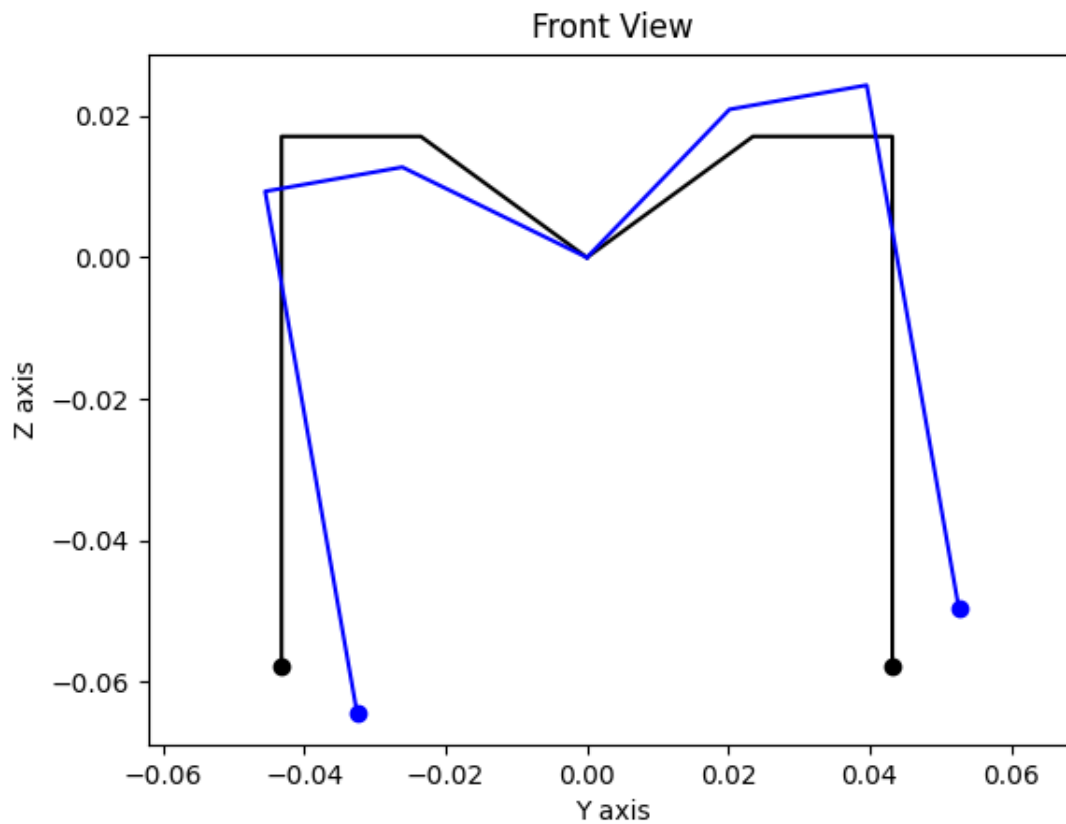
```

ax.plot(wireframe[1][1][4], wireframe[1][2][4], 'ko')
ax.plot(wireframe_rot_rf[1], wireframe_rot_rf[2], color='b')
ax.plot(wireframe_rot_lf[1], wireframe_rot_lf[2], color='b')
ax.plot(wireframe_rot_rf[1][4], wireframe_rot_rf[2][4], 'bo')
ax.plot(wireframe_rot_lf[1][4], wireframe_rot_lf[2][4], 'bo')

plt.axis('equal')
ax.set_xlabel('Y axis')
ax.set_ylabel('Z axis')
plt.grid(visible=False, which='major')
plt.title('Front View')

```

```
Text(0.5, 1.0, 'Front View')
```



8.5.4 Yaw

Rotation around the Z axis by angle α

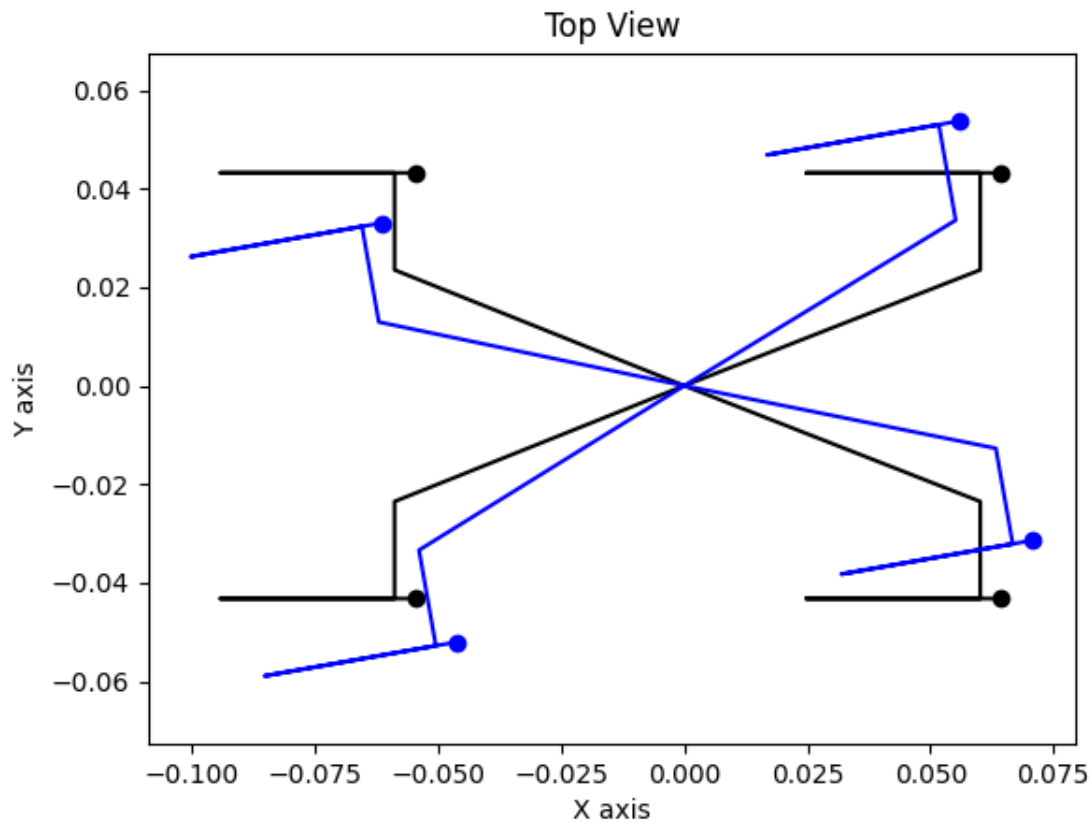
```
axis = [0, 0, 1]
alpha = np.radians(10)
wireframe_rot_rf = np.dot(rotation_matrix(axis, alpha), wireframe[0])
wireframe_rot_lf = np.dot(rotation_matrix(axis, alpha), wireframe[1])
wireframe_rot_rh = np.dot(rotation_matrix(axis, alpha), wireframe[2])
wireframe_rot_lh = np.dot(rotation_matrix(axis, alpha), wireframe[3])
```

```
fig = plt.figure()
ax = fig.add_subplot()
ax.plot(wireframe[0][0], wireframe[0][1], color='k')
ax.plot(wireframe[1][0], wireframe[1][1], color='k')
ax.plot(wireframe[2][0], wireframe[2][1], color='k')
ax.plot(wireframe[3][0], wireframe[3][1], color='k')
ax.plot(wireframe[0][0][4], wireframe[0][1][4], 'ko')
ax.plot(wireframe[1][0][4], wireframe[1][1][4], 'ko')
ax.plot(wireframe[2][0][4], wireframe[2][1][4], 'ko')
ax.plot(wireframe[3][0][4], wireframe[3][1][4], 'ko')

ax.plot(wireframe_rot_rf[0], wireframe_rot_rf[1], color='b')
ax.plot(wireframe_rot_lf[0], wireframe_rot_lf[1], color='b')
ax.plot(wireframe_rot_rh[0], wireframe_rot_rh[1], color='b')
ax.plot(wireframe_rot_lh[0], wireframe_rot_lh[1], color='b')
ax.plot(wireframe_rot_rf[0][4], wireframe_rot_rf[1][4], 'bo')
ax.plot(wireframe_rot_lf[0][4], wireframe_rot_lf[1][4], 'bo')
ax.plot(wireframe_rot_rh[0][4], wireframe_rot_rh[1][4], 'bo')
ax.plot(wireframe_rot_lh[0][4], wireframe_rot_lh[1][4], 'bo')

plt.axis('equal')
ax.set_xlabel('X axis')
ax.set_ylabel('Y axis')
plt.grid(visible=False, which='major')
plt.title('Top View')
```

```
Text(0.5, 1.0, 'Top View')
```



```
lp = gc.getRPYH(0, 0, 0, 0.01)
for i in range(len(gc.legs)):
    for j in range(len(gc.joints)):
        fn = "%s_%s" % (gc.legs[i], gc.joints[j])
        with open("../controller/servos/rpy/%s" % fn, "w") as f:
            if 'hip' in fn:
                f.write("%.10f\n" % lp[i][j])
            if 'upper' in fn:
                f.write("%.10f\n" % (float(lp[i][j]) - np.pi))
            if 'lower' in fn:
                f.write("%.10f\n" % -lp[i][j])
```


A FRESH LOOK AT GAITS

When we looked at gaits earlier our focus was on getting the robot to move. We developed a foot trajectory that we applied to all 4 legs with a phase shift to opposite side front and hind legs.

We now look at different ways we can move the legs. There are different names used in literature and we look at gaits that we call walk, trot and gallop.

9.1 Reference

The following sources helped me to refine my understanding

A Quadruped Robot Exhibiting Spontaneous Gait Transitions from Walking to Trotting to Galloping

WoLF: the Whole-body Locomotion Framework for Quadruped Robots

champ

9.2 Foot Trajectory planning

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

```
from gaits3D import GaitController3D
gc = GaitController3D()
```

```
trajectory = np.zeros((gc.number_of_points * 2 - 2, 3))
```

```
trajectory[:, 0] = gc._stance_phase(100, False) + list(gc._swing_phase()[0])
trajectory[:, 2][gc.number_of_points:] = list(gc._swing_phase()[1])
```

```
trajectory1 = np.transpose(np.dot(gc._rotation_matrix([0, 0, 1], np.radians(-10)), np.
    ↳ transpose(trajectory)))
```

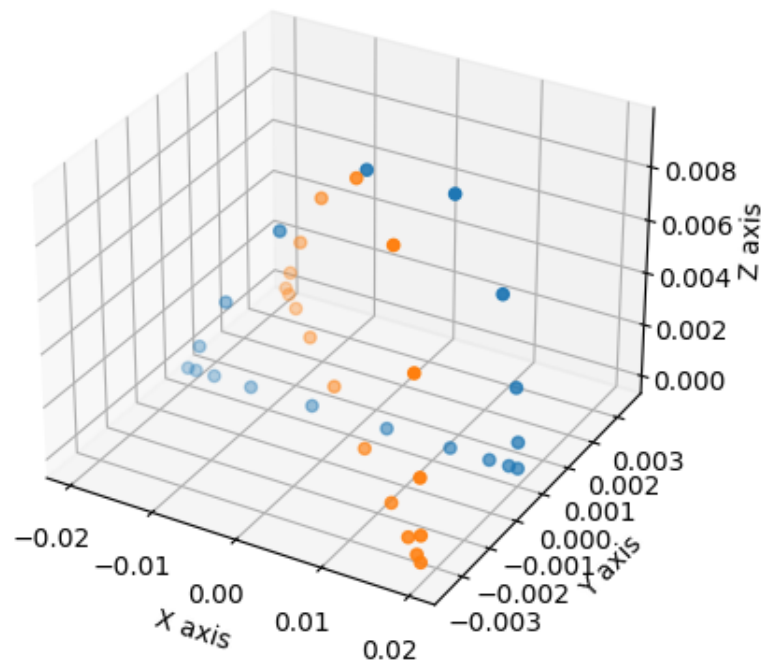
```
fig = plt.figure()
ax = fig.add_subplot(projection='3d')
ax.scatter(trajectory[:, 0], trajectory[:, 1], trajectory[:, 2])
```

(continues on next page)

(continued from previous page)

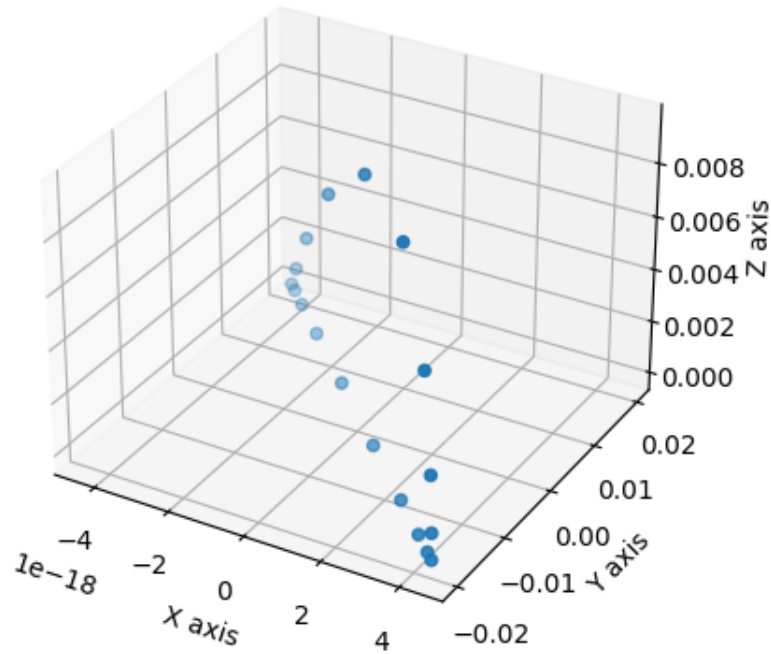
```
ax.scatter(trajecory1[:, 0], trajecory1[:, 1], trajecory1[:, 2])
ax.set_xlabel('X axis')
ax.set_ylabel('Y axis')
ax.set_zlabel('Z axis')
```

```
Text(0.5, 0, 'Z axis')
```



```
traj = gc._trajectory(np.radians(90), 100, False)
fig = plt.figure()
ax = fig.add_subplot(projection='3d')
ax.scatter(traj[:, 0], traj[:, 1], traj[:, 2])
ax.set_xlabel('X axis')
ax.set_ylabel('Y axis')
ax.set_zlabel('Z axis')
```

```
Text(0.5, 0, 'Z axis')
```



```

traj = gc._trajectory(np.radians(90), 100, False)
for leg in range(len(gc.legs)):
    for j in range(len(gc.joints)):
        fn = "%s_%s" % (gc.legs[leg], gc.joints[j])
        with open("../controller/servos/leg-test3d/%s" % fn, "w") as f:
            for tick in range(len(traj)):
                fp = gc.mpInverseKin(gc.legs[leg], gc.neutral_foot_position[leg] +
    ↪traj[tick])
                if 'hip' in fn:
                    f.write("%.10f\n" % fp[j])
                if 'upper' in fn:
                    f.write("%.10f\n" % (float(fp[j])-np.pi))
                if 'lower' in fn:
                    f.write("%.10f\n" % -fp[j])

```


PARAMETERS

The parameters that we want to expose to steer the robot are forward (x direction) and sideward (y direction) velocity and we call them `vel_x` and `vel_y`

10.1 Velocity

`vel_x` and `vel_y` are in a range of `[-100, 100]` and should be considered as percentage of maximum speed. We calculate the speed as the maximum of the absolute values of `vel_x` and `vel_y`

10.2 Direction

The direction is the direction of the vector `[vel_x, vel_y]` To turn the body of the robot we will move the front legs in the indicated direction but keep the hind legs in the x direction.

10.3 Number of points in trajectory

A foot trajectory is the combination of the stance phase and the swing phase, each of these phases is approximated by a number of points. The total number of points in all phases are the number of points in the trajectory.

10.4 Frequency

We process each point of the trajectory in a loop. The frequency determines how many points we process per second. If we increase the number of points by keeping the frequency we slow down the movement. If we decrease the frequency and keep the same number of points we also slow down the movement.

10.5 Step length

Length of a step

10.6 Step height

Height of a step.

```
# Direction
for vel_x in [-100, -50, 0, 50, 100]:
    for vel_y in [-100, -50, 0, 50, 100]:
        alpha = np.arctan2(vel_y, vel_x)
        print("vel_x: %s vel_y: %s alpha: %s" %(vel_x, vel_y, np.degrees(alpha)))
```

```
vel_x: -100 vel_y: -100 alpha: -135.0
vel_x: -100 vel_y: -50 alpha: -153.434948822922
vel_x: -100 vel_y: 0 alpha: 180.0
vel_x: -100 vel_y: 50 alpha: 153.434948822922
vel_x: -100 vel_y: 100 alpha: 135.0
vel_x: -50 vel_y: -100 alpha: -116.56505117707799
vel_x: -50 vel_y: -50 alpha: -135.0
vel_x: -50 vel_y: 0 alpha: 180.0
vel_x: -50 vel_y: 50 alpha: 135.0
vel_x: -50 vel_y: 100 alpha: 116.56505117707799
vel_x: 0 vel_y: -100 alpha: -90.0
vel_x: 0 vel_y: -50 alpha: -90.0
vel_x: 0 vel_y: 0 alpha: 0.0
vel_x: 0 vel_y: 50 alpha: 90.0
vel_x: 0 vel_y: 100 alpha: 90.0
vel_x: 50 vel_y: -100 alpha: -63.43494882292201
vel_x: 50 vel_y: -50 alpha: -45.0
vel_x: 50 vel_y: 0 alpha: 0.0
vel_x: 50 vel_y: 50 alpha: 45.0
vel_x: 50 vel_y: 100 alpha: 63.43494882292201
vel_x: 100 vel_y: -100 alpha: -45.0
vel_x: 100 vel_y: -50 alpha: -26.56505117707799
vel_x: 100 vel_y: 0 alpha: 0.0
vel_x: 100 vel_y: 50 alpha: 26.56505117707799
vel_x: 100 vel_y: 100 alpha: 45.0
```

```
# Speed
for vel_x in [-100, -50, 0, 50, 100]:
    for vel_y in [-100, -50, 0, 50, 100]:
        speed = max(abs(vel_y), abs(vel_x))
        print("vel_x: %s vel_y: %s speed: %s" %(vel_x, vel_y, speed))
```

```
vel_x: -100 vel_y: -100 speed: 100
vel_x: -100 vel_y: -50 speed: 100
vel_x: -100 vel_y: 0 speed: 100
vel_x: -100 vel_y: 50 speed: 100
vel_x: -100 vel_y: 100 speed: 100
vel_x: -50 vel_y: -100 speed: 100
```

(continues on next page)

(continued from previous page)

```

vel_x: -50 vel_y: -50 speed: 50
vel_x: -50 vel_y: 0 speed: 50
vel_x: -50 vel_y: 50 speed: 50
vel_x: -50 vel_y: 100 speed: 100
vel_x: 0 vel_y: -100 speed: 100
vel_x: 0 vel_y: -50 speed: 50
vel_x: 0 vel_y: 0 speed: 0
vel_x: 0 vel_y: 50 speed: 50
vel_x: 0 vel_y: 100 speed: 100
vel_x: 50 vel_y: -100 speed: 100
vel_x: 50 vel_y: -50 speed: 50
vel_x: 50 vel_y: 0 speed: 50
vel_x: 50 vel_y: 50 speed: 50
vel_x: 50 vel_y: 100 speed: 100
vel_x: 100 vel_y: -100 speed: 100
vel_x: 100 vel_y: -50 speed: 100
vel_x: 100 vel_y: 0 speed: 100
vel_x: 100 vel_y: 50 speed: 100
vel_x: 100 vel_y: 100 speed: 100

```

```

# Align number of points with speed
number_of_points = 10
speed = 55
number_of_points + int((100 - speed)/2)

```

32

10.7 Gait Scheduling

The sequence how we move each leg determines the gait. There are different gaits and even different names for these gaits. We study walk, trot and gallop.

```

number_of_points = 10
stance_n_o_p = number_of_points
swing_n_o_p = number_of_points - 2

```

```

fig, axs = plt.subplots(ncols=2, nrows=2)
for i in range(4):
    axs[0,0].plot([0, 3*stance_n_o_p + swing_n_o_p], [i+1, i+1], 'k', linewidth=10)
    start = i * stance_n_o_p
    axs[0,0].plot([start, start + swing_n_o_p], [i+1, i+1], 'b', linewidth=10)
for i in range(4):
    axs[1,0].plot([0, stance_n_o_p + swing_n_o_p], [i+1, i+1], 'k', linewidth=10)
    if i % 2:
        start = 0
    else:
        start = stance_n_o_p
    axs[1,0].plot([start, start + swing_n_o_p], [i+1, i+1], 'b', linewidth=10)
for i in range(4):
    axs[1,1].plot([0, stance_n_o_p + swing_n_o_p], [i+1, i+1], 'k', linewidth=10)
    if i < 2:

```

(continues on next page)

(continued from previous page)

```

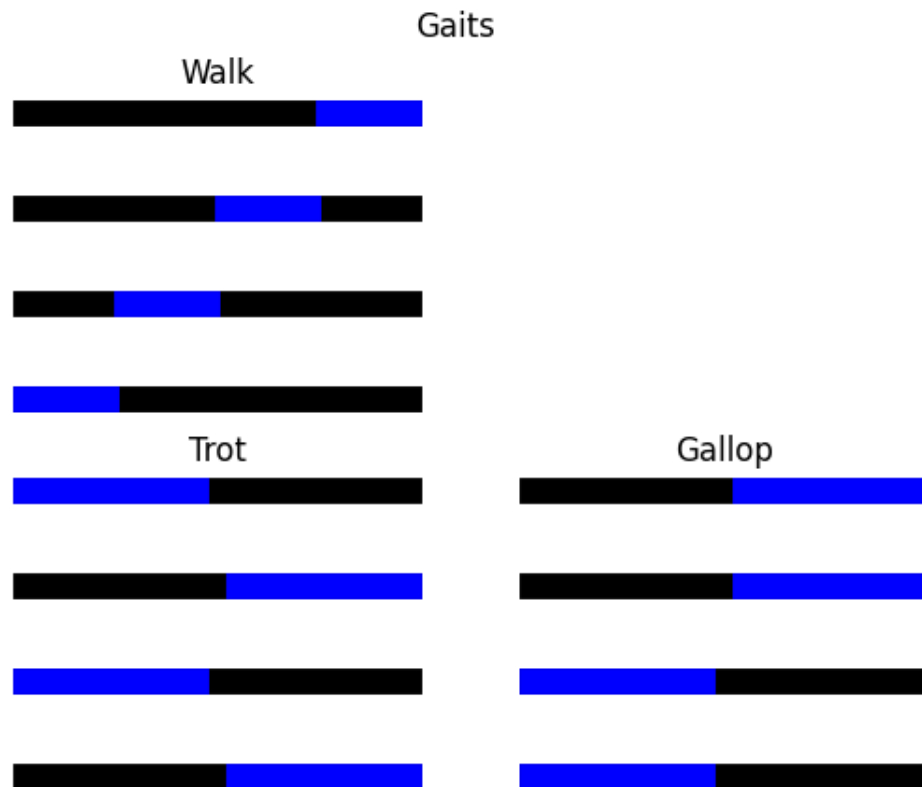
    start = 0
    else:
        start = stance_n_o_p
    axs[1,1].plot([start, start + swing_n_o_p], [i+1, i+1], 'b', linewidth=10)

axs[0,1].plot([0], [0], 'k', linewidth=0)

axs[0,0].axis('off')
axs[0,1].axis('off')
axs[1,0].axis('off')
axs[1,1].axis('off')

plt.suptitle('Gaits')
axs[0,0].set_title('Walk')
axs[1,0].set_title('Trot')
axs[1,1].set_title('Gallop')
plt.grid(visible=False, which='major')

```



```

isWalking = True
speed = 100
swp = gc._swing_phase()
stp = gc._stance_phase(speed, isWalking)
traj_len = len(stp) + len(list(swp[0]))
trajectory = np.zeros((traj_len, 3))

```

(continues on next page)

(continued from previous page)

```
trajectory[:, 0] = stp + list(swp[0])
trajectory[:, 2][len(stp):] = list(swp[1])
```

```
trajectory
```

```
array([[ 0.02      ,  0.      ,  0.      ],
       [ 0.01976851,  0.      ,  0.      ],
       [ 0.01946748,  0.      ,  0.      ],
       [ 0.01907758,  0.      ,  0.      ],
       [ 0.01857522,  0.      ,  0.      ],
       [ 0.01793232,  0.      ,  0.      ],
       [ 0.01711661,  0.      ,  0.      ],
       [ 0.0160929 ,  0.      ,  0.      ],
       [ 0.01482561,  0.      ,  0.      ],
       [ 0.0132831 ,  0.      ,  0.      ],
       [ 0.01144382,  0.      ,  0.      ],
       [ 0.00930368,  0.      ,  0.      ],
       [ 0.00688321,  0.      ,  0.      ],
       [ 0.00423212,  0.      ,  0.      ],
       [ 0.00142851,  0.      ,  0.      ],
       [-0.00142851,  0.      ,  0.      ],
       [-0.00423212,  0.      ,  0.      ],
       [-0.00688321,  0.      ,  0.      ],
       [-0.00930368,  0.      ,  0.      ],
       [-0.01144382,  0.      ,  0.      ],
       [-0.0132831 ,  0.      ,  0.      ],
       [-0.01482561,  0.      ,  0.      ],
       [-0.0160929 ,  0.      ,  0.      ],
       [-0.01711661,  0.      ,  0.      ],
       [-0.01793232,  0.      ,  0.      ],
       [-0.01857522,  0.      ,  0.      ],
       [-0.01907758,  0.      ,  0.      ],
       [-0.01946748,  0.      ,  0.      ],
       [-0.01976851,  0.      ,  0.      ],
       [-0.02      ,  0.      ,  0.      ],
       [-0.01846521,  0.      ,  0.00099723],
       [-0.01504552,  0.      ,  0.00303655],
       [-0.00829051,  0.      ,  0.00634544],
       [ 0.00204055,  0.      ,  0.00948565],
       [ 0.01218792,  0.      ,  0.00948565],
       [ 0.01780867,  0.      ,  0.00634544],
       [ 0.01960035,  0.      ,  0.00303655],
       [ 0.01996106,  0.      ,  0.00099723]])
```

```
np.roll(trajectory, 2*len(list(swp[0])), axis=0)
```

```
array([[ -0.0160929 ,  0.      ,  0.      ],
       [ -0.01711661,  0.      ,  0.      ],
       [ -0.01793232,  0.      ,  0.      ],
       [ -0.01857522,  0.      ,  0.      ],
       [ -0.01907758,  0.      ,  0.      ],
       [ -0.01946748,  0.      ,  0.      ],
       [ -0.01976851,  0.      ,  0.      ]])
```

(continues on next page)

(continued from previous page)

```

[-0.02      , 0.      , 0.      ],
[-0.01846521, 0.      , 0.00099723],
[-0.01504552, 0.      , 0.00303655],
[-0.00829051, 0.      , 0.00634544],
[ 0.00204055, 0.      , 0.00948565],
[ 0.01218792, 0.      , 0.00948565],
[ 0.01780867, 0.      , 0.00634544],
[ 0.01960035, 0.      , 0.00303655],
[ 0.01996106, 0.      , 0.00099723],
[ 0.02      , 0.      , 0.      ],
[ 0.01976851, 0.      , 0.      ],
[ 0.01946748, 0.      , 0.      ],
[ 0.01907758, 0.      , 0.      ],
[ 0.01857522, 0.      , 0.      ],
[ 0.01793232, 0.      , 0.      ],
[ 0.01711661, 0.      , 0.      ],
[ 0.0160929 , 0.      , 0.      ],
[ 0.01482561, 0.      , 0.      ],
[ 0.0132831 , 0.      , 0.      ],
[ 0.01144382, 0.      , 0.      ],
[ 0.00930368, 0.      , 0.      ],
[ 0.00688321, 0.      , 0.      ],
[ 0.00423212, 0.      , 0.      ],
[ 0.00142851, 0.      , 0.      ],
[-0.00142851, 0.      , 0.      ],
[-0.00423212, 0.      , 0.      ],
[-0.00688321, 0.      , 0.      ],
[-0.00930368, 0.      , 0.      ],
[-0.01144382, 0.      , 0.      ],
[-0.0132831 , 0.      , 0.      ],
[-0.01482561, 0.      , 0.      ]])

```

```

angle = 0
gait_plan = gc._gait_plan(angle, 'trot', speed)

```

```
gait_plan
```

```

[array([[ 0.00000000e+00,  4.72692333e-01, -1.46888687e+00],
        [ 0.00000000e+00,  4.89908254e-01, -1.47760213e+00],
        [ 0.00000000e+00,  5.27975809e-01, -1.49569849e+00],
        [ 0.00000000e+00,  6.02710883e-01, -1.52635772e+00],
        [ 0.00000000e+00,  7.19487562e-01, -1.56024931e+00],
        [ 0.00000000e+00,  8.47633960e-01, -1.57399526e+00],
        [ 0.00000000e+00,  9.41936384e-01, -1.56301227e+00],
        [ 0.00000000e+00,  9.92911100e-01, -1.54634332e+00],
        [ 0.00000000e+00,  1.01629038e+00, -1.53523721e+00],
        [ 0.00000000e+00,  1.02631047e+00, -1.52966073e+00],
        [ 0.00000000e+00,  1.02837863e+00, -1.56440623e+00],
        [ 0.00000000e+00,  1.02568103e+00, -1.63284977e+00],
        [-5.55111512e-17,  1.02568103e+00, -1.63284977e+00],
        [-1.11022302e-16,  9.92795726e-01, -1.73448601e+00],
        [-1.66533454e-16,  8.79543832e-01, -1.80700560e+00],
        [-1.66533454e-16,  7.05247135e-01, -1.76487319e+00],
        [-1.11022302e-16,  5.75177742e-01, -1.64993230e+00],
        [-5.55111512e-17,  5.11434443e-01, -1.55213281e+00],

```

(continues on next page)

(continued from previous page)

```

[ 0.00000000e+00, 4.84103451e-01, -1.49584728e+00]]),
array([[ 0.00000000e+00, 1.02837863e+00, -1.56440623e+00],
[ 5.55111512e-17, 1.02568103e+00, -1.63284977e+00],
[-1.11022302e-16, 9.92795726e-01, -1.73448601e+00],
[-5.55111512e-17, 8.79543832e-01, -1.80700560e+00],
[-5.55111512e-17, 7.05247135e-01, -1.76487319e+00],
[-1.11022302e-16, 5.75177742e-01, -1.64993230e+00],
[ 5.55111512e-17, 5.11434443e-01, -1.55213281e+00],
[ 0.00000000e+00, 4.84103451e-01, -1.49584728e+00],
[ 0.00000000e+00, 4.72692333e-01, -1.46888687e+00],
[ 0.00000000e+00, 4.89908254e-01, -1.47760213e+00],
[ 0.00000000e+00, 5.27975809e-01, -1.49569849e+00],
[ 0.00000000e+00, 6.02710883e-01, -1.52635772e+00],
[ 0.00000000e+00, 7.19487562e-01, -1.56024931e+00],
[ 0.00000000e+00, 8.47633960e-01, -1.57399526e+00],
[ 0.00000000e+00, 9.41936384e-01, -1.56301227e+00],
[ 0.00000000e+00, 9.92911100e-01, -1.54634332e+00],
[ 0.00000000e+00, 1.01629038e+00, -1.53523721e+00],
[ 0.00000000e+00, 1.02631047e+00, -1.52966073e+00]]),
array([[ 0.00000000e+00, 1.02837863e+00, -1.56440623e+00],
[-5.55111512e-17, 1.02568103e+00, -1.63284977e+00],
[-1.11022302e-16, 9.92795726e-01, -1.73448601e+00],
[-1.66533454e-16, 8.79543832e-01, -1.80700560e+00],
[-1.66533454e-16, 7.05247135e-01, -1.76487319e+00],
[-1.11022302e-16, 5.75177742e-01, -1.64993230e+00],
[-5.55111512e-17, 5.11434443e-01, -1.55213281e+00],
[ 0.00000000e+00, 4.84103451e-01, -1.49584728e+00],
[ 0.00000000e+00, 4.72692333e-01, -1.46888687e+00],
[ 0.00000000e+00, 4.89908254e-01, -1.47760213e+00],
[ 0.00000000e+00, 5.27975809e-01, -1.49569849e+00],
[ 0.00000000e+00, 6.02710883e-01, -1.52635772e+00],
[ 0.00000000e+00, 7.19487562e-01, -1.56024931e+00],
[ 0.00000000e+00, 8.47633960e-01, -1.57399526e+00],
[ 0.00000000e+00, 9.41936384e-01, -1.56301227e+00],
[ 0.00000000e+00, 9.92911100e-01, -1.54634332e+00],
[ 0.00000000e+00, 1.01629038e+00, -1.53523721e+00],
[ 0.00000000e+00, 1.02631047e+00, -1.52966073e+00]]),
array([[ 0.00000000e+00, 4.72692333e-01, -1.46888687e+00],
[ 0.00000000e+00, 4.89908254e-01, -1.47760213e+00],
[ 0.00000000e+00, 5.27975809e-01, -1.49569849e+00],
[ 0.00000000e+00, 6.02710883e-01, -1.52635772e+00],
[ 0.00000000e+00, 7.19487562e-01, -1.56024931e+00],
[ 0.00000000e+00, 8.47633960e-01, -1.57399526e+00],
[ 0.00000000e+00, 9.41936384e-01, -1.56301227e+00],
[ 0.00000000e+00, 9.92911100e-01, -1.54634332e+00],
[ 0.00000000e+00, 1.01629038e+00, -1.53523721e+00],
[ 0.00000000e+00, 1.02631047e+00, -1.52966073e+00],
[ 0.00000000e+00, 1.02837863e+00, -1.56440623e+00],
[ 5.55111512e-17, 1.02568103e+00, -1.63284977e+00],
[-1.11022302e-16, 9.92795726e-01, -1.73448601e+00],
[-5.55111512e-17, 8.79543832e-01, -1.80700560e+00],
[-5.55111512e-17, 7.05247135e-01, -1.76487319e+00],
[-1.11022302e-16, 5.75177742e-01, -1.64993230e+00],
[ 5.55111512e-17, 5.11434443e-01, -1.55213281e+00],
[ 0.00000000e+00, 4.84103451e-01, -1.49584728e+00]]])

```


VISUALIZATION AND SIMULATION ENVIRONMENTS

11.1 Bullet Real-Time Physics Simulation

PyBullet

11.2 Open Dynamics Engine

ODE

11.3 Dynamic Animation and Robotics Toolkit

DART

11.4 Open Robot Control Software

OROCOS

11.5 Gazebo

Gazebo

11.6 Todo

This section needs more work

SOFTWARE FRAMEWORKS FOR THE DEVELOPMENT OF ROBOTIC SYSTEMS

Frameworks exist to ease the construction of complex robot systems by offering reusable building blocks that can be easily integrated.

12.1 Rocks

<https://www.rock-robotics.org> An example of an early(?) framework

12.2 ROS - Robot Operating System

<https://www.ros.org> Ros is widely used and comes in 2 versions, ROS1 and ROS2. Although the design principles do not differ too much between the two versions ROS1 is EOF (end of life) and replaced by ROS2. There are still many packages that are ROS1 only but as I am new to ROS in general I will focus on ROS2.

12.2.1 Todo

More details on ROS and some simple examples for ROS2 like <https://github.com/gnaur/simbot>