TECHNISCHE
UNIVERSITÄT
WIEN

Computational Bionics

Project Exercise 1

Ehrenstraßer, Nick
Ekström, Hugo
Muren, Axel

317.047 Computational Bionics, 2024W
16.12.2024

www.tuwien.at

Introduction

Spinal Cord Injury (SCI) impacts up to 500,000 people annually, often leading to paralysis and long-term immobility. Exoskeletons offer a promising solution to help affected individuals regain mobility and independence.

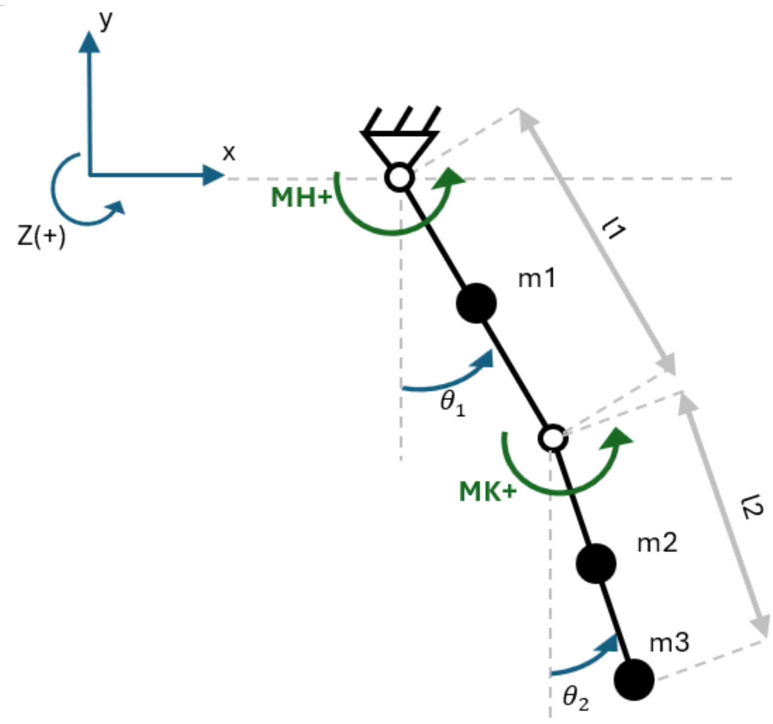
This project focuses on modeling key subcomponents of an exoskeleton, specifically the **leg**, **hip drive**, and **knee angle sensor**. The following protocol details the required steps, assumptions, and Python code used for the simulation.

TABLE OF CONTENTS

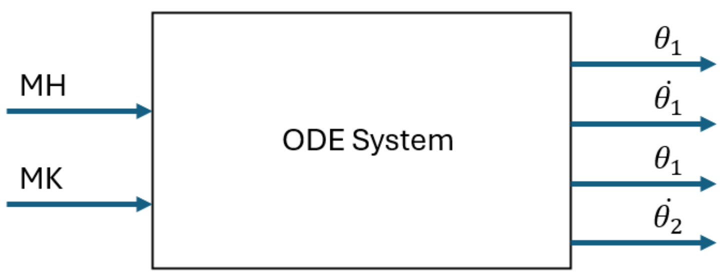
Computational Bionics | Exercise 1: Modeling the leg

In this exercise, an ODE system for a leg composed of three components (thigh m1, shank m2, and foot m3) is modeled. The following assumptions are made:

- The body segments are treated as point masses located at their respective centers of gravity.
- All angles and directions are defined as illustrated in the sketch below.
- The hip is considered a fixed point in space



In this model, the forward dynamics of the leg are simulated. This means that the input consists of the moments at the hip and knee, and the resulting angles and angular velocities are to be determined by the differential equation system, as shown in the graph below:



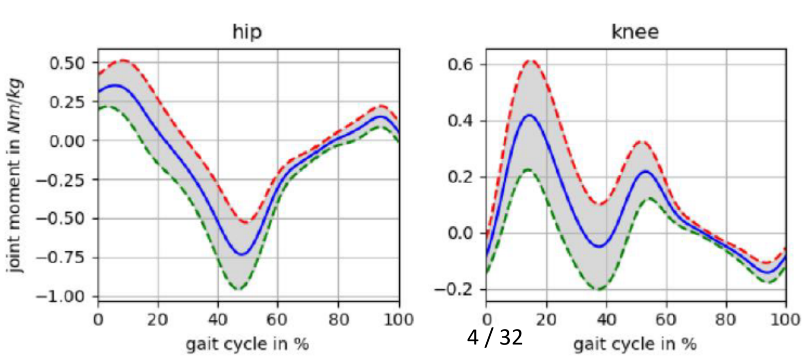
Part 1: Setting up the system

To start off, some python libraries are imported and all important input parameters are defined as numerical values:

In [29]:
```
import numpy as np
import sympy as smp
from scipy.integrate import solve_ivp
from scipy.interpolate import CubicSpline
import matplotlib.pyplot as plt
import time

# input parameters
g_val = 9.81 # m/s^2
bw = 100 # kg, body weight
m1_val = 0.09 * bw + 0.73 # kg, thigh mass
m2_val = 0.055 * bw - 0.43 # kg , shank mass
m3_val = 0.001 * bw + 0.34 # kg, foot mass
l1_val = 0.5  # m, thigh length
l2_val = 0.5  # m, shank length

print("\n",
    "g =", g_val, "m/s²\n",
    "bw =", bw, "kg\n",
    "m1 =", m1_val, "kg\n",
    "m2 =", m2_val, "kg\n",
    "m3 =", round(m3_val,4), "kg\n",
    "l1 =", l1_val, "m\n",
    "l2 =", l2_val, "m")
```

```
g = 9.81 m/s²
bw = 100 kg
m1 = 9.73 kg
m2 = 5.07 kg
m3 = 0.44 kg
l1 = 0.5 m
l2 = 0.5 m
```

To determine the moments, typical moment patterns from the literature are used. The graph shown below serves as a reference. From this graph, values were *approximated* at 0%, 10%, 20%, etc. of the gait cycle and then interpolated using a CubicSpline to create a continuous function.



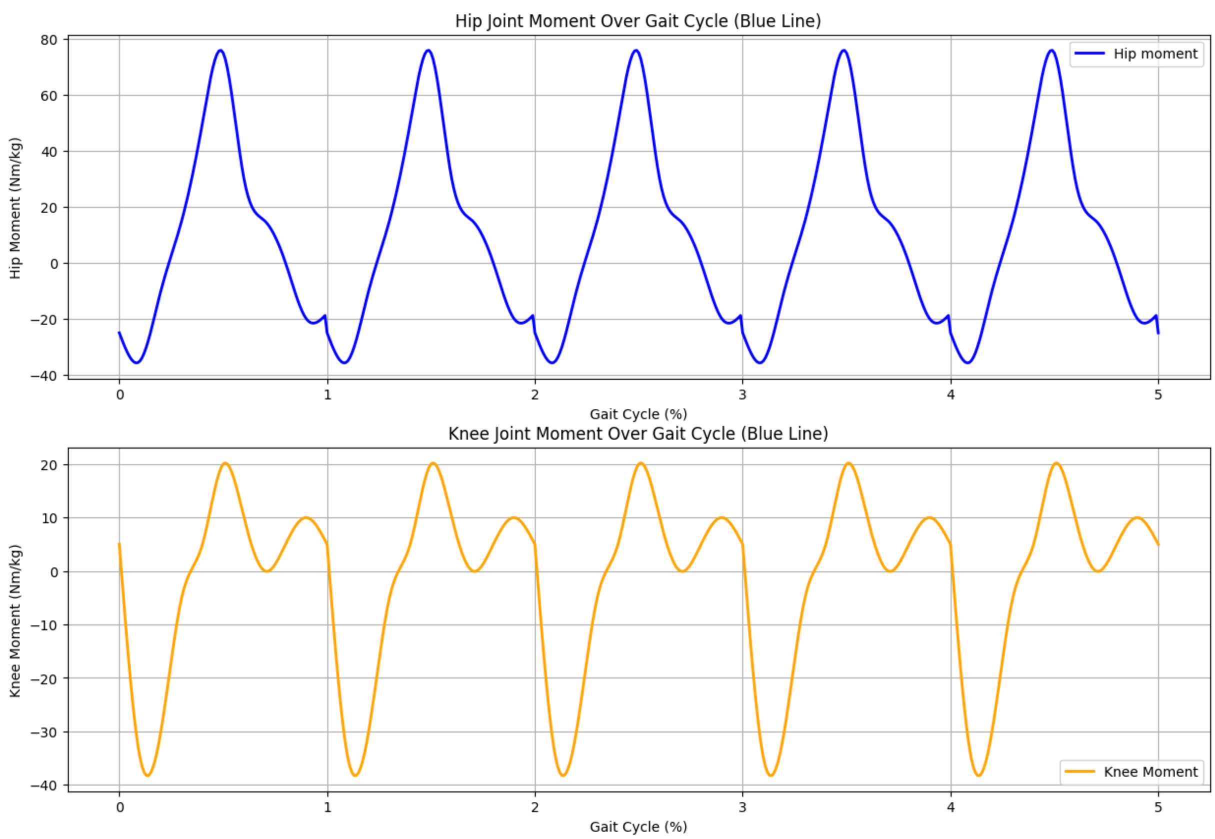
```
In [30]: # Key anchor points from graph
percentages= np.array([0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100])
MH_approx = np.array([0.25, 0.35, 0.10, -0.15, -0.50, -0.75, -0.30, -0.15, 0.00, 0.
MK_approx = np.array([-0.05, 0.35, 0.30, 0.05, -0.05, -0.20, -0.10, 0.0, -0.05, -0.

# scale to bodyweigth
MH_scaled = [element * (-1*bw) for element in MH_approx]
MK_scaled = [element * (-1*bw) for element in MK_approx]

gait_cycle = np.linspace(0, 100, 101)  # array = 0, 1, 2, 3, ..., 100, len=101

# Create a cubic spline interpolation function
cs_hip = CubicSpline(percentages, MH_scaled, bc_type='natural')
cs_knee = CubicSpline(percentages, MK_scaled, bc_type='natural')
```

The simulation is to be conducted for 5 seconds, where one gait cycle (0–100%) corresponds to 1 second.

```
In [31]: # simulation parameters
ts = 0 # simulation start time (s)
tf = 5 # simulation finish time (s)
duration = tf - ts # simulation duration (s)

frequency = 1  # gait-cycle per second
T_stride = 1 / frequency  # duration of a gait-cycle (s)

num_points = 501  # Auflösung

t_eval = np.linspace(0, duration, num_points)  # simulation time points
t_normalized = (t_eval % T_stride) / T_stride * 100 # Time normalization (current p

#Calculate moments for the normalized time points
MH_vals = cs_hip(t_normalized)  # hip moment
MK_vals = cs_knee(t_normalized)  # knee moment

MH_val = [round(value, 4) for value in MH_vals]
MK_val = [round(value, 4) for value in MK_vals]

# convert list to np.array
MH_val = np.array(MH_val)
MK_val = np.array(MK_val)

# Create a figure with two subplots
fig, axs = plt.subplots(2, 1, figsize=(15, 10))
```

```python
# Plot Hip joint moment on the first subplot
axs[0].plot(t_eval, MH_val, color='blue', linewidth=2, label='Hip moment')  # splin
axs[0].set_title('Hip Joint Moment Over Gait Cycle (Blue Line)')
axs[0].set_xlabel('Gait Cycle (%)')
axs[0].set_ylabel('Hip Moment (Nm/kg)')
axs[0].grid(True)
axs[0].legend()

# Plot Knee joint moment on the second subplot
axs[1].plot(t_eval, MK_val, color='orange', linewidth=2, label='Knee Moment')  # sp
axs[1].set_title('Knee Joint Moment Over Gait Cycle (Blue Line)')
axs[1].set_xlabel('Gait Cycle (%)')
axs[1].set_ylabel('Knee Moment (Nm/kg)')
axs[1].grid(True)
axs[1].legend()
```

Out[31]: <matplotlib.legend.Legend at 0x26620ee7c90>



Part 2: Create the Lagrange function and calculate kinetic quantities

The system of equations is created using the Python library "Sympy". In this process, equations are initially defined, computed, and simplified **symbolically**. Subsequently, the symbolic expressions need to be converted into numerical values.

In [21]:
```python
# Define symbolic variables
t, g = smp.symbols('t g')
m1, m2, m3 = smp.symbols('m1 m2 m3')  # Masses
l1, l2 = smp.symbols('l1 l2')  # Center of mass distances
```

```python
theta1, theta2 = smp.symbols(r'\theta_1 \theta_2', cls=smp.Function)

theta1 = theta1(t)
theta2 = theta2(t)

# Angular velocities and accelerations
theta1_d = smp.diff(theta1, t)
theta2_d = smp.diff(theta2, t)
theta1_dd = smp.diff(theta1_d, t)
theta2_dd = smp.diff(theta2_d, t)

# Input torques
MH = smp.Function('MH')(t)
MK = smp.Function('MK')(t)

# Positions of masses, according to Figure 1
x1 = l1 / 2 * smp.sin(theta1)
y1 = -l1 / 2 * smp.cos(theta1)

x2 = l1 * smp.sin(theta1) + l2 / 2 * smp.sin(theta2)
y2 = -l1 * smp.cos(theta1) - l2 / 2 * smp.cos(theta2)

x3 = l1 * smp.sin(theta1) + l2 * smp.sin(theta2)
y3 = -l1 * smp.cos(theta1) - l2 * smp.cos(theta2)

# Kinetic energy
T1 = 1 / 2 * m1 * (smp.diff(x1, t)**2 + smp.diff(y1, t)**2)
T2 = 1 / 2 * m2 * (smp.diff(x2, t)**2 + smp.diff(y2, t)**2)
T3 = 1 / 2 * m3 * (smp.diff(x3, t)**2 + smp.diff(y3, t)**2)
T = T1 + T2 + T3

# Potential energy
V1 = m1 * g * y1
V2 = m2 * g * y2
V3 = m3 * g * y3
V = V1 + V2 + V3

# Lagrangian
L = T - V

# Lagrange equations
LE1 = smp.diff(smp.diff(L, theta1_d), t) - smp.diff(L, theta1) - MH
LE2 = smp.diff(smp.diff(L, theta2_d), t) - smp.diff(L, theta2) - MK
LE1 = smp.simplify(LE1)
LE2 = smp.simplify(LE2)
```

The Lagrange equations now take the following form:

In [5]:
```python
LE1
```

Out[5]:

$$0.5gl_1m_1\sin(\theta_1(t)) + 1.0gl_1m_2\sin(\theta_1(t)) + 1.0gl_1m_3\sin(\theta_1(t)) + 0.25l_1^2m_1\frac{d^2}{dt^2}\theta_1(t) + 1.0l_1^2m_2\frac{d^2}{dt^2}\theta$$

$$(\theta_1(t) - \theta_2(t))\frac{d^2}{dt^2}\theta_2(t) + 1.0l_1l_2m_3\sin(\theta_1(t) - \theta_2(t))\left(\frac{d}{dt}\theta_2(t)\right)^2 + 1.0l_1l_2m_3\cos(\theta_1(t) - \theta_2(t))\frac{d^2}{dt^2}$$

In [6]:
```
LE2
```

Out[6]:

$$0.5gl_2m_2\sin(\theta_2(t)) + 1.0gl_2m_3\sin(\theta_2(t)) - 0.5l_1l_2m_2\sin(\theta_1(t)-\theta_2(t))\left(\frac{d}{dt}\theta_1(t)\right)^2 + 0.5l_1l_2m_2\cos$$
$$(\theta_1(t)-\theta_2(t))\frac{d^2}{dt^2}\theta_1(t) + 0.25l_2^2m_2\frac{d^2}{dt^2}\theta_2(t) + 1.0l_2^2m_3\frac{d^2}{dt^2}\theta_2(t) - 1.0\,\mathrm{MK}(t)$$

Solving for $\ddot{\theta}_1$ and $\ddot{\theta}_2$ is very time-consuming due to the use of symbolic equations. The time module can be used to track the computation time. In the final step, the resulting equations are converted into numerical functions using lambdify.

In [23]:
```
# Solve for angular accelerations (theta1_dd, theta2_dd)
st_sols = time.time()
sols = smp.solve([LE1, LE2], (theta1_dd, theta2_dd))
sols[theta1_dd] = smp.simplify(sols[theta1_dd])
sols[theta2_dd] = smp.simplify(sols[theta2_dd])
et_sols = time.time() - st_sols
print('solving successfull, computation time =', et_sols, 's')

# Simplify and convert symbolic solutions to numerical functions
theta1_dd_f = smp.lambdify((g, m1, m2, m3, l1, l2, theta1, theta2, theta1_d, theta2
theta2_dd_f = smp.lambdify((g, m1, m2, m3, l1, l2, theta1, theta2, theta1_d, theta2
```

```
solving successfull, computation time = 109.4919946193695 s
```

Part 3: Definition of the ODE System

Below the ODE system is defined.

To make the model more realistic, the moments are damped depending on the velocity. For this purpose, two damping constants, c_H (hip) and c_K (knee), are introduced. The magnitude of these values significantly influences the behavior of the system and is therefore chosen with caution.

Additionally, the knee moment is adpated to prevent hyperextension of the knee. This adjustment was made based on several simulation runs.

In [32]:
```
# damping coefficient
c_H = 10.0
c_K = 5.0

# ODE system with damping
def dSdt_damped(t, S, g, m1, m2, m3, l1, l2, MH_val, MK_val, t_eval):
    theta1, theta1_d, theta2, theta2_d = S

    # Interpolate input moments
    MH_interp = np.interp(t, t_eval, MH_val)
    MK_interp = np.interp(t, t_eval, MK_val)

    # Apply damping to the moments
    MH_damped = MH_interp - c_H * theta1_d
```

```
    MK_damped = MK_interp - c_K * theta2_d

    # Prevent overextending of the knee, when leg is moving forward
    if theta1 > 0 and theta2 > 0:
          MK_damped += 25 * (theta2-theta1)

    # Calculate angular accelerations
    theta1_dd = theta1_dd_f(g, m1, m2, m3, l1, l2, theta1, theta2, theta1_d, theta2
    theta2_dd = theta2_dd_f(g, m1, m2, m3, l1, l2, theta1, theta2, theta1_d, theta2

    return [theta1_d, theta1_dd, theta2_d, theta2_dd]
```

Next, the initial conditions are set:

- $\theta_1 = 20$ (deg)
- $\dot{\theta}_1 = 0$ (deg/s)
- $\theta_2 = 18$ (deg)
- $\dot{\theta}_1 = 0$ (deg/s)

and the ODE system is solved using *solve_ivp*.

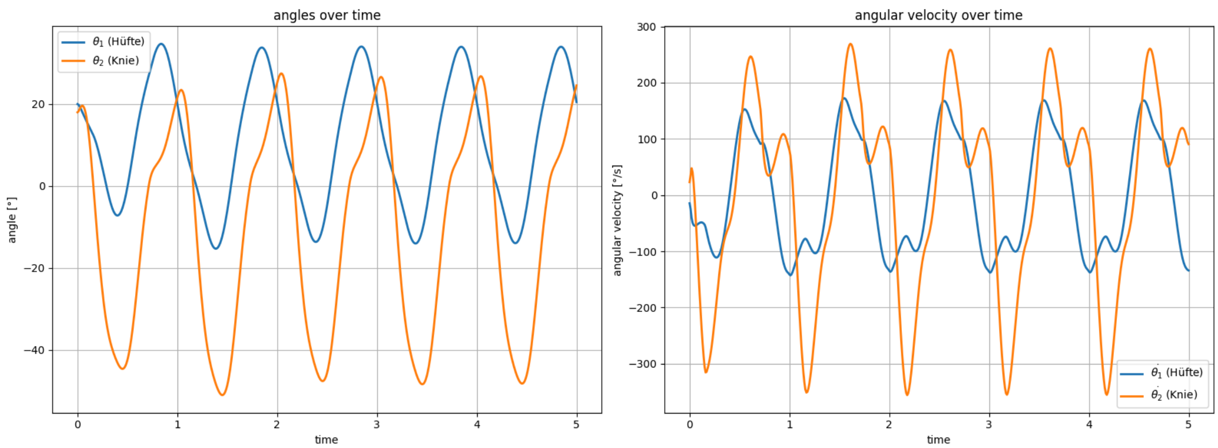
In [33]:
```
# Initial conditions
y0 = [np.radians(20), 0, np.radians(18), 0]

# Solve ODE with damping
solution_damped = solve_ivp(
    dSdt_damped,
    (ts, tf),
    y0,
    t_eval=t_eval,
    args=(g_val, m1_val, m2_val, m3_val, l1_val, l2_val, MH_val, MK_val, t_eval),
    method='RK45',
    max_step= 0.001
)

# Extract results
theta1_sol = solution_damped.y[0] # theta1
theta2_sol = solution_damped.y[2] # theta2

theta1_d_sol = solution_damped.y[1] # theta1_d
theta2_d_sol = solution_damped.y[3] # theta2_d
```

Part 4: visualize the results

To visualize and understand the results of the simulationthe angles (left plot) and angular velocities (right plots) are plotted over the evaluation time

In [35]:
```
fig, axs = plt.subplots(1, 2, figsize=(16, 6))

# Plot the angles on the first subplot
axs[0].plot(t_eval, np.degrees(theta1_sol), label=r'$\theta_1$ (hip)', linewidth=2)
axs[0].plot(t_eval, np.degrees(theta2_sol), label=r'$\theta_2$ (knee)', linewidth=2
```

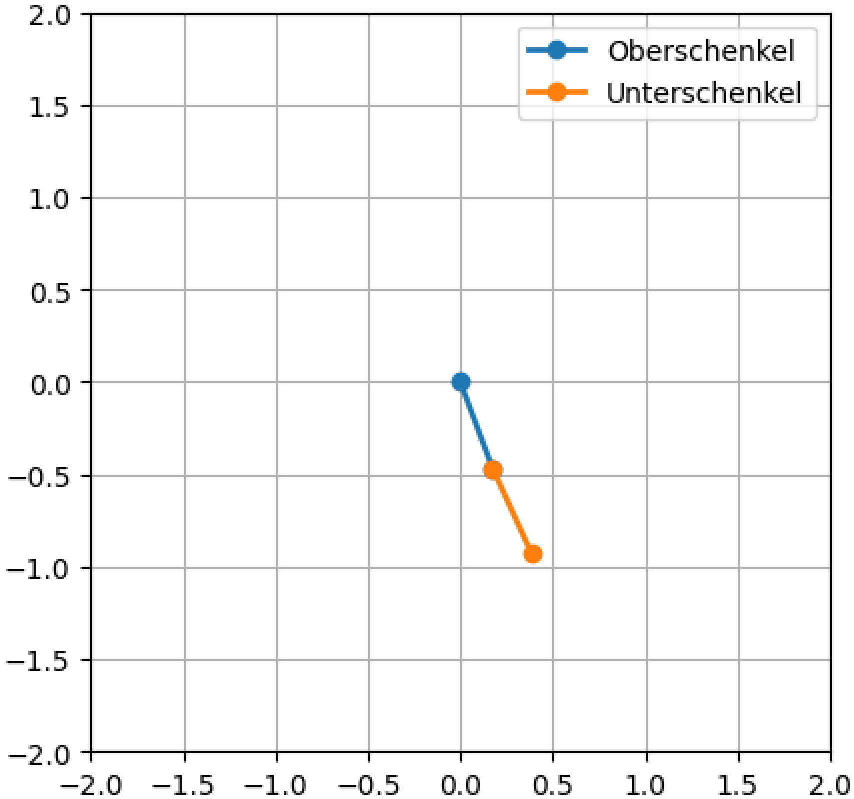
```
axs[0].set_xlabel('time')
axs[0].set_ylabel('angle [°]')
axs[0].set_title('angle over time')
axs[0].legend()
axs[0].grid()

# Plot the angular velocities on the second subplot
axs[1].plot(t_eval, np.degrees(np.gradient(theta1_sol, t_eval)), label=r'$\dot{\the
axs[1].plot(t_eval, np.degrees(np.gradient(theta2_sol, t_eval)), label=r'$\dot{\the
axs[1].set_xlabel('time')
axs[1].set_ylabel('angular velocity [°/s]')
axs[1].set_title('angular velocity over time')
axs[1].legend()
axs[1].grid()

# Show the plots
plt.tight_layout()
plt.show()
```



The amplitudes remain roughly the same. The system appears to be stable. To better interpret the result, an animation of the leg is created:

In [37]:
```
from matplotlib.animation import FuncAnimation, PillowWriter

# hip: fixed point in space (0, 0)
x_H, y_H = 0, 0

# knee:
x_K = l1_val * np.sin(theta1_sol)
y_K = -l1_val * np.cos(theta1_sol)

# foot:
x_S = x_K + l2_val * np.sin(theta2_sol)
y_S = y_K - l2_val * np.cos(theta2_sol)

# generate animation
fig, ax = plt.subplots()
ax.set_xlim(-2, 2)
ax.set_ylim(-2, 2)
ax.set_aspect('equal')
ax.grid()
```

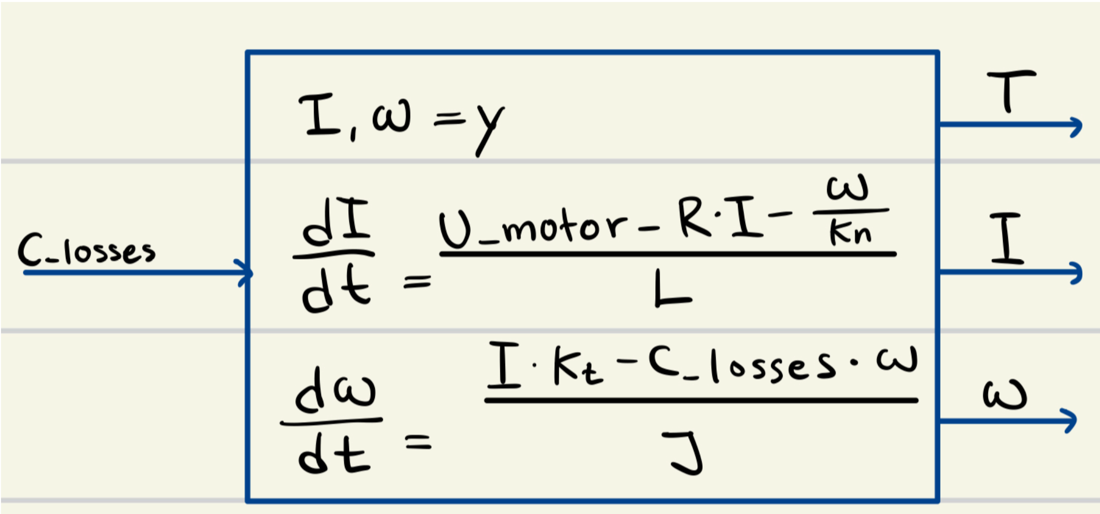
```python
# lines for thigh and shank
line_thigh, = ax.plot([], [], 'o-', label="Oberschenkel", lw=2)
line_shank, = ax.plot([], [], 'o-', label="Unterschenkel", lw=2)

# function to initialise the animation
def init():
    line_thigh.set_data([], [])
    line_shank.set_data([], [])
    return line_thigh, line_shank

# function to update the animation
def update(frame):
    # update positions
    x_data_thigh = [x_H, x_K[frame]]
    y_data_thigh = [y_H, y_K[frame]]
    x_data_shank = [x_K[frame], x_S[frame]]
    y_data_shank = [y_K[frame], y_S[frame]]

    line_thigh.set_data(x_data_thigh, y_data_thigh)
    line_shank.set_data(x_data_shank, y_data_shank)
    return line_thigh, line_shank

ani = FuncAnimation(
    fig, update, frames=len(t_eval), init_func=init, blit=True, interval=10
)

# save animation as GIF
gif_path = "Bionics-EX1_animation1.gif"  # name of the file
ani.save(gif_path, writer=PillowWriter(fps=30))

plt.legend()
plt.show()
```



The simulation shows the movement of the leg during the selected simulation period. The movement still appears somewhat uneven. Hyperextension of the knee could not be completely prevented. In the next step, the moments and their damping can be further adjusted to achieve a more realistic simulation.

Modeling the drive

According to the definition of the input variables, they "**are not influenced by the behavior of the system**". With this definiton in mind, the input variables were interpreted to be the **c_losses** constant alone. Because both the current and the omega affect each other in solving the differential equations, whereas the constant can be tweaked to adjust the three output variables.



Relevant packages were imported.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp
```

Then the relevant constants were computed and listed from the data sheet. A comment here is that to compute the speed constant, Kn, it was said to computed by dividing the rated voltage of the motor by the no load speed. According to the data sheet the rated voltage was **40 V**. We made the assumption that in our case, since we are using a 16 V motor, the rated voltage is **16 V**. We also introduced a dampening coefficient which is named "**c_losses**" here. This coefficient is multiplied by the omega to get the mechanical torque of the motor. Here, the coefficient is **0.0005**. The reason this specific value was chosen was to get a rotational speed of around **80%** of the no load speed.

In [2]:
```
U_motor = 16 #V
R = 151e-3 #ohm
L = 121e-6 #H
n0 = 12916 #rpm, No Load speed
n0_rad = n0 * (2 * np.pi / 60) #rad/s, No Load angular speed
J = 0.056e-4 #kg*m^2, Rotor inertia
kn = n0_rad / U_motor #Speed constant
gear_ratio = 60
efficiency = 0.85
kt = 30e-3  # Torque constant [Nm/A]
c_losses = 0.0005
```

The next step was to define the differential equations. The state variables were the current, **I**, and the angualar velocity, **omega**. The initail states were also defined here, where both the initial current and angular velocity were set to zero. The simulation time was 0.05 seconds because more time was not needed for the model to stabalize.

In [3]:
```
def motor_dynamics(t, y):
    I, omega = y  # State variables: current (I) and angular velocity (omega)
    # Differential equations
    dI_dt = (U_motor - R * I - omega / kn) / L
    domega_dt = (I * kt - c_losses * omega) / J

    return [dI_dt, domega_dt]

y0 = [0, 0]
t_s = 0
t_e = 0.05
t_eval = np.linspace(t_s, t_e, 1000)
```

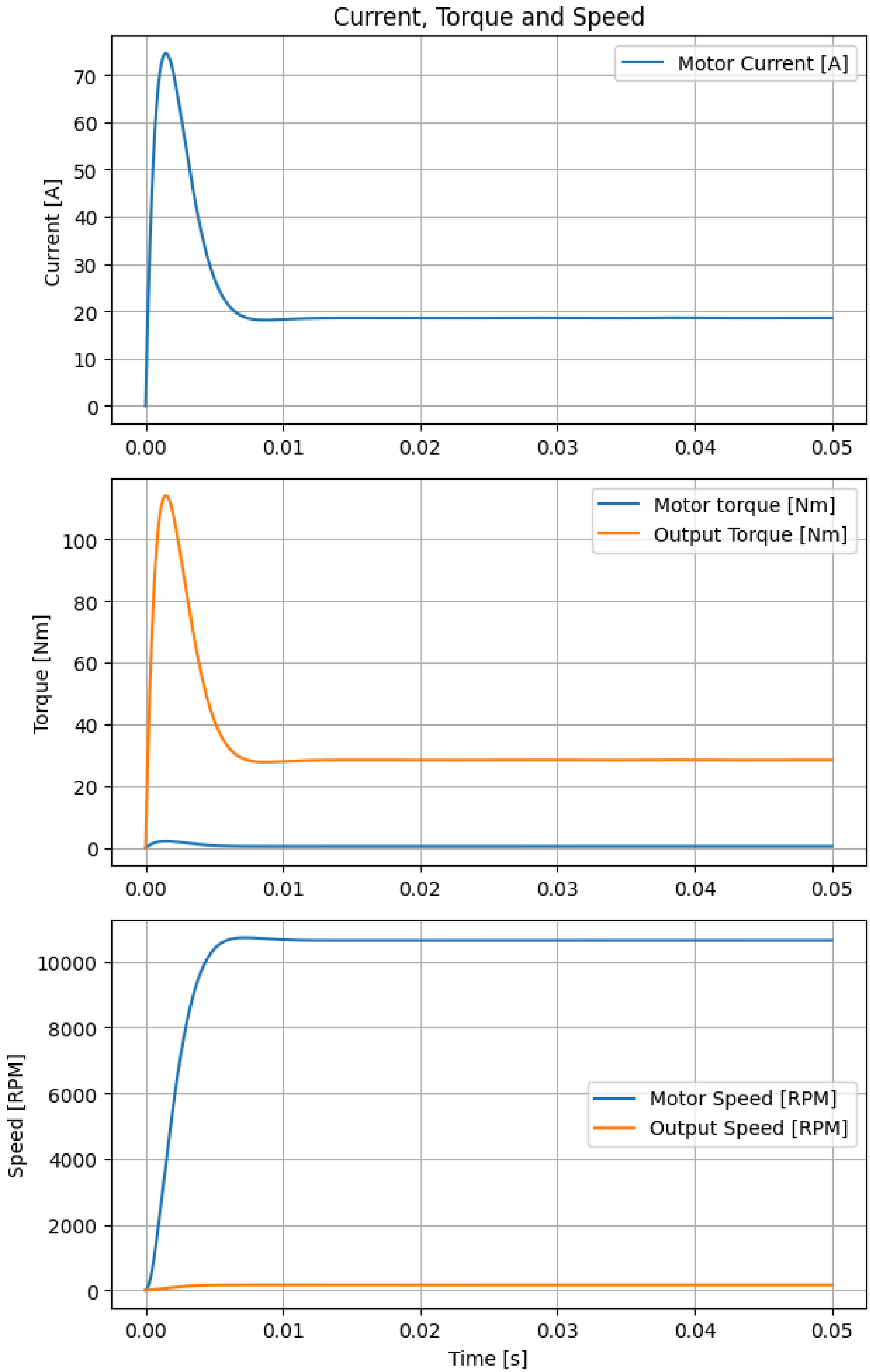
Next, the ODE's were solved and the results were given appropriate names. The angular velocity and the torque for the output were also introduced, which depend on the gear ratio and the efficiency of the gear box.

In [4]:
```
sol = solve_ivp(motor_dynamics, (t_s,t_e), y0, t_eval = t_eval, method='RK45')

# Extract results
t = sol.t # Time [s]
I = sol.y[0] # Current [A]
Torque = kt * I  # Torque [Nm]
Torque_out = Torque * gear_ratio * efficiency #Torque [Nm]
omega_motor = sol.y[1]  # Motor angular velocity [rad/s]
omega_output = (omega_motor * efficiency) / gear_ratio   # Output angular veloci
```

Then the angular velocities were converted to angular speed and plotted together with the current and the torque, which was computed by multiplying the current with the torqe constant "**kt**".

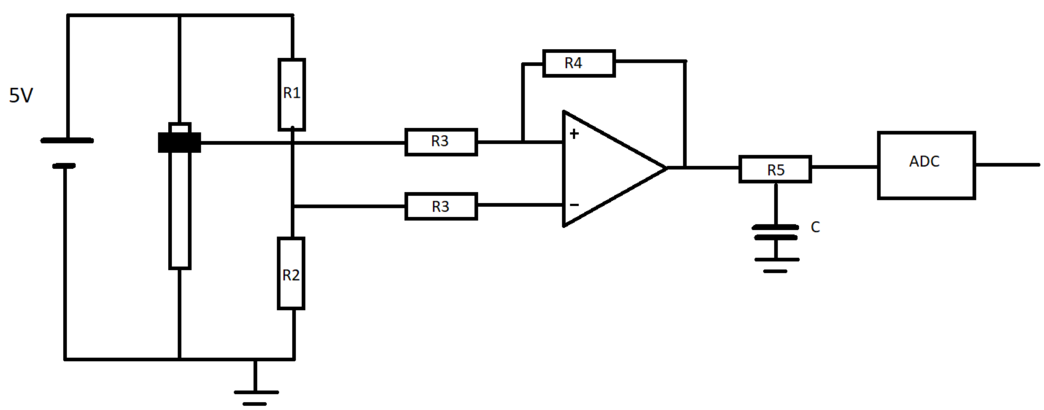
In [5]:
```
rpm_motor = omega_motor * 60 / (2 * np.pi)
rpm_output = omega_output * 60 / (2 * np.pi)

plt.figure().set_figheight(10)
plt.subplot(3, 1, 1)
plt.plot(t, I, label='Motor Current [A]')
plt.ylabel('Current [A]')
plt.title('Current, Torque and Speed')
plt.grid()
plt.legend()

plt.subplot(3, 1, 2)
plt.plot(t, Torque, label='Motor torque [Nm]')
plt.plot(t, Torque_out, label='Output Torque [Nm]')
plt.ylabel('Torque [Nm]')
plt.grid()
plt.legend()

# Speed plot
plt.subplot(3, 1, 3)
```

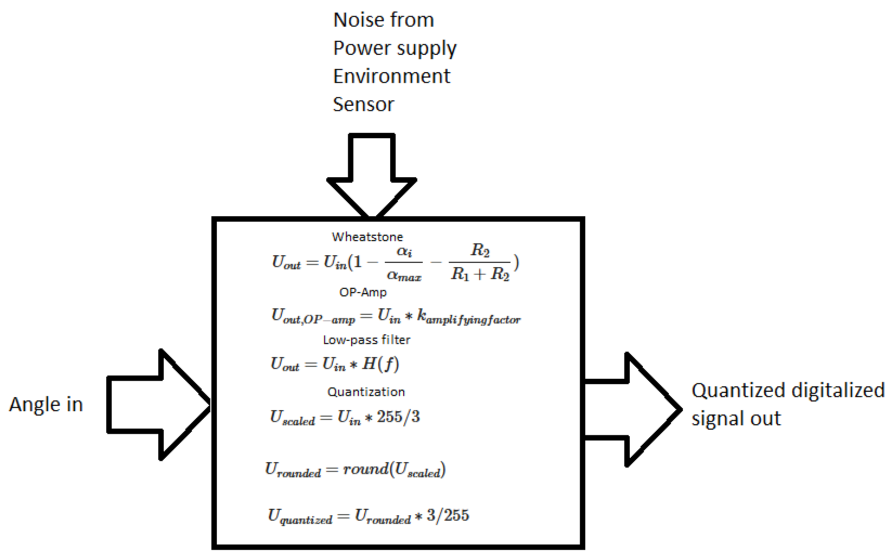
```
plt.plot(t, rpm_motor, label='Motor Speed [RPM]')
plt.plot(t, rpm_output, label='Output Speed [RPM]')
plt.xlabel('Time [s]')
plt.ylabel('Speed [RPM]')
plt.grid()
plt.legend()

plt.tight_layout()
plt.show()
```


Current, Torque and Speed
Motor Current [A]
Current [A]
Motor torque [Nm]
Output Torque [Nm]
Torque [Nm]
Motor Speed [RPM]
Output Speed [RPM]
Speed [RPM]
Time [s]

Computational Bionics | Exercise 1: Modeling of the angle sensor

```python
import numpy as np
from numpy import sin
import pandas as pd
import matplotlib.pyplot as plt
import math
```





Modeling of the sensor

This formula was derived from the Wheatstone bridge

$$U_{out} = U_{in}\left(1 - \frac{\alpha_i}{\alpha_{max}} - \frac{R_2}{R_1 + R_2}\right)$$

This means that for a bigger angle α_i the smaller the $ U_{out} value is gonna be.

By looking at the gait data $ \alpha_{i} $ can be decided for the angles $ \alpha_{1} $ and $ \alpha_{2} $

Max Angle	184,925925
Min Angle	119,773408
Angle Range	65,1525162

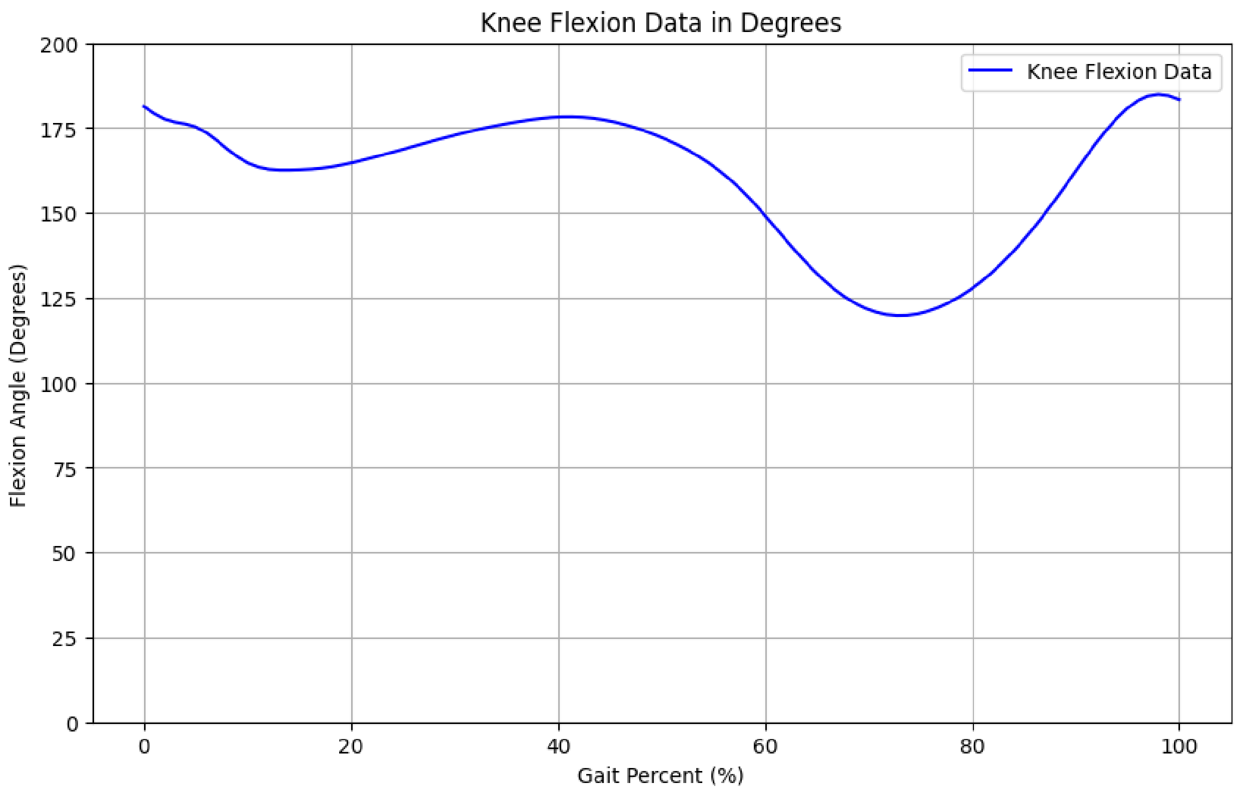
This gives an effective angle shange of about 65 degrees for the gait cycle and $ \alpha_{1} $ and $ \alpha_{2} $ were therefore decided to be

$$\alpha_1 \approx 119$$

$$\alpha_2 \approx 185$$

```
#File and Data Configuration
file_path = 'gait_data.xls'
sheet_name = 'Tabelle1'
column_index = 6
start_row = 1
end_row = 101
desired_max_voltage = 3

# Data Loading and Preprocessing
df = pd.read_excel(file_path, sheet_name=sheet_name)  # Load the Excel file
knee_flexion_data = df.iloc[start_row:end_row, column_index].to_numpy()  # Extract data
gait_percent = np.linspace(0, 100, len(knee_flexion_data))

# Plotting the angles given from the gait data excel sheet
plt.figure(figsize=(10, 6))
plt.plot(gait_percent, knee_flexion_data, label='Knee Flexion Data', color='b')
plt.title('Knee Flexion Data in Degrees')
plt.xlabel('Gait Percent (%)')  # Updated x-axis label
plt.ylabel('Flexion Angle (Degrees)')
plt.ylim(0, 200)
plt.grid(True)
```

```
plt.legend()

# Show the plot
plt.show()
```



Furthermore the resistor values in the Wheatstone bridge need to be decided. Since we know that the bigger the angle α_i is, the smaller the $ U_{out} $value is gonna be.

$$U_{out} = U_{in}\left(1 - \frac{\alpha_2}{\alpha_{max}} - \frac{R_2}{R_1 + R_2}\right) = 0$$

$$\frac{R_2}{R_1 + R_2} = 1 - \frac{\alpha_2}{\alpha_{max}} = \frac{270 - 185}{270}$$

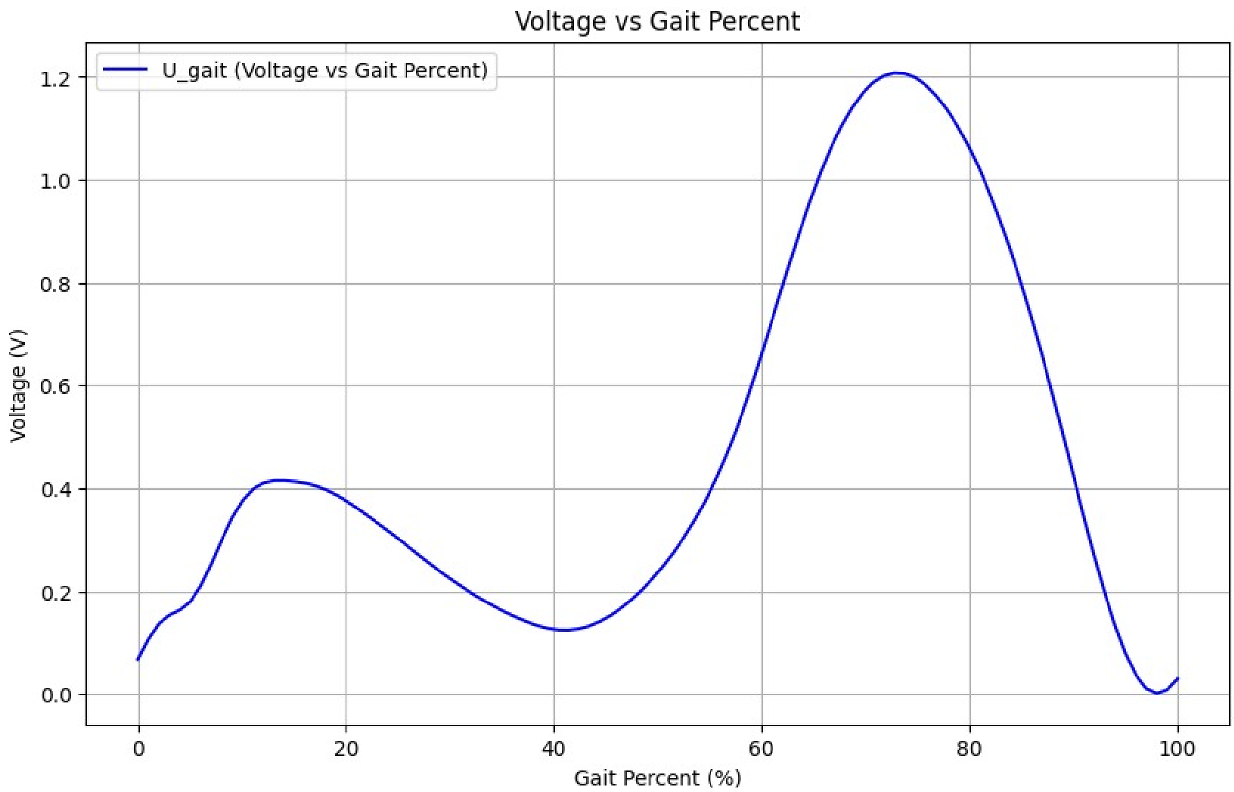
Which gives $R_1 = 18.5\,k\,\Omega, R_2 = 8.5\,k\,\Omega$

This makes the output for the angle $ \alpha_{1} = 119 $

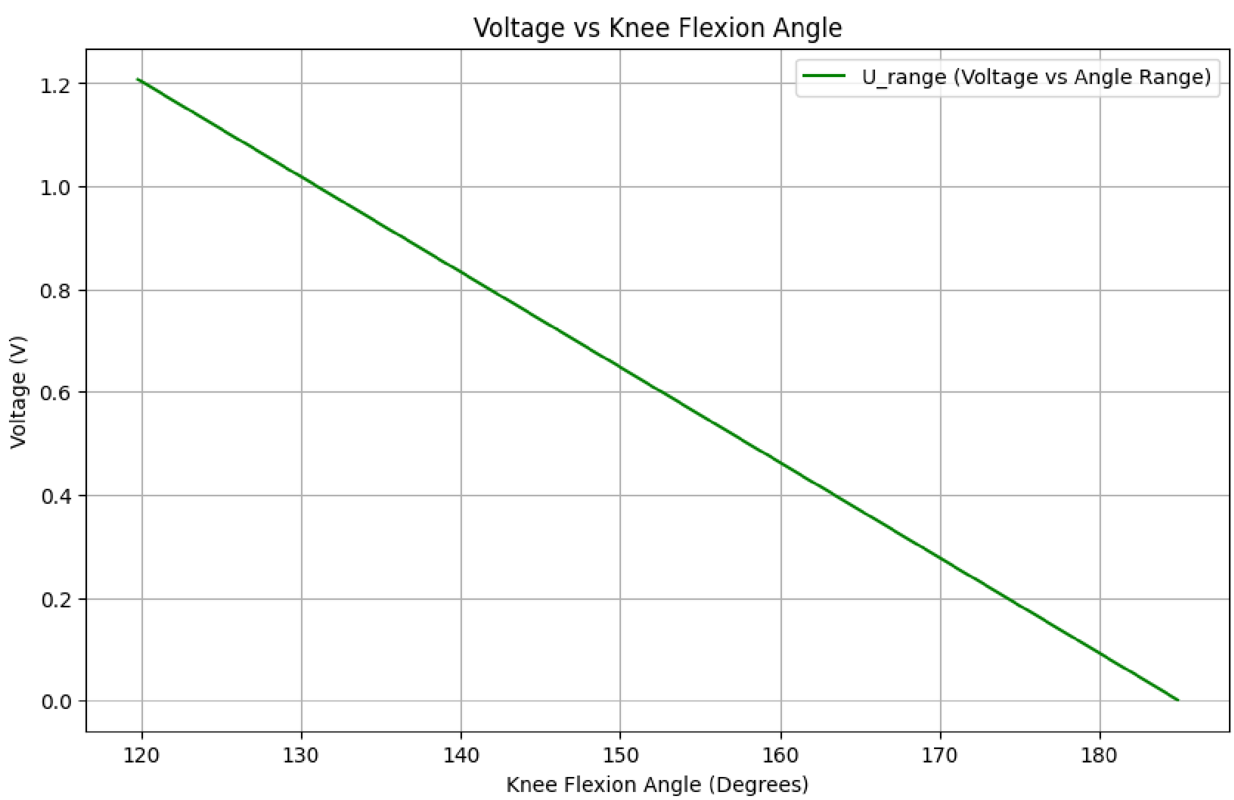
$$U_{out} = U_{in}¿$$

```
U_in = 5  #V
R1 = 18500 #Ohm
R2 = 8500 #Ohm

#Voltage Calculation
```

```
#The calculated output voltage using the values from the knee flexion
data.
U_gait = U_in * (1 - R2 / (R1 + R2) - knee_flexion_data / 270)

#The calculated output voltage using the range of values that the
kneee is in during the gait
angle_min = np.min(knee_flexion_data)
angle_max = np.max(knee_flexion_data)
angles_range = np.linspace(angle_min, angle_max, 100)  # Angle range
U_range = U_in * (1 - 8500 / 27000 - angles_range / 270)  # Ideal
voltage range


# Plot 1: U_gait (Voltage vs Gait Percent)
plt.figure(figsize=(10, 6))
plt.plot(gait_percent, U_gait, label='U_gait (Voltage vs Gait
Percent)', color='blue')
plt.title('Voltage vs Gait Percent')
plt.xlabel('Gait Percent (%)')
plt.ylabel('Voltage (V)')
plt.grid(True)
plt.legend()
plt.show()

# Plot 2: U_range (Voltage vs Angle Range)
plt.figure(figsize=(10, 6))
plt.plot(angles_range, U_range, label='U_range (Voltage vs Angle
Range)', color='green')
plt.title('Voltage vs Knee Flexion Angle')
plt.xlabel('Knee Flexion Angle (Degrees)')
plt.ylabel('Voltage (V)')
plt.grid(True)
plt.legend()
plt.show()
```



Voltage vs Gait Percent
U_gait (Voltage vs Gait Percent)
Voltage (V)
Gait Percent (%)
0.0
0.2
0.4
0.6
0.8
1.0
1.2
0
20
40
60
80
100



Voltage vs Knee Flexion Angle
U_range (Voltage vs Angle Range)
Voltage (V)
Knee Flexion Angle (Degrees)
0.0
0.2
0.4
0.6
0.8
1.0
1.2
120
130
140
150
160
170
180

The Wheatstone bridge gives a voltage significantly lower than what the ADC can input (3 V). We want to amplify the Wheatstone-sensor output so that it reaches closer to 3 V, providing a more detailed knee position.

The amplifying factor is calculated as:

$$k_{\text{amplifying factor}} = \frac{U_{\text{out, max}}}{U_{\text{ADC, in}}} \approx \frac{R4}{R3} = 2.7$$

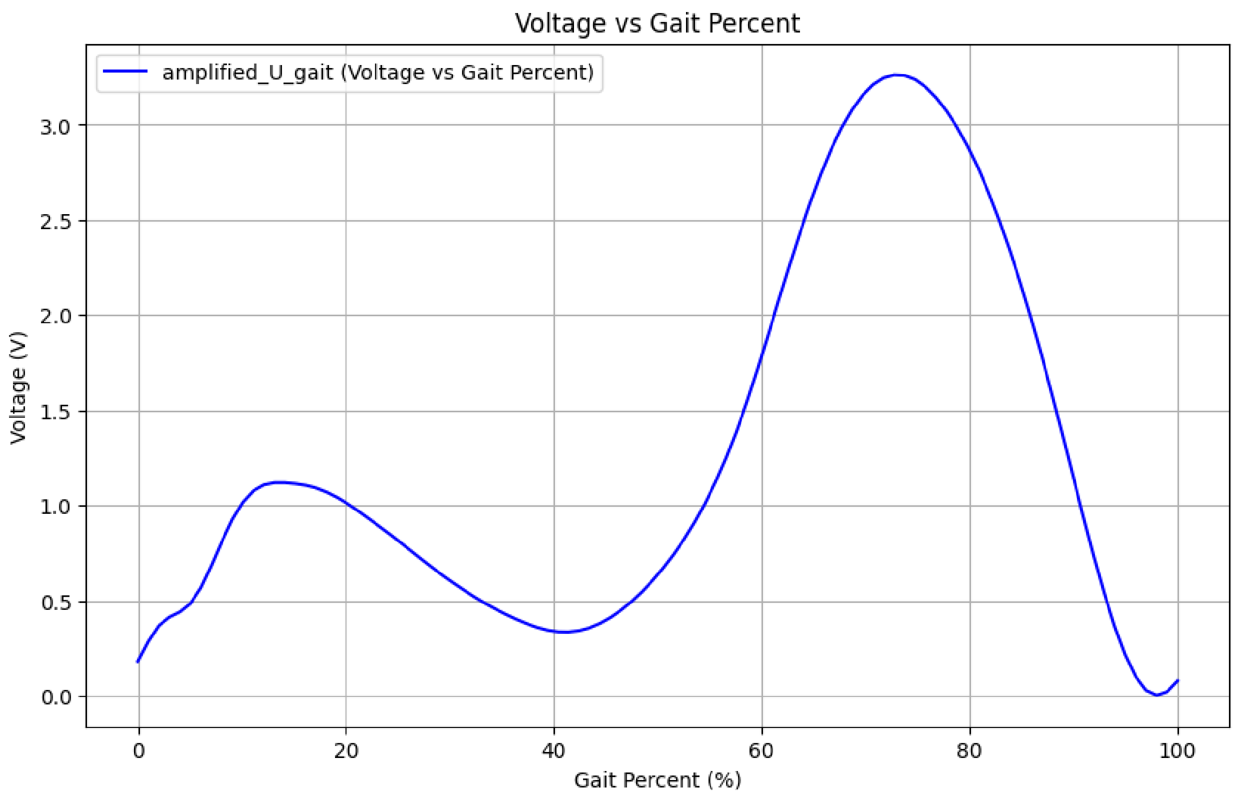
By using a non-inverting amplifier, a broader voltage range can be achieved:

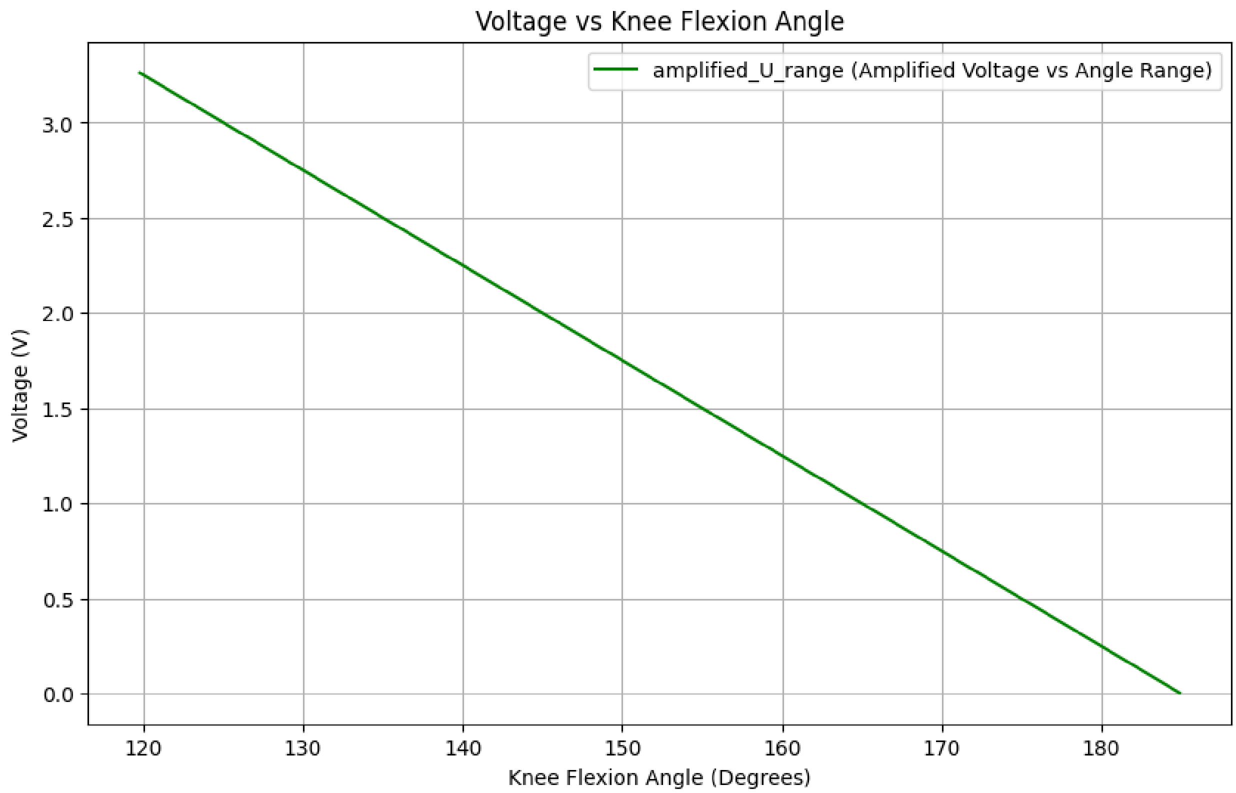
$$U_{out,OP-amp} = U_{in} * k_{amplifyingfactor} \approx \frac{U_{in} * R_4}{R_3} = U_{in} * 2.7$$

```
ADC_max_voltage = 3  # Maximum voltage range for ADC

# Amplification factor calculation
current_max_voltage = np.max(U_range)
amplification_factor = ADC_max_voltage / current_max_voltage

R3 = 10000.0 #Ohm
R4 = R3*amplification_factor/0.9 #The division by 0.9 is because of
the filter step that is yet to come.
R4 = math.floor(R4/ 1000) * 1000

#The 0.9 in the denominator is because the
amplification_factor = R4/(R3)

amplified_U_range = amplification_factor * U_range
amplified_U_gait = amplification_factor * U_gait


print('R3 is', R3 , "Ohm")
print('R4 is', R4 , "Ohm")
print('amplification factor is', amplification_factor)


# Plot 1: U_gait (Amplfied Voltage vs Gait Percent)
plt.figure(figsize=(10, 6))
plt.plot(gait_percent, amplified_U_gait, label='amplified_U_gait
(Voltage vs Gait Percent)', color='blue')
plt.title('Voltage vs Gait Percent')
plt.xlabel('Gait Percent (%)')
plt.ylabel('Voltage (V)')
plt.grid(True)
plt.legend()
plt.show()

# Plot 2: U_range (Amplified Voltage vs Angle Range)
plt.figure(figsize=(10, 6))
```

```
plt.plot(angles_range, amplified_U_range, label='amplified_U_range
(Amplified Voltage vs Angle Range)', color='green')
plt.title('Voltage vs Knee Flexion Angle')
plt.xlabel('Knee Flexion Angle (Degrees)')
plt.ylabel('Voltage (V)')
plt.grid(True)
plt.legend()
plt.show()
```

```
R3 is 10000.0 Ohm
R4 is 27000 Ohm
amplification factor is 2.7
```

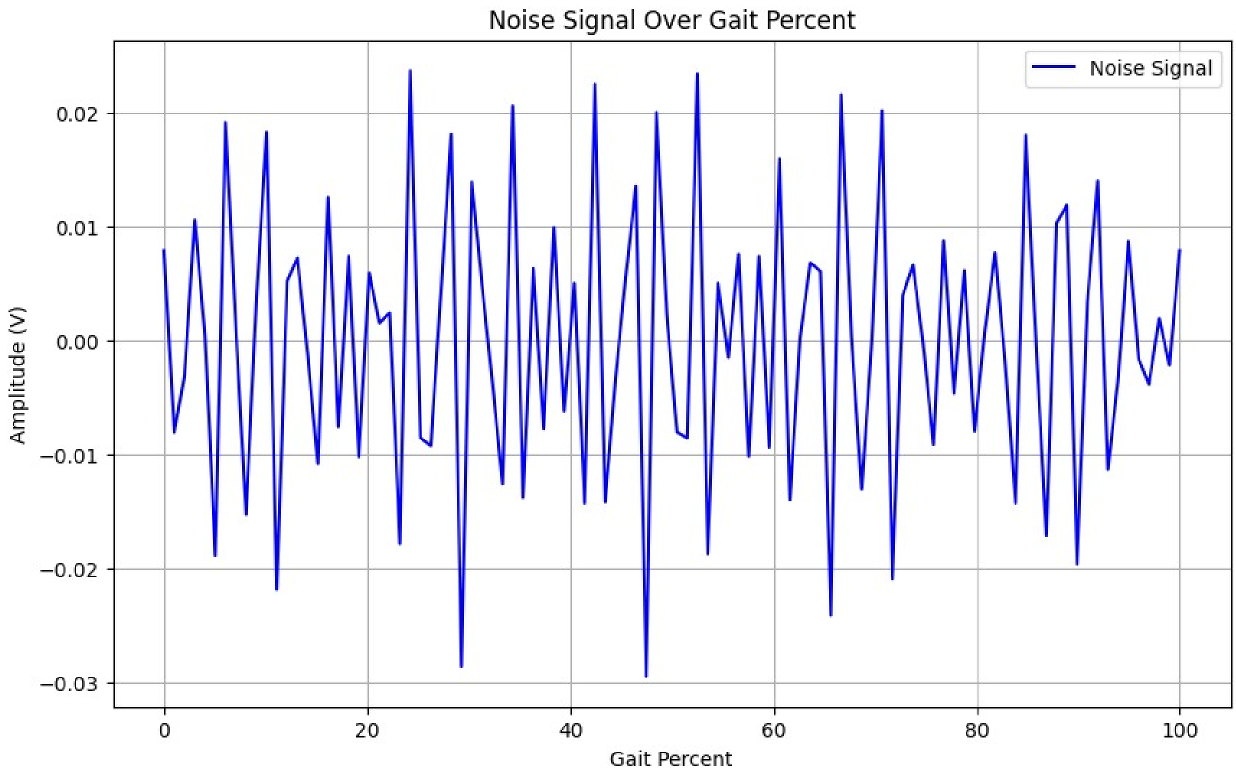


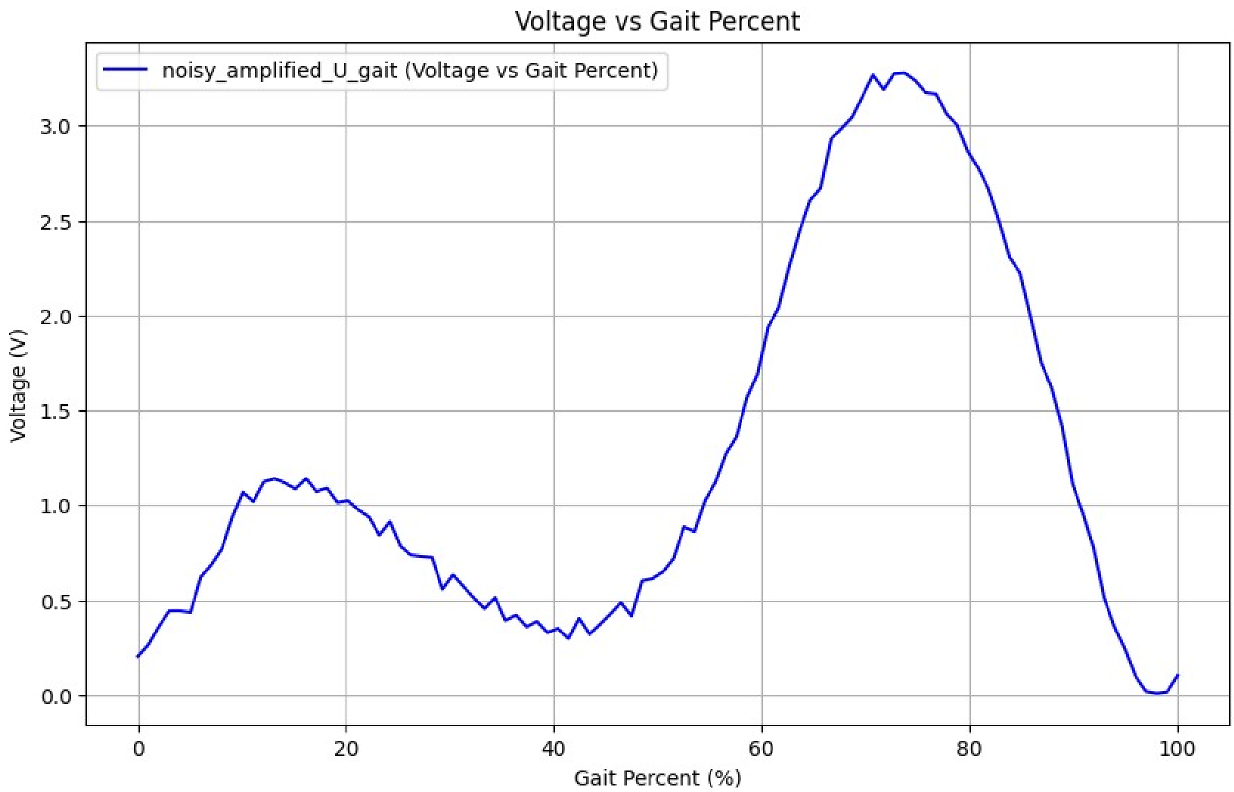


By introducing some noise to the model before the amplifier, like there is in realy life, the ADC will not get very precise values. Accounting for that we know the frequency of the noise, we can introduce a passive low pass filter to reduce the high frequency noise.

```
#Introducing the noise to the model somewhere above in form of a 50-,
66- and 170 Hz signal with each an amplitude of 0.01 V each.

#The 50Hz signal
noise50 = 0.01 * sin(50 * gait_percent * 2 * np.pi / 100 + 0.23)
noise66 = 0.01 * sin(66 * gait_percent * 2 * np.pi / 100 + 2.54)
noise170 = 0.01 * sin(170 * gait_percent * 2 * np.pi / 100)

noise = noise50 + noise66 + noise170

#Applying noise to output and
noisy_amplified_U_gait = (U_gait + noise) * amplification_factor
noisy_amplified_U_range = (U_range + noise) * amplification_factor

# Plotting the noise signal
plt.figure(figsize=(10, 6))
plt.plot(gait_percent, noise, label='Noise Signal', color='b')
plt.title('Noise Signal Over Gait Percent')
plt.xlabel('Gait Percent')
plt.ylabel('Amplitude (V)')
plt.grid(True)
plt.legend()
```

```
plt.show()


#Noise applied to the amplified plot for gait
plt.figure(figsize=(10, 6))
plt.plot(gait_percent, noisy_amplified_U_gait,
label='noisy_amplified_U_gait (Voltage vs Gait Percent)',
color='blue')
plt.title('Voltage vs Gait Percent')
plt.xlabel('Gait Percent (%)')
plt.ylabel('Voltage (V)')
plt.grid(True)
plt.legend()
plt.show()
```





The average number steps per minute is between 94-105 (see link)

Average steps per minute

$$f = \frac{60}{105/2(legs)} \approx 1\,Hz$$

The cutoff frequency for the filter in order to not disturub the given signal should be higher than the frequency for the knee, $f_{cutoff} = 2\,Hz$. The resistance and capacitive values are decided by the relationship below. $C=1 \micro F $

$$R_5 = \frac{1}{2\pi \cdot f_{cutoff} \cdot C} = \frac{1}{2\pi 2\, 10^{-6}} = 79577.47154 \approx 80\,k\Omega$$

The output voltage from a passive low pass filter after the voltage gain shown below is:

$$H(f) = \frac{U_{in}}{U_{out}} = \frac{1}{\sqrt{1+(2\pi f RC)^2}}$$

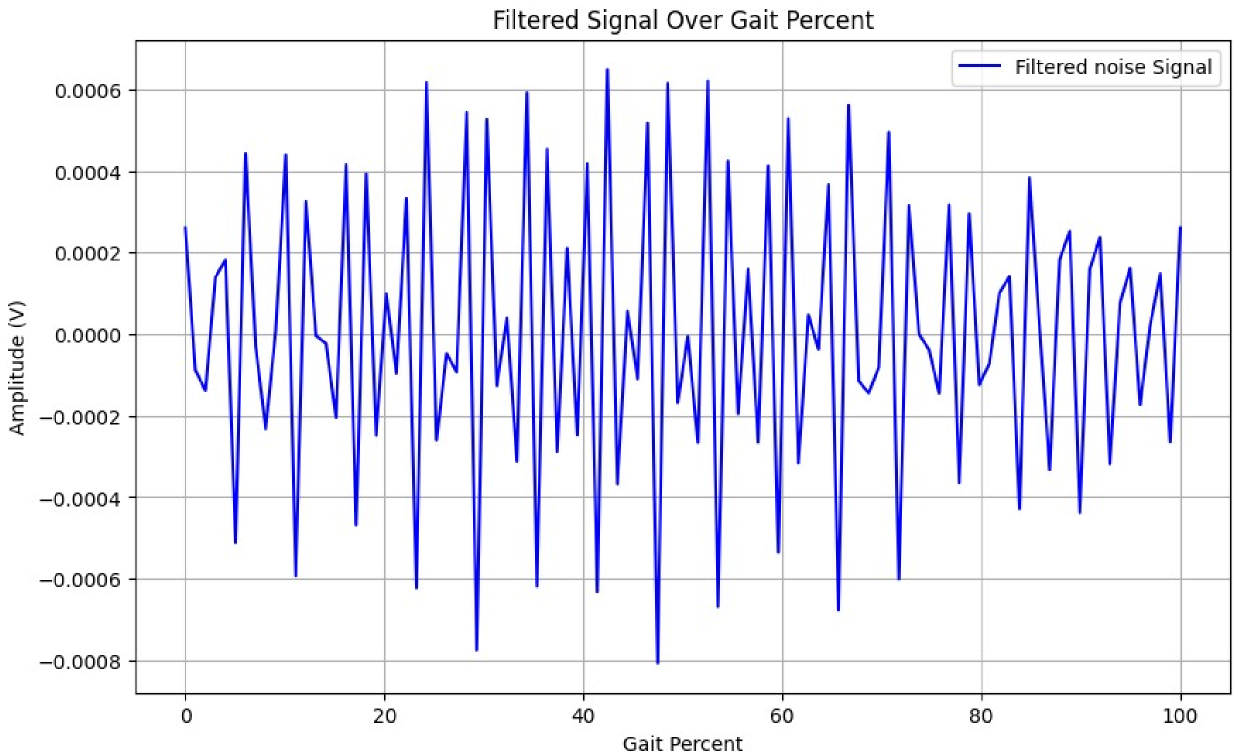
$$U_{out} = U_{in} * H(f)$$

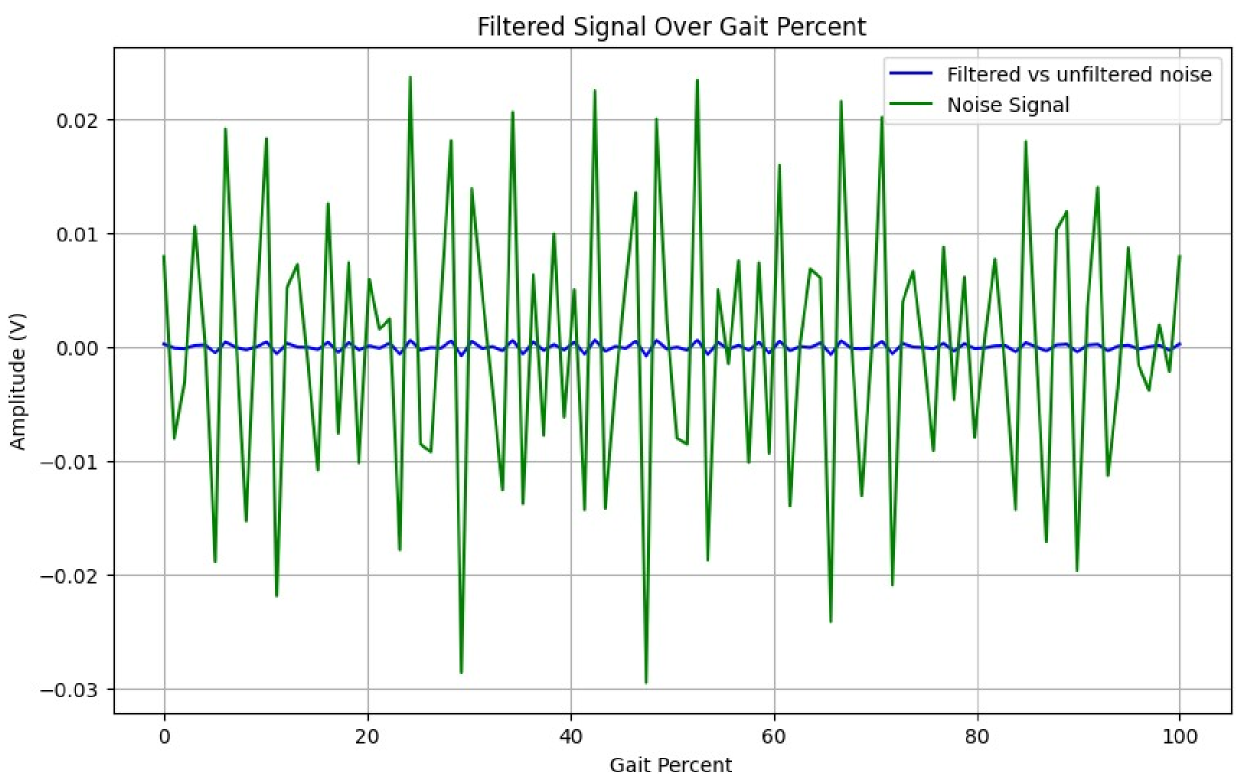
```
def H(f):
    """
    Calculate the transfer function H(f).
    Parameters:
```

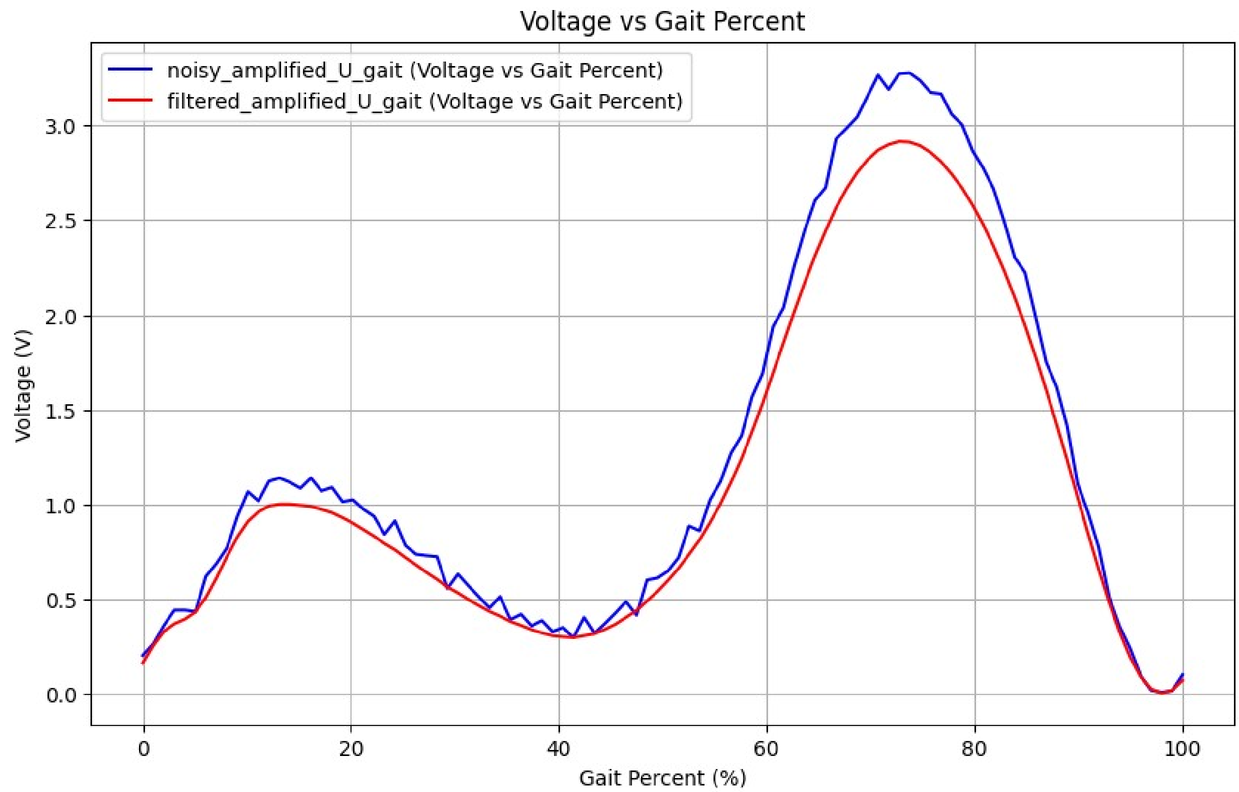
```
    f : float or array-like
        Frequency in Hz.
    Returns:
    float or array-like
        The transfer function H(f).
    """
    R=80000
    C=1*10**-6
    return 1 / np.sqrt(1 + (2 * np.pi * f * R * C) ** 2)

#Noise
filtered50 = H(50)*noise50
filtered66 = H(66)*noise66
filtered170 = H(170)*noise170

filtered_noise = filtered50 + filtered66 + filtered170

print(H(1))

filtered_amplified_U_gait = (H(1)*U_gait + filtered_noise) *
amplification_factor
filtered_amplified_U_range = (H(1)*U_range + filtered_noise) *
amplification_factor


# Plotting the filtered signal
plt.figure(figsize=(10, 6))
plt.plot(gait_percent, filtered_noise, label='Filtered noise Signal',
color='b')
plt.title('Filtered Signal Over Gait Percent')
plt.xlabel('Gait Percent')
plt.ylabel('Amplitude (V)')
plt.grid(True)
plt.legend()
plt.show()

# Plotting the noise signal compared to the filtered one
plt.figure(figsize=(10, 6))
plt.plot(gait_percent, filtered_noise, label='Filtered vs unfiltered
noise', color='b')
plt.plot(gait_percent, noise, label='Noise Signal', color='g')
plt.title('Filtered Signal Over Gait Percent')
plt.xlabel('Gait Percent')
plt.ylabel('Amplitude (V)')
plt.grid(True)
plt.legend()
plt.show()


#Filtered and unfiltered graphs in th gait plot
```

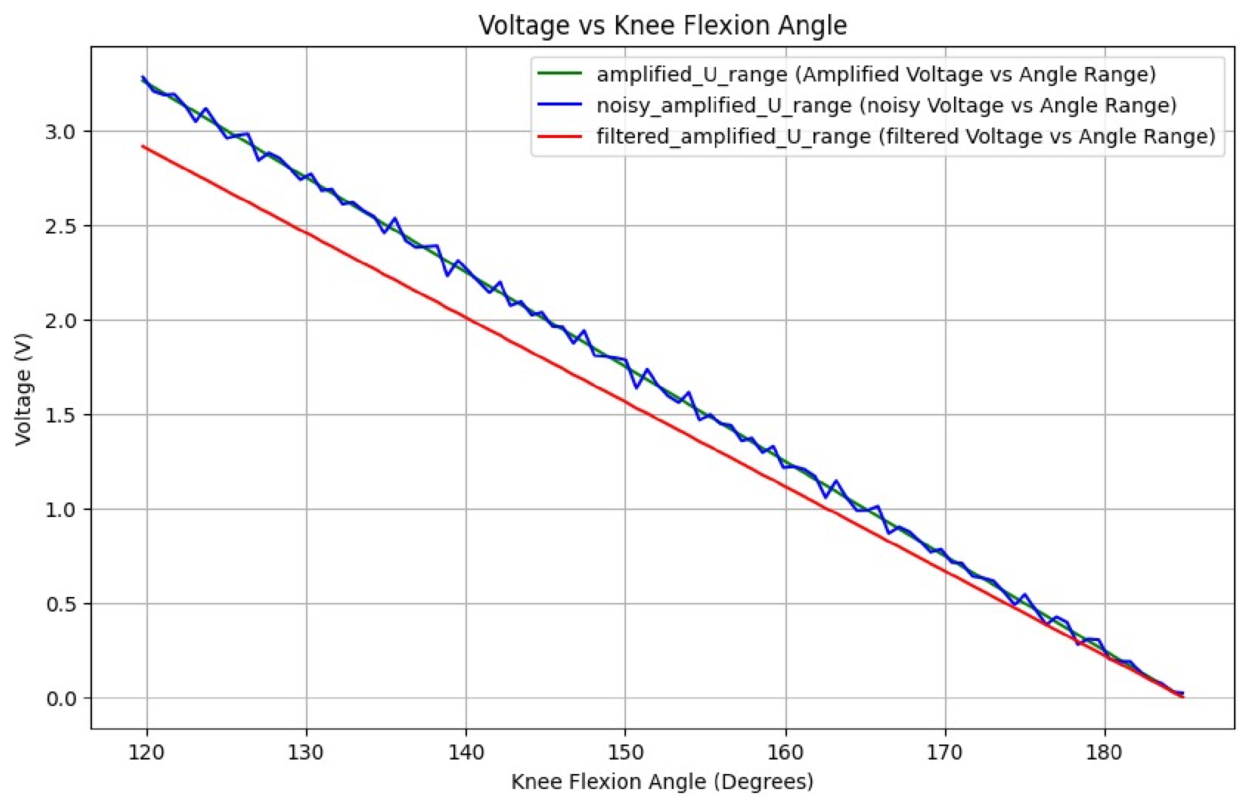
```
plt.figure(figsize=(10, 6))
plt.plot(gait_percent, noisy_amplified_U_gait,
label='noisy_amplified_U_gait (Voltage vs Gait Percent)',
color='blue')
plt.plot(gait_percent, filtered_amplified_U_gait,
label='filtered_amplified_U_gait (Voltage vs Gait Percent)',
color='red')
plt.title('Voltage vs Gait Percent')
plt.xlabel('Gait Percent (%)')
plt.ylabel('Voltage (V)')
plt.grid(True)
plt.legend()
plt.show()


#Unnoise and noise graphs in the angle range plot
plt.figure(figsize=(10, 6))
plt.plot(angles_range, amplified_U_range, label='amplified_U_range
(Amplified Voltage vs Angle Range)', color='green')
plt.plot(angles_range, noisy_amplified_U_range,
label='noisy_amplified_U_range (noisy Voltage vs Angle Range)',
color='blue')
plt.plot(angles_range, filtered_amplified_U_range,
label='filtered_amplified_U_range (filtered Voltage vs Angle Range)',
color='red')
plt.title('Voltage vs Knee Flexion Angle')
plt.xlabel('Knee Flexion Angle (Degrees)')
plt.ylabel('Voltage (V)')
plt.grid(True)
plt.legend()
plt.show()
```

```
0.8934763687940478
```


Filtered Signal Over Gait Percent
Filtered noise Signal
Amplitude (V)
Gait Percent
0.0006
0.0004
0.0002
0.0000
−0.0002
−0.0004
−0.0006
−0.0008
0
20
40
60
80
100


Filtered Signal Over Gait Percent
Filtered vs unfiltered noise
Noise Signal
Amplitude (V)
Gait Percent
0.02
0.01
0.00
−0.01
−0.02
−0.03
0
20
40
60
80
100


Voltage vs Gait Percent
noisy_amplified_U_gait (Voltage vs Gait Percent)
filtered_amplified_U_gait (Voltage vs Gait Percent)
Voltage (V)
Gait Percent (%)
0.0
0.5
1.0
1.5
2.0
2.5
3.0
0
20
40
60
80
100


Voltage vs Knee Flexion Angle
amplified_U_range (Amplified Voltage vs Angle Range)
noisy_amplified_U_range (noisy Voltage vs Angle Range)
filtered_amplified_U_range (filtered Voltage vs Angle Range)
Voltage (V)
Knee Flexion Angle (Degrees)
0.0
0.5
1.0
1.5
2.0
2.5
3.0
120
130
140
150
160
170
180

In order to be able to process the data from the angle sensor the analog output needs to be quantized to a 8 bit digital signal.

First we scale the voltage from a range of $ 0 > U > 3V$ to one of $ 0 > U > 255V$

$$U_{scaled} = U_{in} * 255/3$$

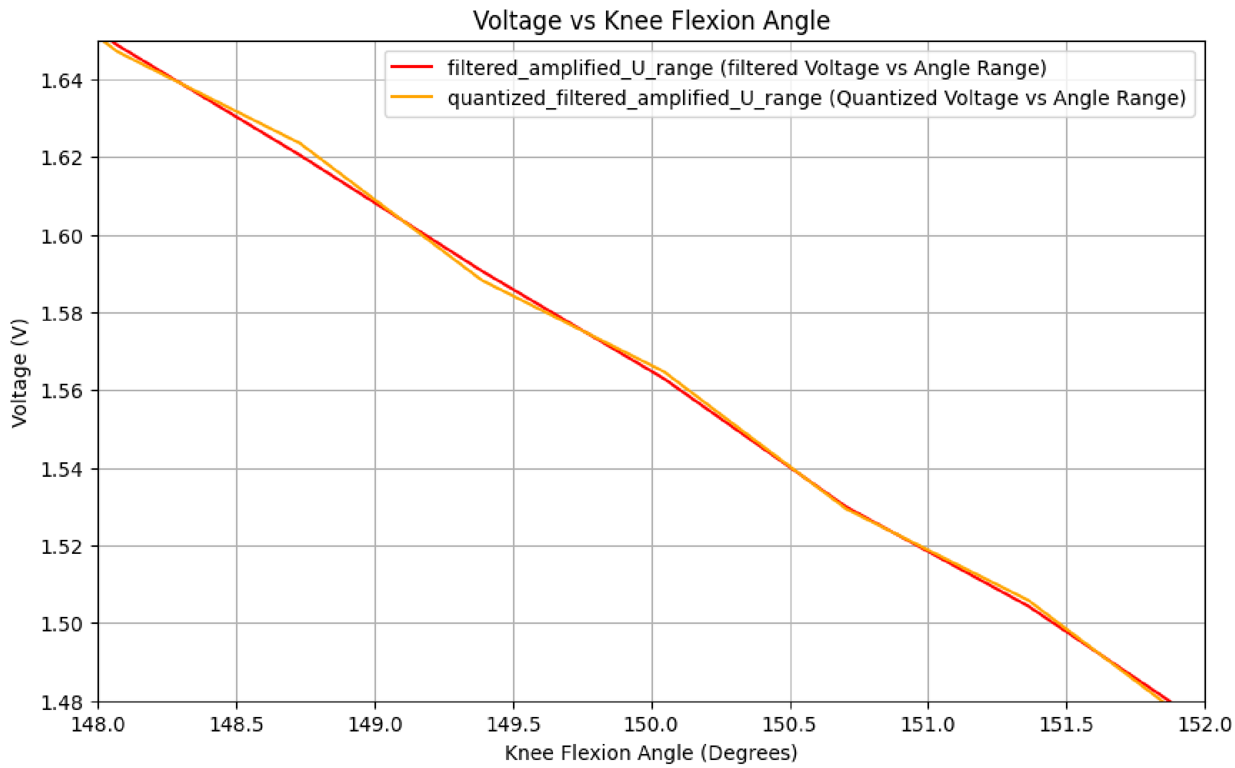
Then we round it to the nearest value digital voltage.

$$U_{rounded} = round(U_{scaled})$$

Finally we scale it down again to the original range and we have a quentized value. $ 0 > U > 3V$

$$U_{quantized} = U_{rounded} * 3/255$$

```
U_scaled = filtered_amplified_U_range * (255 / 3)  # Step 1: Scale input voltage to 8-bit range (0-255)
U_rounded = np.round(U_scaled)  # Step 2: Round to the nearest integer (quantize)
quantized_filtered_amplified_U_range = U_rounded * (3 / 255)  # Step 3: Scale back to original range (0V to 3V)

#Quantized and the filtered values in the same plot
plt.figure(figsize=(10, 6))
plt.plot(angles_range, filtered_amplified_U_range, label='filtered_amplified_U_range (filtered Voltage vs Angle Range)', color='red')
plt.plot(angles_range, quantized_filtered_amplified_U_range, label='quantized_filtered_amplified_U_range (Quantized Voltage vs Angle Range)', color='orange')
plt.title('Voltage vs Knee Flexion Angle')
plt.xlabel('Knee Flexion Angle (Degrees)')
plt.ylabel('Voltage (V)')
plt.grid(True)
plt.ylim(1.48 , 1.65)  # Set x-axis limits
plt.xlim(148, 152)  # Set x-axis limits
plt.grid(True)
plt.legend()
plt.show()
```


Voltage vs Knee Flexion Angle
filtered_amplified_U_range (filtered Voltage vs Angle Range)
quantized_filtered_amplified_U_range (Quantized Voltage vs Angle Range)
Voltage (V)
1.64
1.62
1.60
1.58
1.56
1.54
1.52
1.50
1.48
148.0
148.5
149.0
149.5
150.0
150.5
151.0
151.5
152.0
Knee Flexion Angle (Degrees)