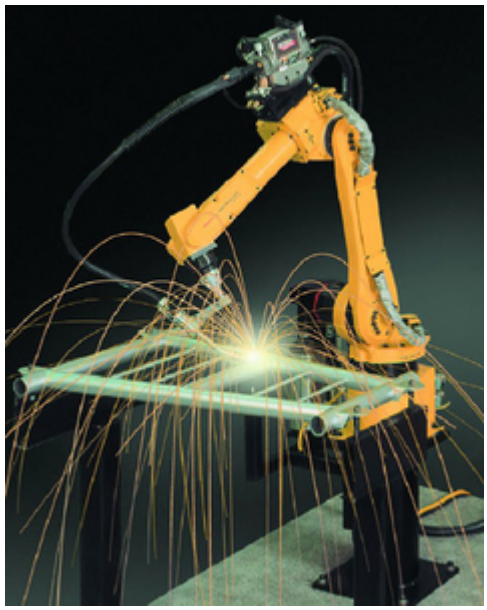
Solving Inverse Kinematics using Gauss-Newton

10 points

Robots are controlled by adjusting *joint variables*. For example, one can adjust the angles of rotational joints or the lengths of translational joints. "Forward kinematics" refers to determining the position and orientation of the robot's end effector (e.g. gripper, welding attachment) from the joint variables. "Inverse kinematics" refers to determining values of the joint variables that achieve a given position and orientation of the end effector. This problem arises whenever one needs a robot to move to a certain position and orientation, e.g. in automobile manufacturing when a robotic arm needs to make cuts or welds at specified positions.



(Photo credit: Wikimedia commons for FANUC Robotics Deutschland (https://commons.wikimedia.org/wiki/File:Arc-welding.jpg))

Position and orientation are represented as a 4×4 matrix, using homogeneous coordinates. Such a 4×4 matrix generically has the following form:

$$T = \begin{bmatrix} U & \boldsymbol{x} \\ \mathbf{0}^T & 1 \end{bmatrix},$$

where $\boldsymbol{x}$ is the position of the end effector, and U is an orthogonal matrix that describes the coordinate transformation between the lab coordinate system and a coordinate system aligned with the end effector. In this problem, we will compute forward and inverse kinematics for arbitrary robots. The way the links relate to each other geometrically is described using the Denavit-Hertenberg convention.

This is a commonly used convention for selecting frames of reference in robotics. In this convention, coordinate frames are attached to the joints between two linking segments of a robot's arm, such that one transformation is associated with the joint, $[Z]$, and the second with the link $[X]$. A robot arm consisting of 3 links, for example, would be written as:

$$[T] = [Z_1][X_1][Z_2][X_2][Z_3][X_3],$$

where $[T]$ represents the transformation locating the end.

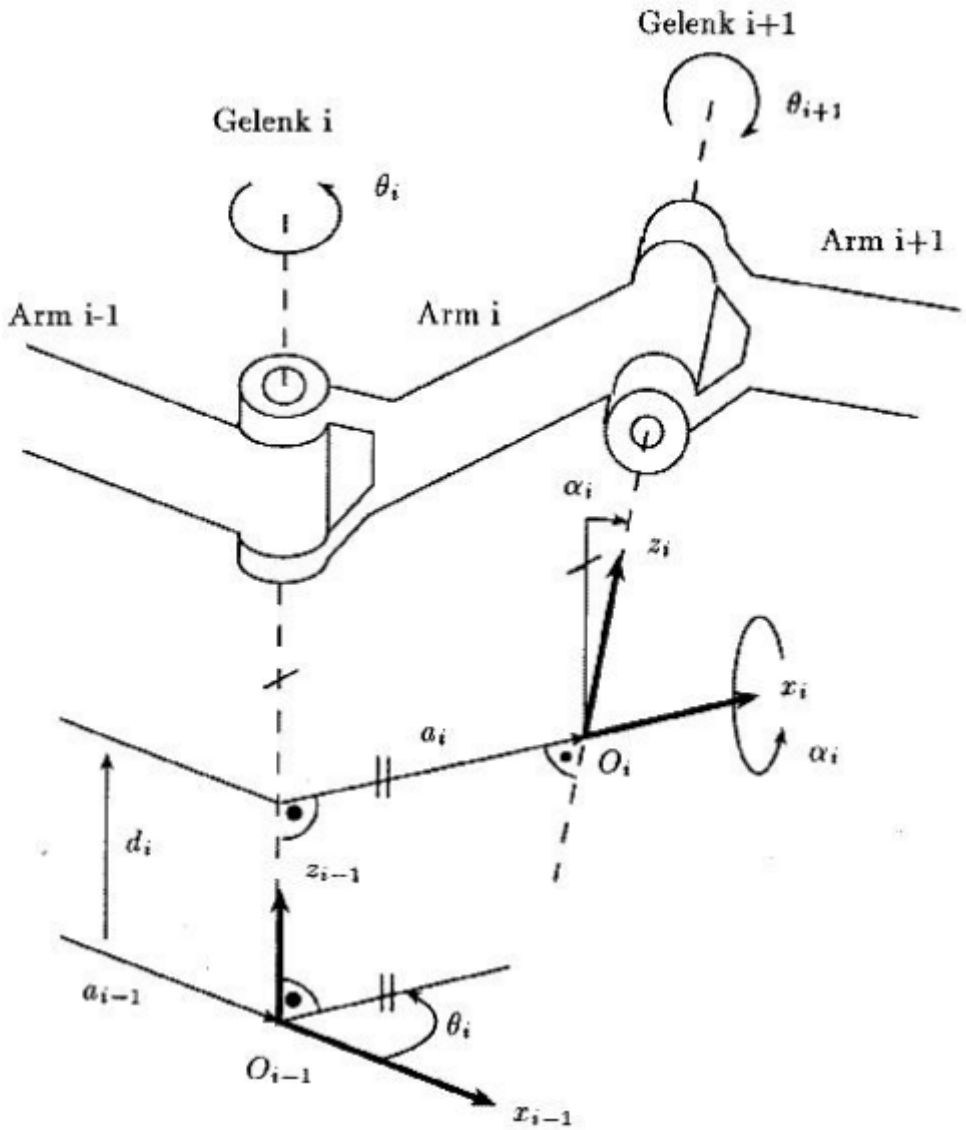
Each of these transformations can be written furthermore as a translational part and a rotational part, so that

$$[Z_i] = \mathrm{Trans}_{Z_i}(d_i)\,\mathrm{Rot}_{Z_i}(\theta_i),$$

and

$$[X_i] = \mathrm{Trans}_{X_i}(r_{i,i+1})\,\mathrm{Rot}_{X_i}(\alpha_{i,i+1}).$$

Using this same notation, each link can be described from the concurrent coordinate system to the previous one:



A visualization of the Denavit-Hartenberg Convention. Translation: "Gelenk" -> joint, "Arm" -> link. Figure credit: Jianwei Zhang, Lecture notes for 'Introduction to Robotics' (https://tams.informatik.uni-hamburg.de/lehre/2014ss/vorlesung/itr/slides/Vorlesung.pdf#page=68), Universität Hamburg, Summer Semester 2014

$$^{n-1}T_n = \mathrm{Trans}_{z_{n-1}}(d_n) \cdot \mathrm{Rot}_{z_{n-1}}(\theta_n) \cdot \mathrm{Trans}_{x_n}(r_n) \cdot \mathrm{Rot}_{x_n}(\alpha_n).$$

Here, each matrix can be written as:

$$\mathrm{Trans}_{z_{n-1}}(d_n) = \left[\begin{array}{ccc|c} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d_n \\ \hline 0 & 0 & 0 & 1 \end{array}\right]$$

$$\mathrm{Rot}_{z_{n-1}}(\theta_n) = \left[\begin{array}{ccc|c} \cos\theta_n & -\sin\theta_n & 0 & 0 \\ \sin\theta_n & \cos\theta_n & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \hline 0 & 0 & 0 & 1 \end{array}\right]$$

$$\text{Trans}_{x_n}(r_n) = \left[\begin{array}{ccc|c} 1 & 0 & 0 & r_n \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \hline 0 & 0 & 0 & 1 \end{array}\right]$$

$$\text{Rot}_{x_n}(\alpha_n) = \left[\begin{array}{ccc|c} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha_n & -\sin\alpha_n & 0 \\ 0 & \sin\alpha_n & \cos\alpha_n & 0 \\ \hline 0 & 0 & 0 & 1 \end{array}\right].$$

Using only the four D-H parameters, any sequence of links in a robotic arm (or "kinematic chain") can be described. They are:

- d: offset along previous z to the common normal
- θ: angle about previous z, from old x to new x
- r: length of the common normal. Assuming a revolute joint, this is the radius about the previous z.
- α: angle about common normal, from old z axis to new z axis.

Any of them can either be fixed or act as a joint variable. In this problem, a link is described by a `Link` class which represents the four D-H parameters. The `Link` constructor takes these as θ, d, α, and r. Some links chained together constitute a kinematic chain, i.e. a robotic arm.

We have supplied a plotting utility that you can use to view the configuration of your robot in three dimensions in `show_chain_views`. This function takes two arguments: the first corresponding to a Python list of `Link` objects associated with the chain and the second corresponding to a Python dictionary mapping Sympy variables to their respective values. If no Sympy variables are present in a list of `Link` objects, one may pass in the empty dictionary, e.g. `{}`.

For example, a Python list of links is created below:

```
import numpy as np
import sympy as sp
DEG_TO_RAD = np.pi/180
example_chain = [
    Link(
      angle=20*DEG_TO_RAD,
      offset=1.5,
      twist=30*DEG_TO_RAD,
      length=0.35,
    ),
    Link(
      angle=0*DEG_TO_RAD,
      offset=1,
      twist=15*DEG_TO_RAD,
      length=0,
    ),
    Link(
      angle=0,
      offset=1,
      twist=0*DEG_TO_RAD,
      length=0,
    ),
]
example_dof_chain = [
    Link(
      angle=20*DEG_TO_RAD,
      offset=1.5,
      twist=sp.Symbol("alpha_1"),
      length=0.35,
    ),
    Link(
      angle=0*DEG_TO_RAD,
      offset=1,
      twist=15*DEG_TO_RAD,
      length=0,
    ),
    Link(
      angle=0,
      offset=1,
      twist=sp.Symbol("alpha_3"),
      length=0,
    ),
]
show_chain_views(example_chain, {})
show_chain_views(example_dof_chain, {"alpha_1":30*DEG_TO_RAD,"alpha_3":15*DEG_TO_RA
D})
```

The `Link` class is provided here:

```
class Link:
    def __init__(self, angle, offset, twist, length):
        self.angle = angle
        self.offset = offset
        self.twist = twist
        self.length = length
```

In this problem, you are tasked with determining the appropriate angles, using the Gauss Newton optimization strategy discussed in class, in order to reach a desired goal point.

In order to apply Gauss-Newton optimization, derviatives of the residual function with respect to the goal point are needed. We will approach this problem by expressing the solution to the forward kinematics problem as matrix expression in the symbolic package `sympy`, and then taking (symbolic) derivatives of that to carry out Gauss-Newton. (Not only is this much easier and less error-prone than trying to carry out the derivatives by hand, doing so also will also give you experience on how to work with symbolic math in Python.)

SymPy Primer (click to view)

The following demo may be accessed on the course web page under "Additional Topics". This may also be accessed at the following link:

Sympy Demonstration Notebook (https://mybinder.org/v2/gh/illinois-scicomp/cs450-s19-binder/master?filepath=misc/An%20Introduction%20to%20Sympy.ipynb)

What do I need to do? (click to view)

In this problem, you are given a list of Sympy variables (`variables`) that need to be solved for such that the robot's end-effector is at the position in `position`. A description of the robot's end-effector is found in `links`, which is a list of `Link` objects. The position of a robot's end-effector is found in the first three rows of the last column of $[T]$, which should be the result of your implementation of `chain_matrix`.

Determine the value needed for each variable in `variables` so that the robot's end-effector is at the position specified in `position`.

Use Gauss-Newton as the optimization method. Write your outputs to `angles` in the same order as `variables`.

In order to verify your `chain_matrix` function, please save the result into `chain_matrix_results` using the `reference_links` provided, e.g.

```
chain_matrix_results = chain_matrix(reference_links)
```

after an implementation of `chain_matrix` has been specified.

All angles are provided in radians. In addition, your solution `angles` should also be in radians.

Please compute the angles with enough precision such that the norm of the residual is less than `1e-6`.

You may not use any functions from `scipy.optimize`.

INPUT:

- `show_chain_views` : function that takes a list of `Link` objects and the value of each degree of freedom (DOF) (in a python dictionary mapping strings/Sympy variables to their respective values). This outputs an imagine of the manipulators in 3D.
- `links` : a list of Links, each with some sympy Symbol DOF associated with the respective link's α or θ value.

- `position` : numpy array specifying the first three components of the position to which the last arm points. In particular, this will be the first three coordinates of the last column of the output of the `chain_matrix` function.
- `reference_links` : list of `Link` objects to be used for reference in checking the result of $[T]$ computation (no Sympy variables here!).
- `variables` : Python list of sympy `Symbol` objects corresponding to the DOFs in `links` .

OUTPUT:

- `angles` : numpy array containing the values of the DOFs as listed in `variables` . Note that, for grading purposes, these must be in the same order as they appear in `variables` .
- `chain_matrix_results` : results from a function which, given a list of `Link` objects with `Sympy` variables substituted, returns the output $[T]$, the transformation locating the end link position. The first three coordinates of the last column of the output represent the coordinates of the end arm. This should be a numpy array. These results should be based on the `reference_links` .

Starter code (click to view)

Answer*

```
    for link in links:
        Trd = sp.eye(4)#offset
        Rttheta = sp.eye(4)#angle
        Trr = sp.eye(4)#length
        Rtalpha = sp.eye(4)#twist
        Trd[2,3] = link.offset
        Rttheta[0,0] = Rttheta[1,1] = sp.cos(link.angle)
        Rttheta[1,0] = sp.sin(link.angle)
        Rttheta[0,1] = -sp.sin(link.angle)
        Trr[0,3] = link.length
        Rtalpha[1,1] = Rtalpha[2,2] = sp.cos(link.twist)
        Rtalpha[2,1] = sp.sin(link.twist)
        Rtalpha[1,2] = -sp.sin(link.twist)
        Tn = Tn@Trd@Rttheta@Trr@Rtalpha
    return Tn

def getvaluedict(valuelist):
    values = {}
    for idx in range(2):
        values[variables[idx]] = valuelist[idx]
    return values
def guass_newton(Tn):
```

Press F9 to toggle full-screen mode. Set editor mode in user profile (/profile/).

Your answer is correct.

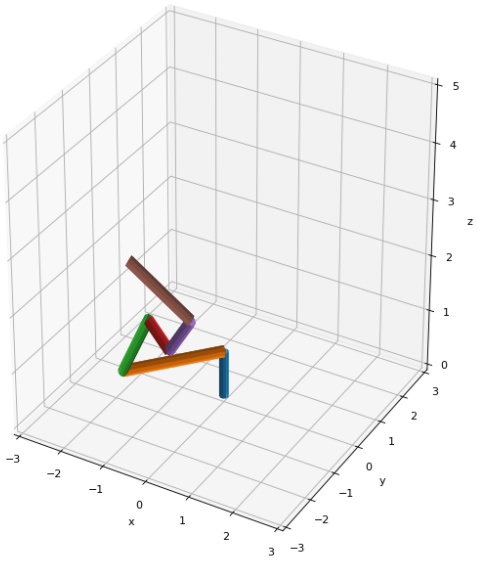
Here is some feedback on your code:

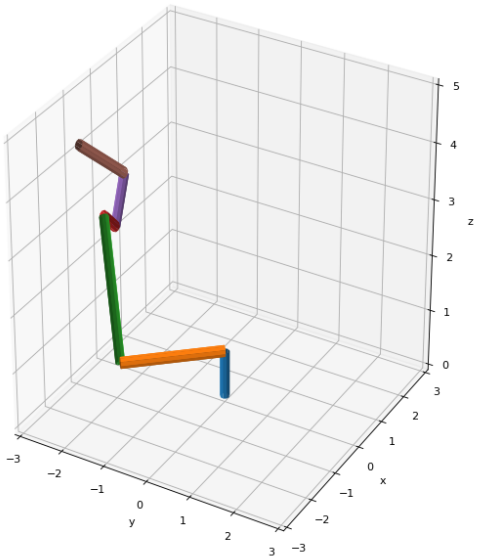
- 'chain_matrix_results' looks good
- angles provided found a valid solution

- Finished.
- Execution time: 3.6 s -- Time limit: 15.0 s

Your code ran on relate-01.cs.illinois.edu.

Your code produced the following plots:
Figure1





The following code is a valid answer:

```
import numpy as np
import numpy.linalg as la
import sympy as sp

import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

DEG_TO_RAD = sp.pi/180
def translation_mat(vec):
    result = sp.eye(4)
    result[:3, -1] = vec
    return result

def rotation_mat(i, j, angle):
    result = sp.eye(4)
    assert i < j
    result[i, i] = sp.cos(angle)
    result[j, j] = sp.cos(angle)
    result[i, j] = -sp.sin(angle)
    result[j, i] = sp.sin(angle)
    return result

def link_rot_mat(link):
    return rotation_mat(0, 1, link.angle)

def link_mat(link):
    return link_rot_mat(link) @ translation_mat([0, 0, link.offset])

def joint_rot_mat(link):
    return rotation_mat(1, 2, link.twist)

def joint_mat(link):
    return joint_rot_mat(link) @ translation_mat([link.length, 0 , 0])

def denavit_hartenberg_mat(link):
    return link_mat(link) @ joint_mat(link)

def chain_matrix(chain):
    result = sp.eye(4)
    for link in chain:
        result = result @ denavit_hartenberg_mat(link)
    return result

x=chain_matrix(links)[0:3,-1]
y=position
z=x
for i in range(3):
    z[i]=(x[i]-y[i])**2.
from sympy.utilities.lambdify import lambdify
array2mat = [{'ImmutableDenseMatrix':np.array}, 'numpy']
lam_f = lambdify(variables, z, modules=array2mat)
lam_J = lambdify(variables, z.jacobian(variables), modules=array2mat)
def gauss_newton(x0,lam_f=lam_f,lam_J=lam_J,tol=1e-8,max_iter=100):
    x=x0
    iter_count = 0
    res=la.norm(lam_f(*x))
    while iter_count < max_iter and res > tol:
```

```
        iter_count+=1
        # the use of this flatten seems like a smell to me, but not sure how to fix
        x=x+la.lstsq(lam_J(*x),-lam_f(*x).flatten(),rcond=None)[0]
        res=la.norm(lam_f(*x))
    return x
nVar = len(variables)
angles = gauss_newton(np.zeros(nVar))
chain_matrix_results = np.array(chain_matrix(reference_links)).astype(np.float64)
```