340W25 PA2

**Due date:** Check the schedule table.

**Submission:** Starter code is provided ([link](https://drive.google.com/file/d/1rVBK5BptpOZICPRYH8__gbx0P2qFwjWw/view?usp=drive_link)) to be filled in. A single **zip file** must be submitted through eClass. Use filename in the format: **pa2-[lastname].zip**. Replace [lastname].

**Total:** 100

# Question 1 (50)

## Objectives

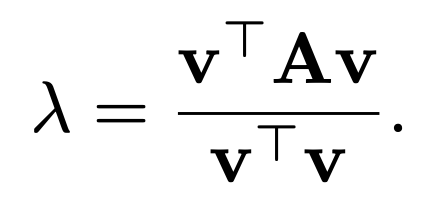
In this question, you will implement several methods for computing eigenvalues and eigenvectors, and compare them to MATLAB’s built-in functions.

## Description

Implement the following algorithms in the provided function templates, to compute eigenvalues and eigenvectors:

1. Normalized power iteration
2. Inverse iteration
   * You can use Matlab’s \ operator to solve the linear system.
3. QR iteration
   * You can use Matlab’s qr function for QR factorization.

Then complete the following tasks. Run main.m to check your work. Note the provided A.mat contains a 3x3 zero matrix. The file is a binary file, so you cannot edit it directly. Rather, use the save command from the MATLAB command line or write your own script.

1. Use a square matrix with at least three rows to show that your implemented normalized power iteration algorithm finds the dominant eigenvalue and the corresponding eigenvector.  
     
   Note that if **v** is an eigenvector of matrix **A**, then the corresponding eigenvalue can be found by the Rayleigh quotient:  
   
2. Use a square matrix with at least three rows to show that your implemented inverse iteration algorithm finds the eigenvalue with the smallest magnitude and the corresponding eigenvector.
3. Use a square matrix with at least three rows to show that your implemented QR iteration algorithm finds all the eigenvalues and the corresponding eigenvectors.
4. Validate your answers for parts A, B, and C by comparing it with the output of Matlab’s default function eig for calculating eigenvalues and eigenvectors.

It is perfectly fine to use the same matrix for all three above, but you can also use different matrices if you choose.

# Question 2 (50)

## Objectives

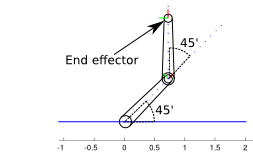
In this question, you will explore the use of non-linear equations on a problem that occurs in robotics/animation. In particular, you will evaluate Newton's and Broyden's methods for non-linear systems of equations in inverse kinematic applications.

## Background

Given a manipulator or a kinematic chain (a set of joints that are joined by links), **forward kinematics** is the problem of determining the position of the joints or only the end-effector, that is, the tip of the manipulator, given the joint angles. This problem is a straightforward application of Euclidean transformations.

**Inverse kinematics** is the inverse problem: determine the joint orientations in terms of angles that satisfy a given set of constraints (e.g., find the joint orientations that put the end-effector at a specific position). This problem is harder than forward kinematics; the analytic solution is often complicated or doesn't exist, so numerical solutions are required. Furthermore, it is possible that the solution is not unique. There may be no solutions or several solutions.

## Description



**Figure 1:** An example of a two-link planar manipulator with link lengths **l** = [1,1]T and joint angles **theta** = [𝜋/4, 𝜋/4]T.

Here, you will focus on a 2D planar manipulator with two links and two rotational degrees of freedom, as in the above figure. We will parameterize our manipulator with a vector of the two link lengths **l** = [*l*1, *l*2]T and a vector of the two joint angles **theta** = [𝜃1, 𝜃2]T.

The non-linear equations for positioning the end effector (i.e., the tip of the manipulator) at a point **p** = [x, y]T are as follows:

*f*1(𝜃1, 𝜃2) = *l*1 cos(𝜃1) + *l*2 cos(𝜃1+ 𝜃2) - *x =* 0,

*f*2(𝜃1, 𝜃2) = *l*1 sin(𝜃1) + *l*2 sin(𝜃1+ 𝜃2) - *y =* 0.

1. Fill in the function [**pos**, **J**] = evalRobot2D(**l**, **theta**) that returns the position, **pos** (2x1 vector), of the end-effector and the **J** of **pos** with respect to **theta** given the rotation angles **theta** and the fixed link lengths **l**. Implement **J** analytically.
2. While the previous question is about analytically calculating **J**, this question is about numerically calculating **J**. Fill in the function **J** = fdJacob2D(**l**, **theta**, h) that uses central differences (explained below) to compute the Jacobian using scalar h>0.
3. Evaluate several different values for h by comparing the finite-difference Jacobian to the analytic Jacobian for various **theta**.  
   As an example, the central-difference formula for the first column of the Jacobian would be as follows:

( evalRobot2D(**l**, **theta**+[h;0]) - evalRobot2D(**l**, **theta**-[h;0]) ) / (2\*h)

1. Of course, you will have to figure out the formula for the second column of the Jacobian. Answer the following questions in Matlab comments (**Note**: no writeup is required for this lab).
   1. Are the results close enough to be useful?
   2. Why would you use this finite-difference approximation instead of the analytic derivative?
   3. Include your test script in your submission, named **compareJacobian.m**
2. Fill in the function **theta** = invKin2D(**l**, **theta0**, **pos**, n, mode) that returns the rotation angles in a 2x1 vector **theta** that put the end-effector at **pos**. Your method should terminate when a constraint is satisfied, that is, use a threshold on the norm of the difference between the current position and **pos**, e.g. 0.001, or after n iterations, whichever comes sooner. When mode=1, you should use Newton’s method, and when mode=0, you should use Broyden’s method.
   1. [A note](https://drive.google.com/file/d/1kdXoAubn-RQSzQQfeioFHjr4BoVKJnXb/view?usp=drive_link) to help you understand Newton’s method
   2. For Broyden’s method, start with the Jacobian from your evalRobot2D function.
3. Use the provided **eval2D.m**, or an equivalent script (include it in your submission), with your own invKin2D function to evaluate the difference between Newton’s and Broyden’s methods. Explain in Matlab comments. Use the provided **plotRobot2D.m** function (called in eval2D.m) to plot.

Examples and hints:

**Q 2A.**

**Example 1:**

l = [1 1]'

th = [pi/3 pi/3]'

**pos = [ 0.0000 1.7321]'**

**J = [ -1.7321 -0.8660; 0.0000 -0.5000]**

**Example 2:**

l = [2 2]'

th = [pi/2 pi/2]'

**pos = [ -2.0000 2.0000]'**

**J = [-2.0000 -0.0000; -2.0000 -2.0000]**

**Q 2B.**

**Example 1:**

l = [1 1]'

th = [pi/3 pi/3]'

h = 0.01

**J = [ -1.7320 -0.8660; -0.0000 -0.5000]**

**2. Example 2:**

l = [2 2]'

th = [pi/2 pi/2]'

h = 0.5

**J = [ -1.9177 -0.0000; -1.9177 -1.9177]**

**Q 2C:** You should be able to verify your implementation using the provided eval2D.m and plotRobot2D.m scripts.