15. Matrix Libraries

Introduction:

Python is similar to Matlab in many ways. There is a stark difference when it comes to dealing with matrices, however:

- Matlab is a matrix language designed for scientists and engineers
- Python is a language designed for the general public

In Python, arrays are treated like strings, not matrices. This results in some strange results, such as

```
>> A = [1,2,3]
>> B = 2*A
>> print(B)
[[1, 2, 3],[1,2,3]]
```

This is a problem since matrices make some problems *much* easier to solve. In Python 3, a library called NumPi is commonly used to work with matrices. Unfortunately, NumPi doesn't exist for microPython. That's not a problem, however. With a little coding, you can write your own matrix library.

In this lecture, we'll create matrix functions for the following operations:

- Display(A) — display an nxm matrix with a formatted output
- Zeros(n,m) — create an nxm matrix containing all zeros
- Ones(n,m) — create an nxm matrix containing all ones
- Eye(n,n) — create an nxn matrix with ones on the diagonal (identity matrix)
- Random(n,m) — create an nxm matrix with random numbers in the range of (0,1)
- Transpose(A) — return the transpose of A
- Add(A,B) — return the sum of matrix A + B
- Multiply(A,B) — return the result of AB
- Multk(A,k) — return the scalar multiplication kA
- Inverse(A) — return the matrix inverse of A

Matrix Operations

Let's start with a definition of an nxm matrix. Define a 2x3 matrix

$$A = \begin{bmatrix} a & b & c \\ d & e & f \end{bmatrix}$$

as the Python variable

```
A = [[a,b,c],[d,e,f]]
```

With this definition, you can address element (i,j) with

```
A[i][j]
```

Note that this is slightly different than the way Matlab works. With Matlab

```
A = [1,2,3]              1x3 matrix
```

```
A = [1;2;3]              3x1 matrix
```

In Python, this would be treated as a string of three elements. To make it a 1x3 matrix, you need a second set of brackets

```
A = [[1,2,3]]            1x3 matrix
A = [[1],[2],[3]]        3x1 matrix
```

Matrix Functions

display(A): Displays a matrix, formatted with 3 decimal places

size(A): Return the dimensions of matrix A

```
>>> import matrix
>>> A = [[1,2,3],[4,5,6]]
>>> matrix.display(A)
     1.000     2.000     3.000
     4.000     5.000     6.000
>>> N = matrix.size(A)
>>> print(N)
[2, 3]
```

Transpose(A): Return the transpose of A

```
>>> AT = matrix.transpose(A)
>>> matrix.display(AT)
     1.000     4.000
     2.000     5.000
     3.000     6.000
```

mult_k(A,k): scalar muliply

```
>>> B = matrix.mult_k(A, 2.3)
>>> matrix.display(B)
     2.300     4.600     6.900
     9.200    11.500    13.800
```

zeros(n,m): return a zeros matrix with dimension nxm

```
>>> A = matrix.zeros(2,4)
>>> matrix.display(A)
     0.000     0.000     0.000     0.000
     0.000     0.000     0.000     0.000
```

eye(n,m): return an identity marix

```
>>> I = matrix.eye(4,4);
>>> matrix.display(I)
     1.000     0.000     0.000     0.000
     0.000     1.000     0.000     0.000
     0.000     0.000     1.000     0.000
     0.000     0.000     0.000     1.000
```

rand(n,m): return an nxm matrix with random numbers in the range of (0,1)

```
>>> A = matrix.rand(4,4)
>>> matrix.display(A)
     0.810     0.540     0.170     0.934
     0.222     0.291     0.498     0.551
     0.696     0.200     0.510     0.867
     0.994     0.764     0.786     0.378
```

matrix.add(A,B): return A+B.

- matrix A and B must have the same dimensions

```
>>> C = matrix.add(A,I)
>>> matrix.display(C)
     1.810     0.540     0.170     0.934
     0.222     1.291     0.498     0.551
     0.696     0.200     1.510     0.867
     0.994     0.764     0.786     1.378
```

matrix.inverse: return the inverse of a matrix

- The matrix must be nxn
- Gauss elimination is used to find the inverse (no limit on matrix size)

```
>>> AI = matrix.inv(A)
>>> matrix.display(AI)
    -0.143    -2.481     1.440     0.668
     1.379     1.671    -2.727     0.412
    -1.479     1.034     0.734     0.461
     0.667     0.997     0.194    -0.902
```

matrix.mult(A,B): return A*B

- A and B must have compatible dimensions

```
>>> C = matrix.mult(A, AI)
>>> matrix.display(C)
     1.000    -0.000     0.000     0.000
    -0.000     1.000     0.000     0.000
     0.000    -0.000     1.000     0.000
     0.000    -0.000     0.000     1.000
```

Convolution & Probability Functions

You can input probability density functions, such as 4, 6, and 8-sided dice, and use convolution to find the pdf the sum of die rolls.

For example, input the numbers 0..10 in to a vector:

```
>>> k = matrix.linspace(0,1,10)
>>> matrix.display([k])
 0.000  1.000  2.000  3.000  4.000  5.000  6.000  7.000  8.000
9.000  10.000
```

You can also input the pdf for a 6-sided die (p = 1/6 for numbers 1..6). Using convolution, you can determine the odds when rolling two six-sided dice (2d6) - which is the damage done by the D&D spell *Flaming Sphere*:

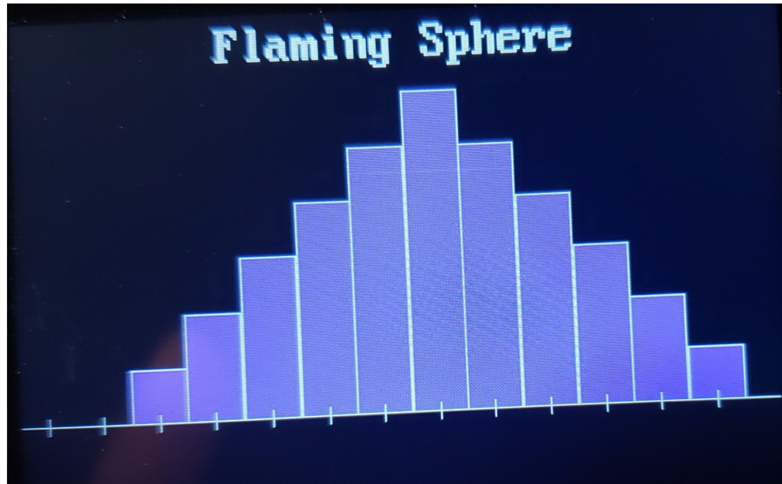
```
import matrix
import LCD

d6 = matrix.uniform(1,6,matrix.linspace(0,1,6))
d6x2 = matrix.conv(d6,d6)

matrix.display([d6x2])
0.000  0.000  0.028  0.056  0.083  0.111  0.139  0.167  0.139
0.111  0.083  0.056  0.028
```

This gives the odds of doing damage:

Odds of doing x damage												
0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	0.028	0.056	0.083	0.111	0.139	0.167	0.139	0.111	0.083	0.056	0.028

Note that this is where bar-charts are much more useful. Plotting this data on a bar chart gives a better idea of what this looks like:

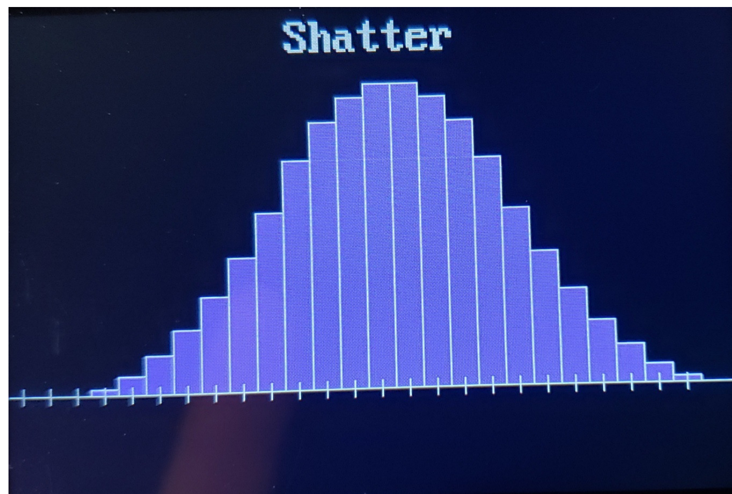
```
import matrix
import LCD

d6 = matrix.uniform(1,6,matrix.linspace(0,1,6))
d6x2 = matrix.conv(d6,d6)

Navy = LCD.RGB(0,0,5)
White = LCD.RGB(150,150,150)
Red = LCD.RGB(150,0,0)
LtBlue = LCD.RGB(50,50,150)

LCD.Init()
LCD.Clear(Navy)

matrix.BarChart(d6x2, White, LtBlue)
matrix.Title('Flaming Sphere',White, Navy)
```



The pdf for the sum of two 6-sided dice

More complicated (and higher-level spells) can be found a similar way. *Shatter* does 3d8 damage - meaning convolve the pdf for an 8-sided die three times:

```
import matrix
import LCD

d8 = matrix.uniform(1,8,matrix.linspace(0,1,8))
d8x2 = matrix.conv(d8,d8)
d8x3 = matrix.conv(d8x2,d8)

Navy = LCD.RGB(0,0,5)
White = LCD.RGB(150,150,150)
Red = LCD.RGB(150,0,0)
LtBlue = LCD.RGB(50,50,150)

LCD.Init()
LCD.Clear(Navy)

matrix.BarChart(d8x3, White, LtBlue)
matrix.Title('Shatter',White, Navy)
```

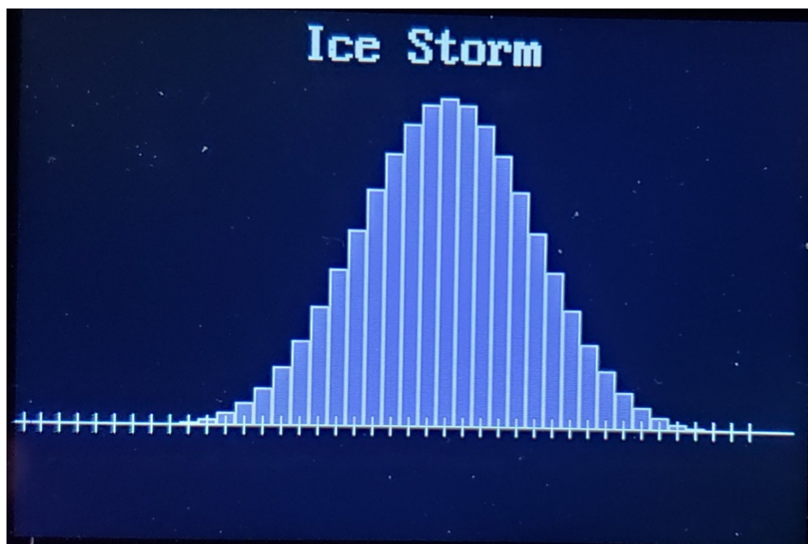


pdf for the sum of three 8-sided dice (3d8): also the damage for the D&D spell *shatter*

Note that after summing only three dice, the pdf is a bell-shaped curve. This is the *central limit theorem* in action.

Finally, you can mix and match dice - convolution doesn't care. The spell *Ice Storm* does 2d8 + 4d6 damage. In Python:

```
import matrix
import LCD

d6 = matrix.uniform(1,6,matrix.linspace(0,1,6))
d6x2 = matrix.conv(d6,d6)
d6x4 = matrix.conv(d6x2,d6x2)
d8 = matrix.uniform(1,8,matrix.linspace(0,1,8))
d8x2 = matrix.conv(d8,d8)
IceStorm = matrix.conv(d8x2,d6x4)

Navy = LCD.RGB(0,0,5)
White = LCD.RGB(150,150,150)
Red = LCD.RGB(150,0,0)
LtBlue = LCD.RGB(50,50,150)

LCD.Init()
LCD.Clear(Navy)

matrix.BarChart(IceStorm, White, LtBlue)
matrix.Title('IceStorm',White, Navy)
```



pdf for the spell *Ice Storm:* 4d6 + 2d8

With the sum of six dice, the bell-shaped curve of a normal distribution is becoming even more apparent.

Least Squares Curve Fitting

Given a set of data (x, y), determine the least squares curve fit

$$y \approx ax^2 + bx + c$$

This problem is easier to solve if you place it in matrix form

$$Y = \begin{bmatrix} x^2 & x & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = BA$$

The least-squares solution for A is

$$A = (B^T B)^{-1} B^T Y$$

For example, determine a parabolic approximation for sin(x) over the range of (0,1)

$$\sin(x) \approx ax^2 + bx + c$$

First, define the basis function, B:

$$B = \begin{bmatrix} x^2 & x & 1 \end{bmatrix}$$

Then compute A using matrix operations. In Python:

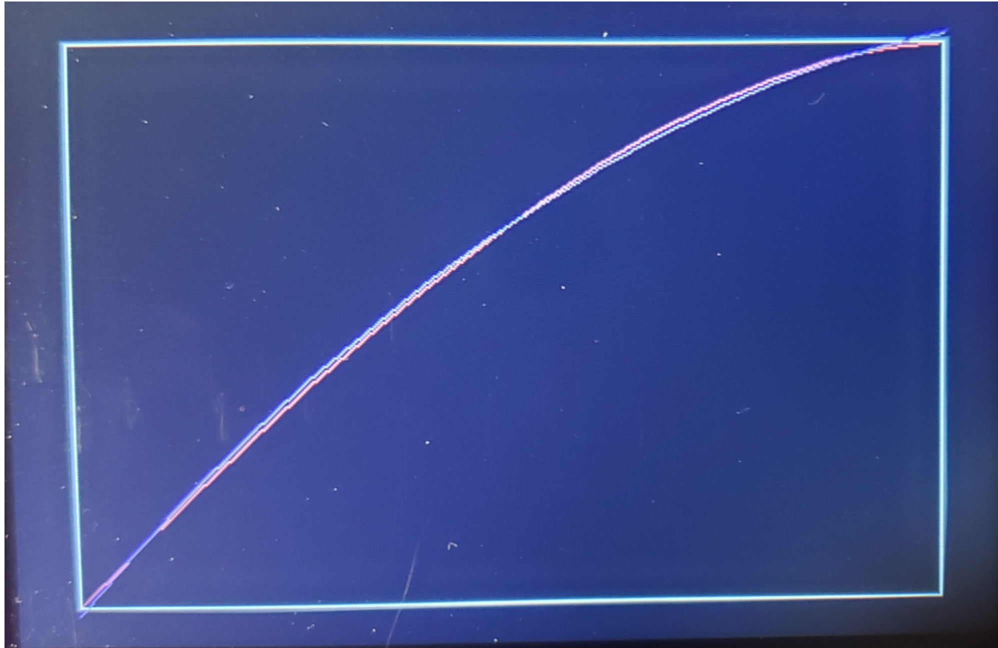
```
import matrix
import LCD
import math
import random


# approximate sin(t) = a*t^2 + b*t + c over 0<t<1.5

t = matrix.linspace(0,0.1,1.59)
t = matrix.transpose([t])
n = len(t)

Y = matrix.zeros(n, 1)
for i in range(0,n):
    Y[i][0] = math.sin(t[i][0])

t2 = matrix.power(t,2)
t0 = matrix.power(t,0)
B = matrix.append(t2,t)
B = matrix.append(B,t0)

Bt = matrix.transpose(B)
BtB = matrix.mult(Bt,B)
BtBi = matrix.inv(BtB)
BtY = matrix.mult(Bt,Y)
A = matrix.mult(BtBi,BtY)
matrix.display(A)

Navy = LCD.RGB(0,0,5)
Pink = LCD.RGB(150,50,50)
White = LCD.RGB(150,150,150)
LtBlue = LCD.RGB(50,50,150)

LCD.Init()
LCD.Clear(Navy)

Yf = matrix.mult(B, A)

matrix.Plot(t,Y,0,0,1,1,Pink)
matrix.Plot(t,Yf,0,0,1,1,LtBlue)
```

```
-0.326
 1.175
 0.017
```

The curve fit is

$$\sin(t) \approx -0.326t^2 + 1.175t + 0.017$$

The graphic display shows the actual data (Y) and the curve fir (Yf)



sin(t) vs. t (pink0 and least-squares curve fit (blue)

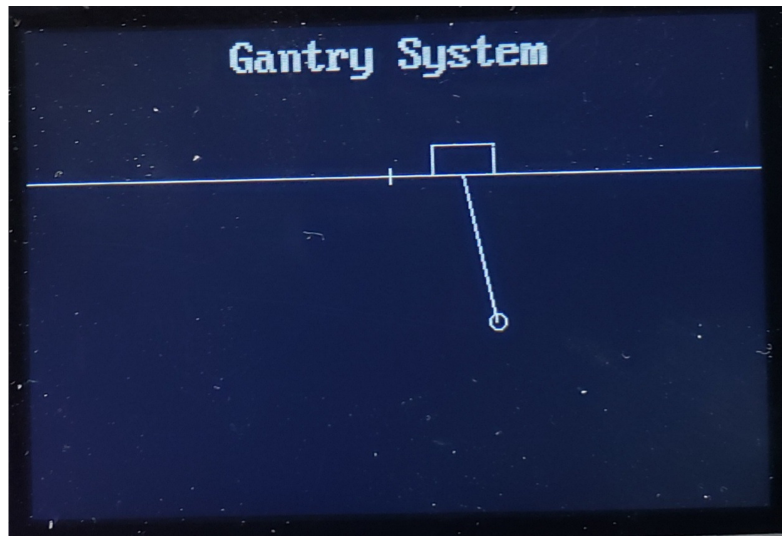
Where least squares really shines is when you're collecting data and want to curve fit your actual data. More on this later when we cover data collection and recursive least squares.

Dynamic Simulations: Gantry System

Finally, the dynamics of systems such as a gantry system can be found using matrix operations. For this system

- A force is applied to a mass (the rectangle at the top)
- This pushes the mass left and right
- Attached to the mass is a 1m string, connected to a 1kg load.

This models something you find in shop floors: an overhead gantry system is used to lift and move heavy objects, such as an engine block, across the shop floor. The goal of this system is to move the grantry while avoiding swinging motion of the load



Gantry System from ECE 463 Lecture #7

To simulate this system on a Pi-Pico board, you first need the dynamics. Taking the results from lecture #7 for ECE 463 Modern Controls, the dynamics of a gantry system are:

$$\begin{bmatrix} 3 & \cos\theta \\ \cos\theta & 1 \end{bmatrix}\begin{bmatrix} \ddot{x} \\ \ddot{\theta} \end{bmatrix} = \begin{bmatrix} \dot{\theta}^2 \sin\theta \\ -g\sin\theta \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} F$$

where

- x is the position of the cart
- θ is the angle of the beam
- F is the force on the base, and
- g is the acceleration due to gravity.

At any instant, the acceleration can be found by multiplying on the left by the inverse of the mass matrix. Once the acceleration is known,

- integrate once to get velocity (or angular velocity)
- integrate again to get position (or angle)

Manual control can be implemented by making the force acting on the cart (F) proportional to the left-right position of the joystick.

The program for implementing the gantry system has several subroutines:

Gantry Dynamics: Input the state of the gantry system and its input, return the derivative of the state.

The state is defined as the position and velocity:

$$X = \begin{bmatrix} x \\ \theta \\ sx \\ s\theta \end{bmatrix}$$

```
def GantryDynamics(X, U):
    x = X[0]
    q = X[1]
    dx = X[2]
    dq = X[3]
    g  = 9.8
    M = [[3, math.cos(q)],[math.cos(q),1]]
    C = [[dq*dq*math.sin(q)],[-g*math.sin(q)]]
    F = [[U],[0]]
    Mi = matrix.inv(M)
    MC = matrix.mult(Mi,C)
    MF = matrix.mult(Mi,F)
    ddX = matrix.add(MC,MF)
    dX = [dx,dq,ddX[0][0],ddX[1][0]]
    return(dX)
```

The display routine displays the gantry on the graphics LCD

```
def GantryDisplay(X, Color):
    x = X[0]
    q = X[1]
    x0 = 240 + x*100
    y0 = 100
    x1 = x0 + 100*math.sin(q)
    y1 = y0 + 100*math.cos(q)

    Navy = LCD.RGB(0,0,5)
    White = LCD.RGB(150,150,150)
    #LCD.Clear(Navy)
    LCD.Line(0,y0,479,y0,White)
    LCD.Line(240,y0-5,240,y0+5,White)
    LCD.Box(x0-20,y0,x0+20,y0-20,Color)
    LCD.Line(x0,y0,x1,y1,Color)
    LCD.Circle(x1,y1,5,Color)
```

Integrate implements numerical integration using Euler integration

```
def Integrate(X, dX, dt):
    Y = [0,0,0,0]
    for i in range(0,4):
        Y[i] = X[i] + dX[i] * dt
    return(Y)
```

The main routine then loops every 20ms to simulate the gantry system

```
# Gantry
import matrix
import LCD
import math
import time
from machine import ADC

def GantryDynamics:
# insert code here
def GantryDisplay:
# insert code here
def Integrate:
# insert code here

#------------------------------------

a2d0 = machine.ADC(0)
F0 = a2d0.read_u16()

White = LCD.RGB(150,150,150)
Navy = LCD.RGB(0,0,5)

LCD.Init()
LCD.Clear(Navy)

k = 50 / 32000

X = [-2,0,0,0]
dt = 0.02
t = 0
while(t < 10):
    F = ( a2d0.read_u16() - F0 ) * k
    dX = GantryDynamics(X, F)
    GantryDisplay(X,Navy)
    X = Integrate(X, dX, dt)
    t = t + dt
    GantryDisplay(X,White)
    time.sleep(0.01)
```

Similar techniques can be used to model other dynamic systems such as a heat equation, inverted pendulum, double pendulum, ball and beam systems, etc.

Summary:

Matrices are really useful: they make some problems such as least-squares curve fitting much easier to solve. While Python is not a matrix language, by writing your own matrix functions you can manipulate matrices. The result isn't as user friendly as Matlab, but it works.