Ch. 1 - ~~MATLAB~~ Python Introduction

A shortened "slide-y" version of the notebook companion to Ch. 1 in Kutz.

Why python?

In modern era,essential aspects of engineering include

- "BIG DATA"
- Simulations

To fully participate need:

- Understanding of basic numerical methods
- Ability read/understand code
- Ability to implement customized algorithms

Most-used languages

- Statisticians mostly use R
- Engineers often use Python (and we'll follow along)

Virtues of python

- Interpreted language for ease of development
- Just-in-time (JIT) compilers are available to achieve compiled performance
    - Subject of ME 574
- Large, mature set of libraries
- Clean syntax
    - Not too much punctuation
    - No need to specify data types explicitly

§ 1.1 Vectors and Matrices

- Engineering simulations involve large collections of numbers
- We would like those collections to behave like vectors and matrices
- "Base" Python stores a collection of data in a **container** such as `list`
- But a `list` does NOT behave like a vector

```
x = [1,2,3] #associate a list with the name `x`
y = [4,5,6] #associate a list with the name `y`
x+y #evaluate the "sum" of the two lists
```

>>> [1, 2, 3, 4, 5, 6] #'+' concatenates instead of adding

Data container for matrix/vector operations: numpy array

- Mash-up of numerics and python $\rightarrow$ `numpy`
- To access array capabilities in python:

```
import numpy as np
```

- Utilize `ndarray` containers

Python-language version of § 1.1

```python
import numpy as np   #import numpy (abbreviated as `np`)

# create a vector named x with entries 1,3,2
x = np.array([1,3,2])
```

- In notebook, output of last line appears in output cell:

  >>>array([1,3,2])

- To get output from python file, use `print`:

```python
# print the string "Print output: x = " followed by the value associated with the name x
print("Print x = ", x, "; type(x) = ", type(x))
```

>>> x = [1 3 2]; type(x) = numpy.ndarray

Create the corresponding column vector

```
x_t = np.array([[1],[3],[2]])

# print the value associated with the name x_t
print("x_t =\n", x_t)
```

```
>>> x_t =
[[1]
[3]
[2]]
```

- This should make it clear why we tend to print row vectors when possible...

Verify vector operation properties

- Vector addition, scalar multiplication, inner product

```
print('sum of x and x is', x + x)
print('3*x = ', 3*x)
print('dot product x.x is', np.dot(x,x_t))
```

```
>>> sum of x and x is [2 6 4]
3*x = [3 9 6]
dot product x.x is [14]
```

Creating a sequence of values

```
#integers from 0 through 10 using range
print(range(0,11,1)) #can abbreviate as range(11)
```

>>> range(0, 11)

- `range` uses **lazy** evaluation (to help handle BIG data)
- To print the actual values, "unpack" using `*`

```
print(*range(11))
```

>>> 0 1 2 3 4 5 6 7 8 9 10

- `range(n)` starts at 0 and ends at n-1 (strict inequality)

More sequences

- Create the even numbers from 0 through 10

```
evens = range(0,11,2)
print(*evens, type(evens))
```

>>> 0 2 4 6 8 10, <class 'range'>

- To get the same numbers in an array, use `arange(start, stop, step)`:

```
even_arr = np.arange(0,11,2)
print(even_arr, type(even_arr))
```

>>> [0 2 4 6 8 10], <class 'numpy.ndarray'>

Array of floats

- Can also use `arange` with floats
  `print(np.arange(0, 1, 0.2))`
  >>> [0. 0.2 0.4 0.6 0.8]
- Inclusion of terminal value depends on roundoff error (more soon...)
- Control number of entries instead with `linspace`:

```
v = np.linspace(0,1,6)
print(v)
```

|>>> [0. 0.2 0.4 0.6 0.8 1.]

2D arrays

- Kutz' Eq. (1.1.8) brings us to representing a matrix
- numpy's ndarray type supports arrays of dimension 2 (and higher)

```
A_list = [[1,3,2],[5,6,7],[8,3,1]] #create a nested list of lists
A = np.array(A_list) #use np.array to convert to a 2D array
print(A) # on a separate line for more readable formatting
print(type(A,A.shape, A.size))
```

```
|>>> A=
[[1 3 2]
[5 6 7]
[8 3 1]]
(numpy.ndarray, (3,3), 9)
```

Accessing selected elements

- Use array name followed by comma-separated indices in brackets

  `A[1,2]` >>> 7

- Access an entire row

  `print(A[1])` >>>[5 6 7]

- Access an entire column

  `print(A[:,2])` >>> [2 7 1]

- To actually get a column vector, use `transpose`

```
row_vec = A[:,2] #get elements as row
print(np.transpose([row_vec])) #nest, transpose, and print (can also use row_vec.T)
```

>>> [[2]

[7]

[1]]

Eq(1.1.12)

```python
B = np.array([[1,7,9,2],[2,3,3,4],[3,0,2,6],[6,1,5,5]])
print(B)
```

```
[[1 7 9 2]
[2 3 3 4]
[3 0 2 6]
[6 1 5 5]]
```

```python
print(B[1:3,1])
print("B has type: ", type(B))
```

```
>>> [3 0]
B type is: <class 'numpy.ndarray'>
```

Using matrix attributes

```
print(B[-1,1:B.shape[1]])
```

```
>>> [1 5 5]
```

```
print(B[1:B.shape[0], 1:B.shape[1]])
```

```
[[3 3 4]
[0 2 6]
[1 5 5]]
```

```
D = np.array([B[0,1:4],B[0:3,2]])
print(D)
```

```
[[7 9 2]
[9 3 2]]
```

Handling complex entries

- python uses `j` to represent $\sqrt{-1}$
- Create a complex 1×3 array. Compute transpose and complex conjugate:

```
x=np.array([[3+2j,1,8]])
print("x=", x)
x_t = x.transpose()
print("Transpose of x=\n", x_t)
x_c = x.conjugate()
print("Conjugate of x=", x_c)
```

```
>>> x= [[3.+2.j 1.+0.j 8.+0.j]]
Transpose of x=
[[3.+2.j]
[1.+0.j]
[8.+0.j]]
Conjugate of x= [[3.-2.j 1.-0.j 8.-0.j]]
```

Eq.(1.1.9) Vector maniupulations

```
x = np.array([-1,2,3,5,-2])
print(x)
print(x > 0) #boolean values depending on sign of x
print(1*(x>0)) #same as above with true->1, false->0
print(x+(x > 0)) #add vector above to x
print(3*(x > 0)) #multiply the 0/1 vector by 3
```

```
>>> [-1 2 3 5 -2]
[False True True True False]
[0 1 1 1 0]
[-1 3 4 6 -2]
[0 3 3 3 0]
```

§ 1.2 Logic, Loops, and Iterations

- Algorithms require logical control of execution (based on count or intermediate results)
- Essential control structures:
  - `for` loop
  - `if-elif-else` conditionals
- First example (in MATLAB with % comments)

```
a=0 % assign value zero to variable named a
for j=1:5 % loop over values of j in the set{1,2,3,4,5}
    a=a+j % increment the value of a by the current value of j
end % terminate the loop
```

```python
a=0 # assign the value zero to the variable a
# loop over j in {1,2,3,4,5}
# note that range is "start inclusive" and "end exclusive"
for j in range(1,6):
    print('a=', a, ' ; j=',j) # print the current values
    a=a+j # increment a by current value of j
    print('updated value of a is ', a) # print the updated value
print('final value of a is ', a) # terminate the loop (by un-indenting) and print final value
```

```
>>> a= 0 ; j= 1
updated value of a is 1
a= 1 ; j= 2
updated value of a is 3
a= 3 ; j= 3
updated value of a is 6
a= 6 ; j= 4
updated value of a is 10
a= 10 ; j= 5
updated value of a is 15
```

Example with twist #1: increment by 2

```
a=0 # assign the value zero to the variable a
# loop over j in {1,2,3,4,5} noting that range is "start inclusive" and "end exclusive"
for j in range(1,6,2):
    print('a=', a, ' ; j=',j) # print the current values
    a=a+j # increment a by current value of j
    print('updated value of a is ', a) # print the updated value
print('final value of a is ', a) # terminate the loop (by un-indenting) and print final value
```

```
>>> a= 0 ; j= 1
updated value of a is 1
a= 1 ; j= 3
updated value of a is 4
a= 4 ; j= 5
updated value of a is 9
final value of a is 9
```

## Example with twist #2: iterate over list of values

```
a=0 # assign the value zero to the variable a
j_vals = [1,5,4] # values to iterate over
for j in j_vals:
    print('a=', a, ' ; j=',j) # print the current values
    a=a+j # increment a by current value of j
    print('updated value of a is ', a) # print the updated value
print('final value of a is ', a) # terminate the loop (by un-indenting) and print final value
```

```
>>> a= 0 ; j= 1
updated value of a is 1
a= 1 ; j= 5
updated value of a is 6
a= 6 ; j= 4
updated value of a is 10
final value of a is 10
```

Conditional execution

- MATLAB uses `if-elseif-else` construct
- Python uses `if-elif-else` with the following syntax:

```
if (logical statement 0):
    (expressions to execute when logical statement 0 is true)
elif (logical statement 1):
    (expressions to execute when logical statement 0 is false and logical statement 1 is true)
...
else:
    (expressions to execute when all logical statements are false)
```

- `elif` and `else` clauses are optional
- parentheses are for readability (not required)

Ternary conditional

- Abbreviation for short conditional statements
- square brackets indicate python statements

```
[on_true] if [expression] else [on_false]
```

- Example: choose minimum of 2 values

```
if (a>b):
    a
else:
    b
#tenary equivalent
a if (a>b) else b
```

First algorithm implementation: root-finding by bisection

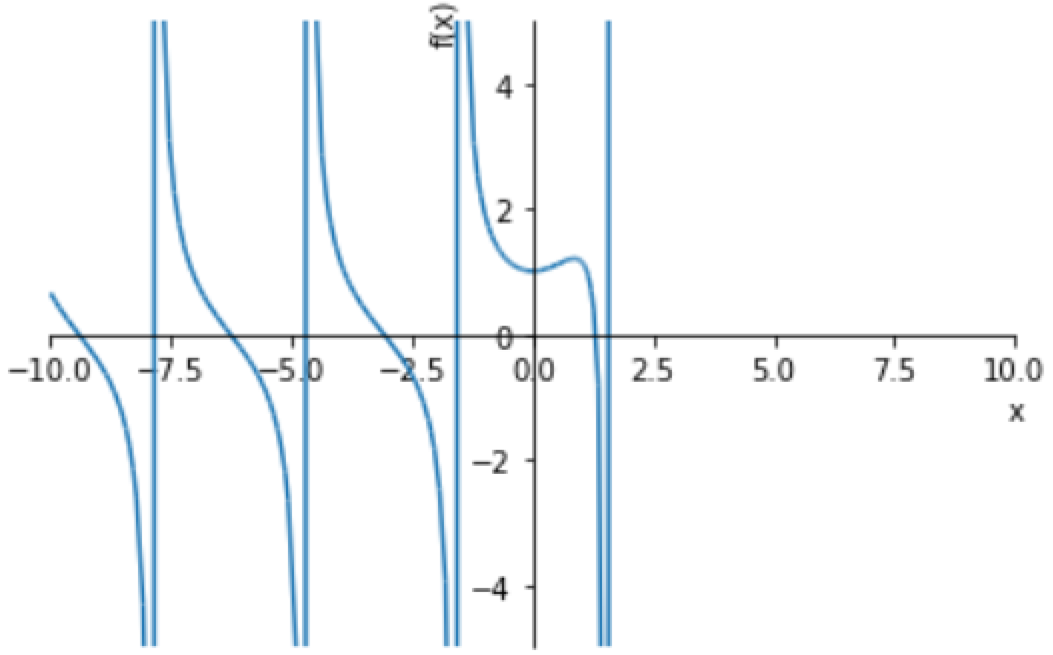
- Practical application of `for` and `if`
- Locate a root of transcendental equation

$$f(x) = exp(x) - tan(x)$$

- Quick aside using `sympy`, the python package for symbolic computation (since mose of what we do will be numeric)

```
from sympy import symbols, exp, tan
from sympy.plotting import plot
x = symbols('x') # set 'x' to be the name of a symbolic variable
plot(exp(x)-tan(x),(x,-10,10), ylim=(-5,5)) # plot the function
```

Plot of the transcendental function

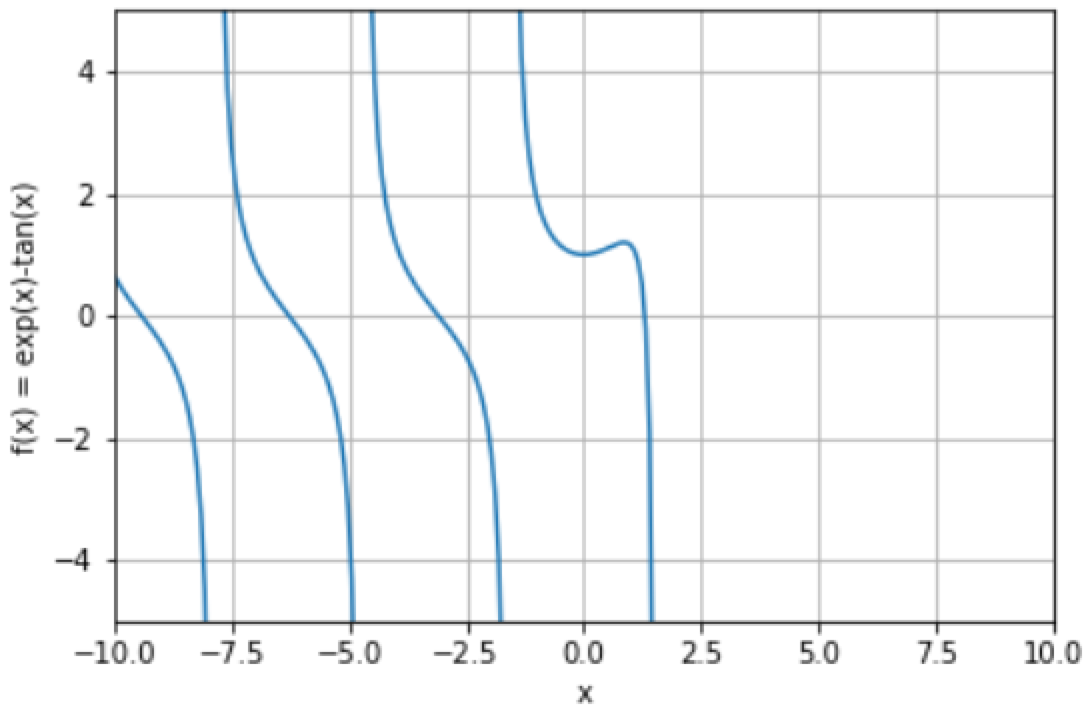


- Sympy helps us make a basic plot
- Removing asymptotes etc. is not so easy

Numerical evaluation and plotting

```python
# numpy version
import matplotlib.pyplot as plt # import matplotlib's pyplot function under the abbreviation plt
numpts = 250 # desired array length
x = np.linspace(-10.,10., numpts) # assign to x the array of equally spaced abscissa values
y = np.zeros(numpts) # assign to y an array of the corresponding length initialized to hold zeros
for i in range(numpts): # loop over the array size
    # compute the function for an input value and store in the corresponding entry in y
    y[i] = np.exp(x[i])-np.tan(x[i])
    # comment out this conditional for a plot including asymptotic lines
    # leave it in to replace large values with the `numpy` designation for infinity which is not plotted
    if abs(y[i]>10):
        y[i]=np.inf
plt.plot(x,y) # create a plot with abscissas (horiz. coords.) from x and ordinates (vert. coords.) from y
plt.axis([-10,10,-5,5]) # plot within the range -10<x<10 and -5<y<5
plt.xlabel('x') # label for the horizontal axis
plt.ylabel('f(x) = exp(x)-tan(x)') #label for the vertical axis
plt.grid(True) # show a grid to help estimating values
plt.savefig("figure.png") #save the plot to a file
plt.show() # show the plot as specified
```

Plot based on grid of sampled data



- This gives us a visual guide of where to hunt for roots

Rootfinding/isolation by bisection

1. Choose a starting interval (with sign change at endpoints)
2. Compute value at midpoint of interval
3. Discard the subinterval whose endpoint values have the same sign
4. Repeat from step 2 until terminal condition is satisfied (typically based on interval size or function value)

Following example in text, choose starting interval $x = [xl, xr] = [-4, -2.8]$ and verify there is a sign change:

```python
xl = -4
xr = -2.8
fl = np.exp(xl)-np.tan(xl) # compute value at left endpoint
fr = np.exp(xr)-np.tan(xr) # compute value at right endpoint
print('f(xl)=',fl,'f(xr)=', fr)
# Test for valid starting interval: product of endpoint function values must be negative
print('Valid starting interval? ', fl*fr<0)
```

```
>>> f(xl)= 1.176136921238312 f(xr)= -0.2947197690259581
Valid starting interval? True
```

```
    print(j,xc,fc) #print iteration number, center location, and center function value
    if fc>0:
        xl=xc #move left boundary
    else:
        xr=xc #move right boundary
    if (abs(fc)<1e-5): #test termination criterion
        break #exit the loop
print('root near x=', xc, 'where f=',fc)
```

```
0 -3.4 0.29769017082775123
1 -3.0999999999999996 0.0034325478079211474
2 -2.9499999999999997 -0.14163219997111393
3 -3.0249999999999995 -0.06856603516934925
4 -3.0624999999999996 -0.032487370202019858
5 -3.08125 -0.014514157563857988
6 -3.0906249999999997 -0.005538307847695087
7 -3.0953124999999995 -0.0010523574492767515
8 -3.0976562499999996 0.0011902131305878
9 -3.0964843749999993 6.895891408374016e-05
10 -3.0958984374999994 -0.0004916913009686621
```

§ 1.3 Iteration: Newton-Raphson Method

- Create the python version of the listing `newton.m` below Eq. (1.3.10).

```
x=-4 #initial estimate of fixed point
max_iter = 1000
for j in range(max_iter):
    x = x-(np.exp(x)-np.tan(x))/(np.exp(x)-(1/np.cos(x))**2)
    f=np.exp(x)-np.tan(x)

    if abs(f)<1e-5:
        break

print('root is near',x,'where function value is',f)
```

```
>>> root is near -3.096414598500469 where function value is 2.194579281487863e-06
```

§ 1.4 Function Calls, Input/Output, & Debugging

- When a set of evaluations will be done repeatedly, group them as a function
  - Enhance clarity
  - Save keyboarding
  - Simplify debugging
- Write docstrings so you and others (including future you) understand your code

```python
def myfun(x):
    '''
    example of function creation from Section 1.4
    myfun takes one numeric input x and produces the numeric output x**3+sin(x)
    '''
    return x**3+np.sin(x)
```

Execution

- Previous slide defines the function but does not make any computation happen
- Need to call for execution and provide input values
- How do you know what inputs to provide?

  ```
  help(myfun)
  ```
  >>> Help on function myfun in module **main**:

myfun(x)

example of function creation from Section 1.4

myfun takes one numeric input x and produces the numeric output x**3+sin(x)

```
print("myfun(1) = ", myfun(1), "; 1**3 + sin(1) = ", 1**3 + np.sin(1))
```
>>> myfun(1) = 1.8414709848078965 ; 1**3 + sin(1) = 1.8414709848078965

Function evaluations in a control loop

- For an array of input values, compute a corresponding array of output values

```
x = np.linspace(0,10,11) # create an array of input values
y = np.zeros(11) # create array (initialized to zeros) to store output values
for i in range(11):
    y[i]=myfun(x[i])
    print('myfun(',x[i],')=',y[i]) #print the individual input/output relations
print('Input and output as arrays:') #print input and output arrays
print('x=',x)
print('myfun(x)=',y)
```

```
>>> myfun( 0.0 )= 0.0
myfun( 1.0 )= 1.8414709848078965 ...
Input and output as arrays:
x= [ 0. 1. 2. 3. 4. 5. 6. 7. 8. 9. 10.]
myfun(x)= [ 0. 1.84147098 8.90929743 27.14112001 63.2431975
124.04107573 215.7205845 343.6569866 512.98935825 729.41211849
999.45597889]
```

Multiple inputs/arguments