TEXAS ADVANCED COMPUTING CENTER

WWW.TACC.UTEXAS.EDU

The University of Texas at Austin

Python 101/201

S. Charlie Dey

Director, Training and Professional Development

Agenda

- Introduction to the Jupyter Notebook
- Welcome to Python
- Basic Linear Algebra
- Using Numpy
- Using Pandas
- Matplotlib
- Interactive Plots using MPLD3

What are Jupyter Notebooks?

A web-based, interactive computing tool for capturing the whole computation process: developing, documenting, and executing code, as well as communicating the results.

How do Jupyter Notebooks Work?

An open notebook has exactly one interactive session connected to a kernel which will execute code sent by the user and communicate back results. This kernel remains active if the web browser window is closed, and reopening the same notebook from the dashboard will reconnect the web application to the same kernel.

What's this mean?
Notebooks are an interface to kernel, the kernel executes your code and outputs back to you through the notebook. The kernel is essentially our programming language we wish to interface with.

TACC

Jupyter Notebooks, Structure

- Code Cells

  Code cells allow you to enter and run code
  Run a code cell using Shift-Enter

- Markdown Cells

  Text can be added to Jupyter Notebooks using Markdown cells. Markdown is a popular markup language that is a superset of HTML.

TACC

Jupyter Notebooks, Structure

- Markdown Cells

  You can add headings:

  # Heading 1
  # Heading 2
  ## Heading 2.1
  ## Heading 2.2

  You can add lists

  1. First ordered list item
  2. Another item
  ⋅⋅* Unordered sub-list.
  1. Actual numbers don't matter, just that it's a number
  ⋅⋅1. Ordered sub-list
  4. And another item.

Jupyter Notebooks, Structure

- Markdown Cells

pure HTML

```
<dl>
  <dt>Definition list</dt>
  <dd>Is something people use sometimes.</dd>

  <dt>Markdown in HTML</dt>
  <dd>Does *not* work **very** well. Use HTML <em>tags</em>.</dd>
</dl>
```

And even, Latex!

```
$e^{i\pi} + 1 = 0$
```

Jupyter Notebooks, Workflow

Typically, you will work on a computational problem in pieces, organizing related ideas into cells and moving forward once previous parts work correctly. This is much more convenient for interactive exploration than breaking up a computation into scripts that must be executed together, as was previously necessary, especially if parts of them take a long time to run.

Jupyter Notebooks, Workflow

Let a traditional paper lab notebook be your guide:

Each notebook keeps a historical (and dated) record of the analysis as it's being explored.

The notebook is not meant to be anything other than a place for experimentation and development.

Notebooks can be split when they get too long.

Notebooks can be split by topic, if it makes sense.

TACC

Jupyter Notebooks, Shortcuts

- **`Shift-Enter`: run cell**
    - Execute the current cell, show output (if any), and jump to the next cell below. If `Shift-Enter` is invoked on the last cell, a new code cell will also be created. Note that in the notebook, typing `Enter` on its own *never* forces execution, but rather just inserts a new line in the current cell. `Shift-Enter` is equivalent to clicking the `Cell | Run` menu item.

Jupyter Notebooks, Shortcuts

- `Ctrl-Enter`: **run cell in-place**
  - Execute the current cell as if it were in "terminal mode", where any output is shown, but the cursor *remains* in the current cell. The cell's entire contents are selected after execution, so you can just start typing and only the new input will be in the cell. This is convenient for doing quick experiments in place, or for querying things like filesystem content, without needing to create additional cells that you may not want to be saved in the notebook.

Jupyter Notebooks, Shortcuts

- **`Alt-Enter`: run cell, insert below**
  - Executes the current cell, shows the output, and inserts a *new* cell between the current cell and the cell below (if one exists). (shortcut for the sequence `Shift-Enter`,`Ctrl-m a`. (`Ctrl-m a` adds a new cell above the current one.))
- **`Esc` and `Enter`: Command mode and edit mode**
  - In command mode, you can easily navigate around the notebook using keyboard shortcuts. In edit mode, you can edit text in cells.

Introduction to Python

Hello World!
Data types
Variables
Arithmetic operations
Relational operations
Input/Output
Control Flow
More Data Types!
Matplotlib

the magic number is:

4

Python

```
print “Hello World!”
```

Let's type that line of code into a Code Cell, and hit Shift-Enter:

```
Hello World!
```

Python

```
print 5
print 1+1
```

Let's add the above into another Code Cell, and hit Shift-Enter

```
5
2
```

Python – Variables

You will need to store data into variables
You can use those variables later on
You can perform operations with those variables
Variables are declared with a **name**, followed by '=' and a **value**

An integer, string,…

When declaring a variable, **capitalization** is important:
'A' <> 'a'

Python – Variables

in a code cell:

```
five = 5
one = 1
print five
print one + one
message = “This is a string”
print message
```

Notice: We're not "typing" our variables, we're just setting them and allowing Python to type them for us.

TACC

Python – Data Types

in a code cell:

```
integer_variable = 100
floating_point_variable = 100.0
string_variable = “Name”
```

Notice: We're not "typing" our variables, we're just setting them and allowing Python to type them for us.

Python – Data Types

Variables have a type

You can check the type of a variable by using the type() function:

```
print type(integer_variable)
```

It is also possible to change the type of some basic types:

`str(int/float)`: converts an integer/float to a string
`int(str)`: converts a string to an integer
`float(str)`: converts a string to a float

Be careful: you can only convert data that actually makes sense to be transformed

TACC

Python – Arithmetic Operations

+	Addition	1 + 1 = 2
-	Subtraction	5 – 3 = 2
/	Division	4 / 2 = 2
%	Modulo	5 % 2 = 1
*	Multiplication	5 * 2 = 10
//	Floor division	5 // 2 = 2
**	To the power of	2 ** 3 = 8

Python – Arithmetic Operations

Some experiments:

```
print 5/2
print 5.0/2
print "hello" + "world"
print "some" + 1
print "number" * 5
print 3+5*2
```

Python – *Arithmetic Operations*

Some more experiments:

```
number1 = 5.0/2
number2 = 5/2
```

what **type()** are they?

```
type(number1)
type(number2)
```

now, convert number2 to an integer:

```
int(number2)
```

Python – *Reading from the Keyboard*

Let put the following into a new Code Cell:

```
var = input("Please enter a number: ")
```

Let's run this cell!

Python – *Reading from the Keyboard*

Let put the following into a new Code Cell:

```
var2 = input("Please enter a string: ")
```

Let's run this cell!

put the word **Hello** as your input.

What happened?

Python – Making the output prettier

Let put the following into a new Code Cell:

```
print "The number that you wrote was : ", numIn
print "The number that you wrote was : %d" % numIn

print "the string you entered was: ", stringIn
print "the string you entered was: %s" % stringIn
```

Want to make it prettier?

\n for a new line

\t to insert a tab

```
print " your string: %s\n your number: %d", %(numIn, stringIn)
```

for floating points, use %f

TACC

Python – Writing to a File

Let put the following into a new Code Cell:

```
my_file = open("output_file.txt",'w')
vars = "This is a string\n"
my_file.write(vars)
var3 = 10
my_file.write("\n")
my_file.write(str(var3))
var4 = 20.0
my_file.write("\n")
my_file.write(str(var4))
my_file.close()
```

TACC

Python – Reading from a File

When opening a file, you need to decide “how” you want to open it:

Just read?

Are you going to write to the file?

If the file already exists, what do you want to do with it?

r	read only (default)
w	write mode: file will be overwritten if it already exists
a	append mode: data will be appended to the existing file

Python – Reading from a File

Let's read from the file we created in the previous cell.

```
my_file = open("output_file.txt",'r')
content = my_file.read()
print content
my_file.close()
```

TACC

Python – Reading from a File

Let's read it line by line

```
my_file = open("output_file.txt",'r')
vars = my_file.readline()
var5 = my_file.readline()
var6 = my_file.readline()
print "String: ", vars
print "Integer: ", var1
print "Float: ", var2
my_file.close()
```

Python – Reading from a File

Tweak it a bit to make the code easier to read… introducing 'with'!

(remember the MAGIC NUMBER!   Hint: it's 4)

```
with open("output_file.txt",'r') as f:
    vars = f.readline()
    var5 = f.readline()
    var6 = f.readline()
    print "String: ", vars
    print "Integer: ", var1
    print "Float: ", var2
```

TACC

Python – Control Flow

So far we have been writing instruction after instruction where every instruction is executed

What happens if we want to have instructions that are only executed if a given condition is true?

Python – if/else/elif

The if/else construction allows you to define conditions in your program

(remember the MAGIC NUMBER!   Hint: it's 4)

```
if conditionA:
    statementA
elif conditionB:
    statementB
else:
    statementD
this line will always be executed (after the if/else)
```

Python – *if/else/elif*

The if/else construction allows you to define conditions in your program

(remember the MAGIC NUMBER!   Hint: it's 4)

```
if conditionA:
    statementA
elif conditionB:
    statementB
else:
    statementD
this line will always be executed (after the if/else)
```

conditions are a datatype known as booleans, they can only be true or false

Python – if/else/elif

Let's look at some example of booleans.

type the following into a code cell

```
a = 2
b = 5

print a>b
print a<b
print a == b
print a != b
print b>a or a==b
print b<a and a==b
```

TACC

Python – *if/else/elif*

A simple example

```
if var>10:
    print "You entered a number greater than 10"
else:
    print "you entered a number less than 10"
```

TACC

Python – if/else/elif

You can also nest if statements together:

```
if condition1:
    statement1
    if condition2:
        statement2
    else:
        if condition3:
            statement3 # when is this statement executed?
else:  # which ‘if’ does this ‘else’ belong to?
    statement4  # when is this statement executed?
```

TACC

Exercise:

enter a number from the keyboard into a variable.

using type casting and if statements, determine if the number is even or odd

Python – For Loops

When we need to iterate, execute the same set of instructions over and over again… we need to loop! and introducing range()

(remember the MAGIC NUMBER!   Hint: it's 4)

```
for x in range(0, 3):
    print "Let's go %d" % (x)
```

Python – For Loops, nested loops

When we need to iterate, execute the same set of instructions over and over again… we need to loop! and introducing range()

(remember the MAGIC NUMBER!   Hint: it's 4)

```
for x in range(0, 3):
    for y in range(0,5):
        print "Let's go %d %d" % (x,y)
```

TACC

Exercise:

using nested for-loops and nested if statements, write a program that loops from 3 to 100 and print out the number if it is a prime number.

Python – While Loops

Sometimes we need to loop while a condition is true...

(remember the MAGIC NUMBER!   Hint: it's 4)

```
i = 0                   # Initialization
while (i < 10):         # Condition
    print i             # do_something
    i = i + 1           # Why do we need this?
```

TACC

Exercise:

using a while loop, find the first triple that satisfies:

a*a + b*b = c*c

where b = a + 1, c = a + 2

Python – lists

A list is a sequence, where each element is assigned a position (index)
First position is 0. You can access each position using []
Elements in the list can be of different type

```
mylist1 = [“first item”, “second item”]
mylist2 = [1, 2, 3, 4]
mylist3 = [“first”, “second”, 3]
print mylist1[0], mylist1[1]
print mylist2[0]
print mylist3
print mylist3[0], mylist3[1], mylist3[2]
print mylist2[0] + mylist3[2]
```

TACC

Python – lists

It’s possible to use slicing:

```
print mylist3[0:3]
print mylist3
```

To change the value of an element in a list, simply assign it a new value:

```
mylist3[0] = 10
print mylist3
```

Python – lists

There's a function that returns the number of elements in a list

```
len(mylist2)
```

Check if a value exists in a list:

```
1 in mylist2
```

Delete an element

```
len(mylist2)
del mylist2[0]
print mylist2
```

Iterate over the elements of a list:

```
for x in mylist2:
    print x
```

TACC

Exercise:

create a 3 lists:

one list, x, holding numbers going from 0 to 2*pi, in steps of .01

one list, y1, holding x*x

one list, y2, holding x*x*x

write these out to a file with the format:

x, y1, y2

Python – lists

There are more functions

```
max(mylist), min(mylist)
```

It’s possible to add new elements to a list:

```
my_list.append(new_item)
```

We know how to find if an element exists, there’s a way to return the position of that element:

```
my_list.index(item)
```

Or how many times a given item appears in the list:

```
my_list.count(item)
```

TACC

Python – user defined functions

User-defined functions are reusable code blocks; they only need to be written once, then they can be used multiple times. They can even be used in other applications, too.
These functions are very useful, from writing common utilities to specific business logic. These functions can also be modified per requirement.
The code is usually well organized, easy to maintain, and developer-friendly.
As user-defined functions can be written independently, the tasks of a project can be distributed for rapid application development.
A well-defined and thoughtfully written user-defined function can ease the application development process.

Python – user defined functions

Step 1: Declare the function with the keyword def followed by the function name.

Step 2: Write the arguments inside the opening and closing parentheses of the function, and end the declaration with a colon.

Step 3: Add the program statements to be executed.

Step 4: End the function with/without return statement.

Python – user defined functions

```
def userDefFunction (arg1, arg2, arg3 ...):
    program statement1
    program statement3
    program statement3
    ....
   return;
```

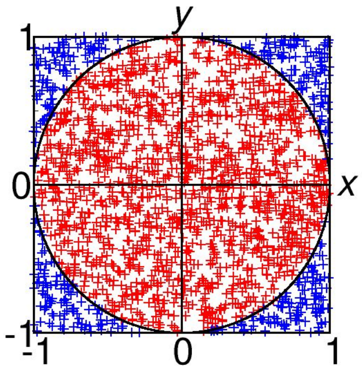
Monte Carlo Pi

Sequential Algorithm

A Monte Carlo algorithm for approximating π uniformly generates the points in the square [-1, 1] x [-1, 1]. Then it counts the points which lie in the inside of the unit circle.

Sequential Algorithm

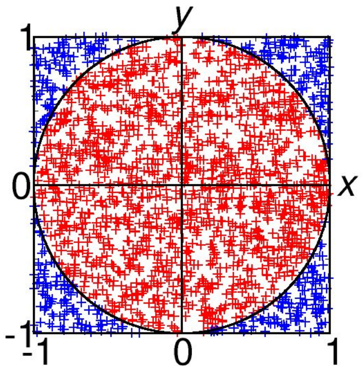
A Monte Carlo algorithm for approximating π uniformly generates the points in the square [-1, 1] x [-1, 1]. Then it counts the points which lie in the inside of the unit circle.



Sequential Algorithm

An approximation of π is then computed by the following formula:

$$4 * \frac{number \quad of \quad points \quad inside}{total \quad number \quad of \quad points}$$



Linear Algebra

Applications

- Matrices in Engineering, such as a line of springs.
- Graphs and Networks, such as analyzing networks.
- Markov Matrices, Population, and Economics, such as population growth.
- Linear Programming, the simplex optimization method.
- Fourier Series: Linear Algebra for functions, used widely in signal processing.
- Linear Algebra for statistics and probability, such as least squares for regression.
- Computer Graphics, such as the various translation, rescaling and rotation of images.

Linear Algebra

Linear algebra is about linear combinations.

Using math on columns of numbers called vectors and arrays of numbers called matrices to create new columns and arrays of numbers.

Linear algebra is the study of lines and planes, vector spaces and mappings that are required for linear transforms.

Linear Algebra

Linear algebra is the mathematics of data.

Matrices and vectors are the language of data.

Let's look at the following:

```
y = 4 * x + 1
```

describes a line on a two-dimensional graph

Linear Algebra

Linear algebra is the mathematics of data.

Matrices and vectors are the language of data.

Let's look at the following:

```
y = 0.1 * x1 + 0.4 * x2
y = 0.3 * x1 + 0.9 * x2
```

line up a system of equations with the same form with two or more unknowns

TACC

Linear Algebra

Linear algebra is the mathematics of data.

Matrices and vectors are the language of data.

Let's look at the following:

```
1 = 0.1 * x1 + 0.4 * x2
3 = 0.3 * x1 + 0.9 * x2
```

line up a system of equations with the same form with two or more unknowns

Linear Algebra

Linear algebra is the mathematics of data.

Matrices and vectors are the language of data.

Let's look at the following, Ax = b :

```
5 = 0.1 * x1 + 0.4 * x2 + x3
10 = 0.3 * x1 + 0.9 * x2 + 2.0 * x3
3 = 0.2 * x1 + 0.3 * x2 - .5 * x3
```

Is there a x1, x2, x3 that solves this system?

TACC

Linear Algebra

Gaussian Elimination

The goals of Gaussian elimination are to make the upper-left corner element a 1

use elementary row operations to get 0s in all positions underneath that first 1

get 1s for leading coefficients in every row diagonally from the upper-left to lower-right corner, and get 0s beneath all leading coefficients.

you eliminate all variables in the last row except for one, all variables except for two in the equation above that one, and so on and so forth to the top equation, which has all the variables. Then use back substitution to solve for one variable at a time by plugging the values you know into the equations from the bottom up..

TACC

Linear Algebra

Gaussian Elimination, Rules

- **You can multiply any row by a constant (other than zero).**
- $-2r_3 \rightarrow r_3$
- **You can switch any two rows.**
- $r_1 \leftrightarrow r_2$
- **You can add two rows together.**
- $r_1 + r_2 \rightarrow r_2$

Linear Algebra

Transpose

A defined matrix can be transposed, which creates a new matrix with the number of columns and rows flipped.

This is denoted by the superscript “T” next to the matrix.

An invisible diagonal line can be drawn through the matrix from top left to bottom right on which the matrix can be flipped to give the transpose.

Linear Algebra

Inversion

Matrix inversion is a process that finds another matrix that when multiplied with the matrix, results in an identity matrix.

Given a matrix A, find matrix B, such that AB or BA = In.

The operation of inverting a matrix is indicated by a -1 superscript next to the matrix; for example, A^-1. The result of the operation is referred to as the inverse of the original matrix; for example, B is the inverse of A.

Linear Algebra

Trace

A trace of a square matrix is the sum of the values on the main diagonal of the matrix (top-left to bottom-right).

Linear Algebra

Determinant

The determinant of a square matrix is a scalar representation of the volume of the matrix.

> *The determinant describes the relative geometry of the vectors that make up the rows of the matrix. More specifically, the determinant of a matrix A tells you the volume of a box with sides given by rows of A.*

— Page 119, No Bullshit Guide To Linear Algebra, 2017

Algorithm

```
double approximatePi(int numSamples)
{
    float x, y;
    int counter = 0;
    for (int s = 0; s != numSamples; s++)
    {
        x = random number between -1, 1;
        y = random number between -1, 1;
```

```
         if (x * x + y * y < 1)
        {
            counter++;
        }
    }

    return 4.0 * counter / numSamples;
}
```

Google to see what command in Python produces a random number

Linear Algebra

Matrix Rank

The rank of a matrix is the estimate of the number of linearly independent rows or columns in a matrix.

Linear Algebra - Matrix Arithmetic

Matrix Addition

Two matrices with the same dimensions can be added together to create a new third matrix.

```
C = A + BC[0,0] = A[0,0] + B[0,0]
```

```
C[1,0] = A[1,0] + B[1,0]
C[2,0] = A[2,0] + B[2,0]
C[0,1] = A[0,1] + B[0,1]
C[1,1] = A[1,1] + B[1,1]
C[2,1] = A[2,1] + B[2,1]
```

Linear Algebra - Matrix Arithmetic

Matrix Subtraction

Similarly, one matrix can be subtracted from another matrix with the same dimensions.

```
C = A - B
```

```
C[0,0] = A[0,0] - B[0,0]
C[1,0] = A[1,0] - B[1,0]
C[2,0] = A[2,0] - B[2,0]
C[0,1] = A[0,1] - B[0,1]
C[1,1] = A[1,1] - B[1,1]
C[2,1] = A[2,1] - B[2,1]
```

Linear Algebra - Matrix Arithmetic

Matrix Multiplication (Hadamard Product)

Two matrices with the same size can be multiplied together, and this is often called element-wise matrix multiplication or the Hadamard product.

It is not the typical operation meant when referring to matrix multiplication, therefore a different operator is often used, such as a circle “o”.

```
C = A o B
C[0,0] = A[0,0] * B[0,0]
C[1,0] = A[1,0] * B[1,0]
C[2,0] = A[2,0] * B[2,0]
C[0,1] = A[0,1] * B[0,1]
C[1,1] = A[1,1] * B[1,1]
C[2,1] = A[2,1] * B[2,1]
```

TACC

Linear Algebra - *Matrix Arithmetic*

Matrix Division

One matrix can be divided by another matrix with the same dimensions.

```
C = A / B
C[0,0] = A[0,0] / B[0,0]
C[1,0] = A[1,0] / B[1,0]
C[2,0] = A[2,0] / B[2,0]
C[0,1] = A[0,1] / B[0,1]
C[1,1] = A[1,1] / B[1,1]
C[2,1] = A[2,1] / B[2,1]
```

Linear Algebra - Matrix Arithmetic

Matrix-Matrix Multiplication (Dot Product)

Matrix multiplication, also called the matrix dot product is more complicated than the previous operations and involves a rule as not all matrices can be multiplied together.

> *One of the most important operations involving matrices is multiplication of two matrices. The matrix product of matrices A and B is a third matrix C. In order for this product to be defined, A must have the same number of columns as B has rows. If A is of shape $m \times n$ and B is of shape $n \times p$, then C is of shape $m \times p$.*

— Page 34, [Deep Learning](), 2016.

TACC

Linear Algebra - *Matrix Arithmetic*

Matrix-Matrix Multiplication (Dot Product)

```
          a11, a12
A =       a21, a22
          a31, a32

          b11, b12
B =       b21, b22

          a11 * b11  +  a12 * b21, a11 * b12  +  a12 * b22
C =       a21 * b11  +  a22 * b21, a21 * b12  +  a22 * b22
          a31 * b11  +  a32 * b21, a31 * b12  +  a32 * b22
```

TACC

Numerical Linear Algebra, Two Different Approaches

- Solve $Ax = b$
- Direct methods:
  - Deterministic
  - Exact up to machine precision
  - Expensive (in time and space)
- Iterative methods:
  - Only approximate
  - Cheaper in space and (possibly) time
  - Convergence not guaranteed

Iterative Methods

Choose any x_0 and repeat

$$x^{k+1} = Bx^k + c$$

until

$$\|x^{k+1} - x^k\|_2 < \epsilon$$

or until

$$\frac{\|x^{k+1} - x^k\|_2}{\|x^k\|} < \epsilon$$

Example of Iterative Solution

Example system

$$\begin{pmatrix} 10 & 0 & 1 \\ 1/2 & 7 & 1 \\ 1 & 0 & 6 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 21 \\ 9 \\ 8 \end{pmatrix}$$

with solution (2,1,1)

Suppose you know (physics) that solution components are roughly the same size, and observe the dominant size of the diagonal, then

$$\begin{pmatrix} 10 & & \\ & 7 & \\ & & 6 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 21 \\ 9 \\ 8 \end{pmatrix}$$

might be a good approximation: solution (2.1, 9/7, 8/6)

Iterative Example

Example system

$$\begin{pmatrix} 10 & 0 & 1 \\ 1/2 & 7 & 1 \\ 1 & 0 & 6 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 21 \\ 9 \\ 8 \end{pmatrix}$$

with solution (2,1,1)

Also easy to solve:

$$\begin{pmatrix} 10 & & \\ 1/2 & 7 & \\ 1 & 0 & 6 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 21 \\ 9 \\ 8 \end{pmatrix}$$

with solution (2.1, 7.95/7, 5.9/6)

Iterative Example

- Instead of solving $Ax = b$ we solved $L\tilde{x} = b.$
- Look for the missing part: $\tilde{x} = x + \Delta x$, then $A\Delta x = A\tilde{x} - b \equiv r$
- Solve again $L\widetilde{\Delta x} = r$ and update $\tilde{\tilde{x}} = \tilde{x} - \widetilde{\Delta x}$

iteration	1	2	3
x_1	2.1000	2.0017	2.000028
x_2	1.1357	1.0023	1.000038
x_3	0.9833	0.9997	0.999995

- Two decimals per iteration. *This is not typical*
- Exact system solving: $O(n^3)$ cost; iteration: $O(n^2)$ per iteration. Potentially cheaper if the number of iterations is low.

TACC

Abstract Presentation

- To solve $Ax = b$; too expensive; suppose $K \approx A$ and solving $Kx = b$ is possible
- Define $Kx_0 = b$, then error correction $x_0 = x + e_0$, and $A(x_0 - e_0) = b$
- so $Ae_0 = Ax_0 - b = r_0$; this is again unsolvable, so
- $K\tilde{e}_0$ and $x_1 = x_0 - \tilde{e}_0$
- Now iterate: $e_1 = x_1 - x$, $Ae_1 = Ax_1 - b = r_1$ et cetera

Error Analysis

- One step

$$
\begin{aligned}
r_1 &= Ax_1 - b = A(x_0 - \tilde{e}_0) - b && (2)\\
&= r_0 - AK^{-1} r_0 && (3)\\
&= (I - AK^{-1}) r_0 && (4)
\end{aligned}
$$

- Inductively:

$$r_n = (I - AK^{-1})^n r_0 \text{ so } r_n \downarrow 0 \text{ if } |\lambda(I - AK^{-1})| < 1$$

- Geometric reduction (or amplification!)
- This is 'stationary iteration': every iteration step the same. Simple analysis, limited applicability

Computationally

If $A = K - N$

then $Ax = b \Longrightarrow Kx = Nx + b \Longrightarrow Kx_{i+1} = Nx_i + b$

(because $Kx = Nx + b$ is a "fixed point" of an iteration)

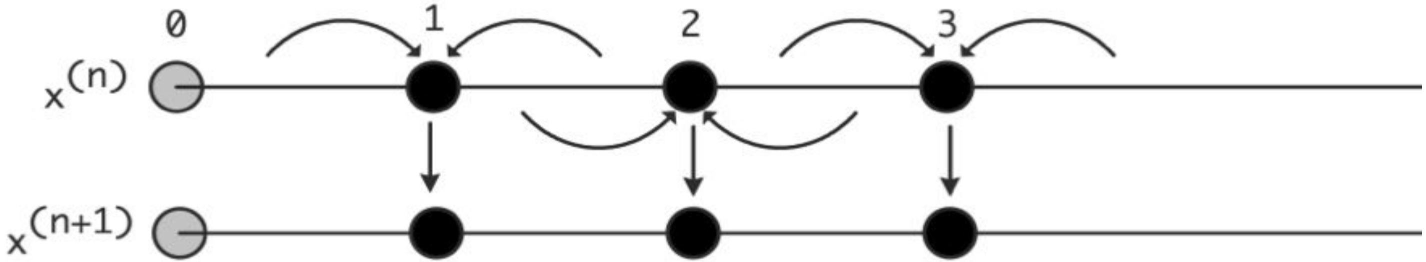
Equivalent to the above, and you don't actually need to form the residual

TACC

Choice of K

- The closer K is to A, the faster the convergence
- Diagonal and lower triangular choice mentioned above: let $A = D_A + L_A + U_A$ be a splitting into diagonal, lower triangular, upper triangular part, then
- Jacobi method: $K = D_A$ (diagonal part),
- Gauss-Seidel method: $K = D_A + L_A$ (lower triangle, including diagonal)
- SOR method: $$K = \omega D_A + L_A$$

Jacobi in Pictures



Jacobi Method

Given a square system of *n* linear equations:

$$A\mathbf{x} = \mathbf{b}$$

where:

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}, \qquad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \qquad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}.$$

Jacobi Method

Then *A* can be decomposed into a diagonal component *D*, and the remainder *R*:

$$A = D + R \qquad \text{where} \qquad D = \begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{bmatrix} \text{ and } R = \begin{bmatrix} 0 & a_{12} & \cdots & a_{1n} \\ a_{21} & 0 & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & 0 \end{bmatrix}.$$

Jacobi Method

The solution is then obtained iteratively via

$$\mathbf{x}^{(k+1)} = D^{-1}(\mathbf{b} - R\mathbf{x}^{(k)}),$$

where $\mathbf{x}^{(k)}$ is the kth approximation or iteration of $\mathbf{x}$ and $\mathbf{x}^{(k+1)}$ is the next or $k+1$ iteration of $\mathbf{x}$. The element-based formula is thus:

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j\neq i} a_{ij}x_j^{(k)}\right), \quad i = 1,2,\ldots,n.$$

The computation of $x_i^{(k+1)}$ requires each element in $\mathbf{x}^{(k)}$ except itself. Unlike the Gauss–Seidel method, we can't overwrite $x_i^{(k)}$ with $x_i^{(k+1)}$, as that value will be needed by the rest of the computation. The minimum amount of storage is two vectors of size n.

Jacobi Method

Algorithm.

- Choose your initial guess, x[0]
- Start iterating, k=0
  - While not converged do
    - Start your i-loop (for i = 1 to n)
      - sigma = 0
        - Start your j-loop (for j = 1 to n)
          - If j not equal to i
            - sigma = sigma + a[i][j] * $x[j]_k$
        - End j-loop
      - $x[i]_k$ = (b[i] – sigma)/a[i][i]
    - End i-loop
  - Check for convergence
- Iterate k, ie. k = k+1

What about the Lower and Upper Triangles?

If we write D, L, and U for the diagonal, strict lower triangular and strict upper triangular and parts of A, respectively,

$$D=\begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & a_{nn} \end{bmatrix}, \quad L=\begin{bmatrix} 0 & 0 & \cdots & 0 \\ a_{21} & 0 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ a_{n1} & \cdots & a_{nn-1} & 0 \end{bmatrix} \text{ and } U=\begin{bmatrix} 0 & a_{12} & \cdots & a_{1n} \\ 0 & 0 & \ddots & \vdots \\ \vdots & \ddots & \ddots & a_{n-1\,n} \\ 0 & \cdots & 0 & 0 \end{bmatrix},$$

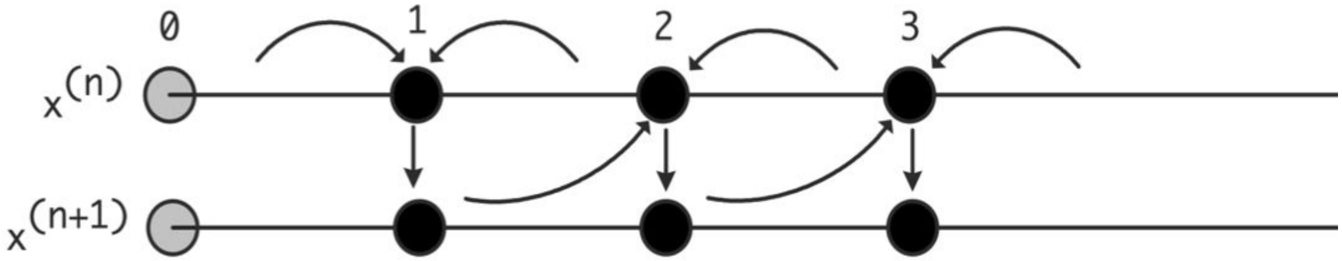
then Jacobi's Method can be written in matrix-vector notation as

$$D\boldsymbol{x}^{(k+1)} + (L+U)\boldsymbol{x}^{(k)} = \boldsymbol{b}$$

so that

$$\boldsymbol{x}^{(k+1)} = D^{-1}[(-L-U)\boldsymbol{x}^{(k)} + \boldsymbol{b}].$$

GS in Pictures



Gauss-Seidel

$K = D_A + L_A$

$Ax=b \Rightarrow (D_A+L_A+U_A)x=b$

$(D_A +L_A)x^{k+1}= -U_A x^k + b$

$\{D_A\}_{ii}=a_{ii}$ $\{U_A \text{ or } L_A\}_{ij}=a_{ij}$ $i\neq j$

Algorithm:

for k = 1, ... until convergence, do:

for i = 1 ... n:

$$//a_{ii}x_i^{(k+1)} + \sum_{j<i} a_{ij}x_j^{(k+1)}) = \sum_{j>i} a_{ij}x_j^{(k)} + b_i \Rightarrow$$
$$x_i^{(k+1)} = a_{ii}^{-1}(-\sum_{j<i} a_{ij}x_j^{(k+1)}) - \sum_{j>i} a_{ij}x_j^{(k)} + b_i)$$

Implementation:

for k = 1, ... until convergence, do:

for i = 1 ... n:

$$x_i = a_{ii}^{-1}(-\sum_{j\neq i} a_{ij}x_j + b_i)$$

TACC

Gauss-Seidel Method

Given a square system of *n* linear equations:

$$A\mathbf{x} = \mathbf{b}$$

where:

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}, \qquad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \qquad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}.$$

TACC

Gauss-Seidel Method

$$A = L_* + U \qquad \text{where} \qquad L_* = \begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ a_{21} & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}, \quad U = \begin{bmatrix} 0 & a_{12} & \cdots & a_{1n} \\ 0 & 0 & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix}.$$

The system of linear equations may be rewritten as:

$$L_* \mathbf{x} = \mathbf{b} - U\mathbf{x}$$

Gauss-Seidel Method

It is defined by the iteration

$$L_*\mathbf{x}^{(k+1)} = \mathbf{b} - U\mathbf{x}^{(k)},$$

where $\mathbf{x}^{(k)}$ is the kth approximation or iteration of $\mathbf{x}$, $\mathbf{x}^{k+1}$ is the next or $k + 1$ iteration of $\mathbf{x}$, and the matrix A is decomposed into a lower triangular component L_*, and a strictly upper triangular component U: $A = L_* + U$.[2]

Which gives us: $$\mathbf{x}^{(k+1)} = L_*^{-1}(\mathbf{b} - U\mathbf{x}^{(k)}).$$

However, by taking advantage of the triangular form of L_*, the elements of $\mathbf{x}^{(k+1)}$ can be computed sequentially using forward substitution:

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j<i} a_{ij}x_j^{(k+1)} - \sum_{j>i} a_{ij}x_j^{(k)}\right), \quad i,j = 1,2,\ldots,n.$$

Gauss-Seidel Method

Algorithm:

- Choose your initial guess, theta[0]
- While not converged do:
    - Start your i-loop (for i = 1 to n)
        - sigma = 0
        - Start your j-loop (for j = 1 to n)
            - If j not equal to i
                - sigma = sigma + a[i][j] * theta[j]
            - End j-loop
            - theta[i] = (b[i] – sigma)/a[i][i]
        - End i-loop
    - Check for convergence
- iterate

Stopping Tests

When to stop converging? Can size of the error be guaranteed?

- Direct tests on error $e_n = x - x_n$ impossible; two choices
- Relative change in the computed solution small:

$$\|x_{n+1} - x_n\| / \|x_n\| < \epsilon$$

- Residual small enough:

$$\|r_n\| = \|Ax_n - b\| < \epsilon$$

Without proof: both imply that the error is less than some other ϵ'

Python – NumPy

"Numerical Python"

open source extension module for Python
provides fast precompiled functions for mathematical and numerical routines
adds powerful data structures for efficient computation of multi-dimensional arrays and matrices.

***NumPy**, First Steps*

Let build a simple list, turn it into a numpy array and perform some simple math.

```
import numpy as np
cvalues = [25.3, 24.8, 26.9, 23.9]
C = np.array(cvalues)
print(C)
```

NumPy, First Steps

Let build a simple list, turn it into a numpy array and perform some simple math.

```
print(C * 9 / 5 + 32)
```

vs.

```
fvalues = [ x*9/5 + 32 for x in cvalues]
print(fvalues)
```

TACC

NumPy, Cooler things

```
import time
size_of_vec = 1000
def pure_python_version():
    t1 = time.time()
    X = range(size_of_vec)
    Y = range(size_of_vec)
    Z = []
    for i in range(len(X)):
        Z.append(X[i] + Y[i])
    return time.time() - t1
```

```
def numpy_version():
    t1 = time.time()
    X = np.arange(size_of_vec)
    Y = np.arange(size_of_vec)
    Z = X + Y
    return time.time() - t1
```

NumPy, Cooler things

Let's see which is faster.

```
t1 = pure_python_version()
t2 = numpy_version()
print(t1, t2)
```

NumPy, Multi-Dimension Arrays

```
A = np.array([ [3.4, 8.7, 9.9],
               [1.1, -7.8, -0.7],
               [4.1, 12.3, 4.8]])
print(A)
print(A.ndim)



B = np.array([ [[111, 112], [121, 122]],
               [[211, 212], [221, 222]],
               [[311, 312], [321, 322]] ])
print(B)
print(B.ndim)
```

NumPy, Multi-Dimension Arrays

The shape function:

```
x = np.array([ [67, 63, 87],
               [77, 69, 59],
               [85, 87, 99],
               [79, 72, 71],
               [63, 89, 93],
               [68, 92, 78]])
print(np.shape(x))
```

TACC

NumPy, Multi-Dimension Arrays

The shape function can also *change* the shape:

```
x.shape = (3, 6)
print(x)


x.shape = (2, 9)
print(x)
```

NumPy, Multi-Dimension Arrays

A couple more examples of shape:

```
x = np.array(42)
print(np.shape(x))

B = np.array([ [[111, 112], [121, 122]],
               [[211, 212], [221, 222]],
               [[311, 312], [321, 322]] ])
print(B.shape)
```

TACC

NumPy, Multi-Dimension Arrays

indexing:

```
F = np.array([1, 1, 2, 3, 5, 8, 13, 21])

# print the first element of F, i.e. the element with the index 0
print(F[0])

# print the last element of F
print(F[-1])

B = np.array([ [[111, 112], [121, 122]],
               [[211, 212], [221, 222]],
               [[311, 312], [321, 322]] ])
print(B[0][1][0])
```

TACC

NumPy, Multi-Dimension Arrays

slicing:

```
A = np.array([
[11,12,13,14,15],
[21,22,23,24,25],
[31,32,33,34,35],
[41,42,43,44,45],
[51,52,53,54,55]])

print(A[:3,2:])


print(A[3:,:])
```

TACC

NumPy, Multi-Dimension Arrays

identity function

```
np.identity(4)
```

NumPy, By Example

The example we will consider is a very simple (read, trivial) case of solving the 2D Laplace equation using an iterative finite difference scheme (four point averaging, Gauss-Seidel or Gauss-Jordan). The formal specification of the problem is as follows. We are required to solve for some unknown function u(x,y) such that $\nabla 2u = 0$ with a boundary condition specified. For convenience the domain of interest is considered to be a rectangle and the boundary values at the sides of this rectangle are given.

```
def TimeStep(self, dt=0.0):
     """Takes a time step using straight forward Python loops."""
     g = self.grid
     nx, ny = g.u.shape
     dx2, dy2 = g.dx**2, g.dy**2
     dnr_inv = 0.5/(dx2 + dy2)
     u = g.u
     err = 0.0
     for i in range(1, nx-1):
         for j in range(1, ny-1):
             tmp = u[i,j]
             u[i,j] = ((u[i-1, j] + u[i+1, j])*dy2 +
                      (u[i, j-1] + u[i, j+1])*dx2)*dnr_inv
             diff = u[i,j] - tmp
             err += diff*diff
     return numpy.sqrt(err)
```

NumPy, By Example

The example we will consider is a very simple (read, trivial) case of solving the 2D Laplace equation using an iterative finite difference scheme (four point averaging, Gauss-Seidel or Gauss-Jordan). The formal specification of the problem is as follows. We are required to solve for some unknown function u(x,y) such that $\nabla 2u = 0$ with a boundary condition specified. For convenience the domain of interest is considered to be a rectangle and the boundary values at the sides of this rectangle are given.

```
def numericTimeStep(self, dt=0.0):
    """Takes a time step using a NumPy expression."""
    g = self.grid
    dx2, dy2 = g.dx**2, g.dy**2
    dnr_inv = 0.5/(dx2 + dy2)
    u = g.u
    g.old_u = u.copy() # needed to compute the error.

    # The actual iteration
    u[1:-1, 1:-1] = ((u[0:-2, 1:-1] + u[2:, 1:-1])*dy2 +
                     (u[1:-1,0:-2] + u[1:-1, 2:])*dx2)*dnr_inv

    return g.computeError()
```

TACC

NumPy, By Example

Jacobi

```
Algorithm.

* Find D, the Diagonal of of A : diag(A)
* Find R, the Remainder of A - D : A - diagflat(A)

* Choose your initial guess, x[0]
    * Start iterating, k=0
        * While not converged do
           * Start your i-loop (for i = 1 to n)
               * sigma = 0
                * Start your j-loop (for j = 1 to n)
                   * If j not equal to i
                       * sigma = sigma + a[i][j] * x[j]k
                 * End j-loop
               * x[i]k = (b[i] – sigma)/a[i][i] : x = (b - dot(R,x)) / D
           * End i-loop
        * Check for convergence
    * Iterate k, ie. k = k+1
```

Pandas, What is it?

A software library written for the Python for data manipulation and analysis. In particular, it offers data structures and operations for manipulating numerical tables and time series

***Pandas**, First Steps*

Let's create a simple data set, and see what Pandas can do.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

***Pandas**, First Steps*

Let's create a simple data set, and see what Pandas can do.

```
s = pd.Series([1,3,5,np.nan,6,8])
s
```

Pandas, First Steps

Let's create a simple data set, and see what Pandas can do.

```
dates = pd.date_range('20180101', periods=6)
dates
```

TACC

Pandas, First Steps

Let's create a simple data set, and see what Pandas can do.

```
df = pd.DataFrame(np.random.randn(6,4),
 index=dates, columns=list('ABCD'))
df
```

***Pandas**, First Steps*

Let's create a simple data set, and see what Pandas can do.

```
df2 = pd.DataFrame({ 'A' : 1.,'B' :
 pd.Timestamp('20130102'),'C' :
 pd.Series(1,index=list(range(4)),dtype='float32'),'D' :
 np.array([3] * 4,dtype='int32'),'E' :
 pd.Categorical(["test","train","test","train"]),'F' :
 'foo' })

df2
```

TACC

Pandas, Viewing Data

Some common/useful functions

```
df.head()
df.tail(3)
df.index
df.columns
df.values
df.describe()
df.T
df.sort_index(axis=1, ascending=False)
df.sort_values(by='B')
```

Pandas, Selecting Data by Label

Some common/useful functions

```
df['A'])
df[0:3]
df['20130102':'20130104']
df.loc[dates[0]]
df.loc[:,['A','B']]
df.loc['20130102':'20130104',['A','B']]
df.loc['20130102',['A','B']]
df.loc[dates[0],'A']
```

TACC

Pandas, Selecting Data by Position

Some common/useful functions

```
df.iloc[3]
df.iloc[3:5,0:2]
df.iloc[[1,2,4],[0,2]]
df.iloc[1:3,:]
df.iloc[:,1:3]
df.iloc[1,1]
df.iat[1,1]
```

TACC

Pandas, Summary of Features

Pandas allow for:

Boolean Indexing
Statistical Operations
Histogramming
Merging Data
SQL Style Joins
SQL Style Appends
SQL Style Grouping
Reshaping
Pivoting
and more!

Pandas, CSV Files

manipulating CSV files.

```
ts = pd.Series(np.random.randn(1000), index=pd.date_range('1/1/2000',
 periods=1000))
ts = ts.cumsum()


df = pd.DataFrame(np.random.randn(1000, 4), index=ts.index,columns=['A',
 'B', 'C', 'D'])
df = df.cumsum()

df.to_csv('foo.csv')
pd.read_csv('foo.csv')
```

Matplotlib, ***What is it?***

It’s a graphing library for Python. It has a nice collection of tools that you can use to create anything from simple graphs, to scatter plots, to 3D graphs. It is used heavily in the scientific Python community for data visualisation.

Matplotlib, First Steps

Let's plot a simple sin wave from 0 to 2 pi.

First let's, get our code started by importing the necessary modules.

```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
```

Matplotlib, First Steps

Let's add the following lines, we're setting up x as an array of 50 elements going from 0 to 2*pi

```
x = np.linspace(0, 2 * np.pi, 50)
plt.plot(x, np.sin(x))
plt.show() # Show the graph.
```

Let's run our cell!

Matplotlib, a bit more interesting

Let's plot another curve on the axis

```
plt.plot(x, np.sin(x),
         x, np.sin(2 * x))
plt.show()
```

Let's run our cell!

TACC

Matplotlib, *a bit more interesting*

Let's see if we can make the plots easier to read

```
plt.plot(x, np.sin(x), 'r-o',
         x, np.cos(x), 'g--')
plt.show()
```

Let's run this cell!

Matplotlib, a bit more interesting

```
Colors:
Blue – ‘b’
Green – ‘g’
Red – ‘r’
Cyan – ‘c’
Magenta – ‘m’
Yellow – ‘y’
Black – ‘k’ (‘b’ is taken by blue so the last letter is used)
White  – ‘w’
```

TACC

Matplotlib, a bit more interesting

```
Lines:
    Solid Line – ‘-‘
    Dashed – ‘–‘
    Dotted – ‘.’
    Dash-dotted – ‘-:’
Often Used Markers:
    Point – ‘.’
    Pixel – ‘,’
    Circle – ‘o’
    Square – ‘s’
    Triangle – ‘^’
```

Matplotlib, **Subplots**

Let's split the plots up into subplots

```
plt.subplot(2, 1, 1) # (row, column, active area)
plt.plot(x, np.sin(x), 'r')
plt.subplot(2, 1, 2)
plt.plot(x, np.cos(x), 'g')
plt.show()
```

using the subplot() function, we can plot two graphs at the same time within the same "canvas". Think of the subplots as "tables", each subplot is set with the number of rows, the number of columns, and the active area, the active areas are numbered left to right, then up to down.

Matplotlib, Scatter Plots

Let's take our sin curve, and make it a scatter plot

```
y = np.sin(x)
plt.scatter(x,y)
plt.show()
```

call the scatter() function and pass it two arrays of x and y coordinates.

TACC

Matplotlib, add a touch of color

Let's mix things up, using random numbers and add a colormap to a scatter plot

```
x = np.random.rand(1000)
y = np.random.rand(1000)
size = np.random.rand(1000) * 50
color = np.random.rand(1000)
plt.scatter(x, y, size, color)
plt.colorbar()
plt.show()
```

TACC

Matplotlib, add a touch of **color**

Let's see what we added, and where that takes us

```
...
plt.scatter(x, y, size, color)
plt.colorbar()
...
```

We brought in two new parameters, size and color. Which will varies the diameter and the color of our points. Then adding the colorbar() gives us nice color legend to the side.

Matplotlib, Histograms

A histogram is one of the simplest types of graphs to plot in Matplotlib. All you need to do is pass the hist() function an array of data. The second argument specifies the amount of bins to use. Bins are intervals of values that our data will fall into. The more bins, the more bars.

```
plt.hist(x, 50)
plt.show()
```

Matplotlib, Adding Labels and Legends

Let's go back to our sin/cos curve example, and add a bit of clarification to our plots

```
x = np.linspace(0, 2 * np.pi, 50)
plt.plot(x, np.sin(x), 'r-x', label='Sin(x)')
plt.plot(x, np.cos(x), 'g-^', label='Cos(x)')
plt.legend() # Display the legend.
plt.xlabel('Rads') # Add a label to the x-axis.
plt.ylabel('Amplitude') # Add a label to the y-axis.
plt.title('Sin and Cos Waves') # Add a graph title.
plt.show()
```

TACC

mpld3, What is it?

An API that merges Matplotlib, the popular Python-based graphing library, and D3js, a popular JavaScript library for creating interactive data visualizations for the web.

mpld3, First Steps

customizing your Jupyter session.

```
pip install --user mpld3
```

TACC

mpld3, Demo

http://mpld3.github.io/index.html

Summary

Using Numpy

build our data

Using Pandas

analyze our data

Animation using Matplotlib

view our data

Interactive Plots using MPLD3

play with our data

Questions? Comments?

charlie@tacc.utexas.edu