

# UNCONSTRAINED OPTIMIZATION: THEORY AND APPLICATION

## PART II: IMPLEMENTATION USING PYTHON

Jesse Quinlan

[jrq2a@virginia.edu](mailto:jrq2a@virginia.edu)

March 13, 2012

# Installation

2

1. Download the Python 3.2 installer for Windows at (choose the 32 bit version--[Python 3.2.2 Windows x86 MSI Installer](http://python.org/getit/))  
<http://python.org/getit/>
2. Run the installer by double clicking on it. Install python to the C:\Python32 directory if the installer requests an installation directory (this should be the default installation directory).
3. Confirm your installation was successful by opening a terminal (go to Start > Run > Type *cmd* > Press enter), type *python*, and press enter. If the interactive shell is not initiated, make sure that C:\Python32 is included in your path variable by typing *PATH C:\Python32;\$PATH\$* and pressing enter.
4. Once you have confirmed Python installed successfully, install the NumPy module by downloading the installer file ([numpy-1.6.1-win32-superpack-python3.2.exe](http://sourceforge.net/projects/numpy/files/NumPy/1.6.1/)) from:  
<http://sourceforge.net/projects/numpy/files/NumPy/1.6.1/>
5. Install by double clicking on the installer file.
6. Confirm installation by typing in the interactive python shell *import numpy*. If you receive no errors, assume installation was successful.

# What is python ?

3

- ❑ Lightweight object-oriented scripting language
- ❑ Open source, freely distributed and supported
- ❑ Interactive shell
- ❑ Interpreted syntax, no compilation/executables
- ❑ Extensible platform useful for a variety of applications (scientific/engineering, web development, scripting, etc)

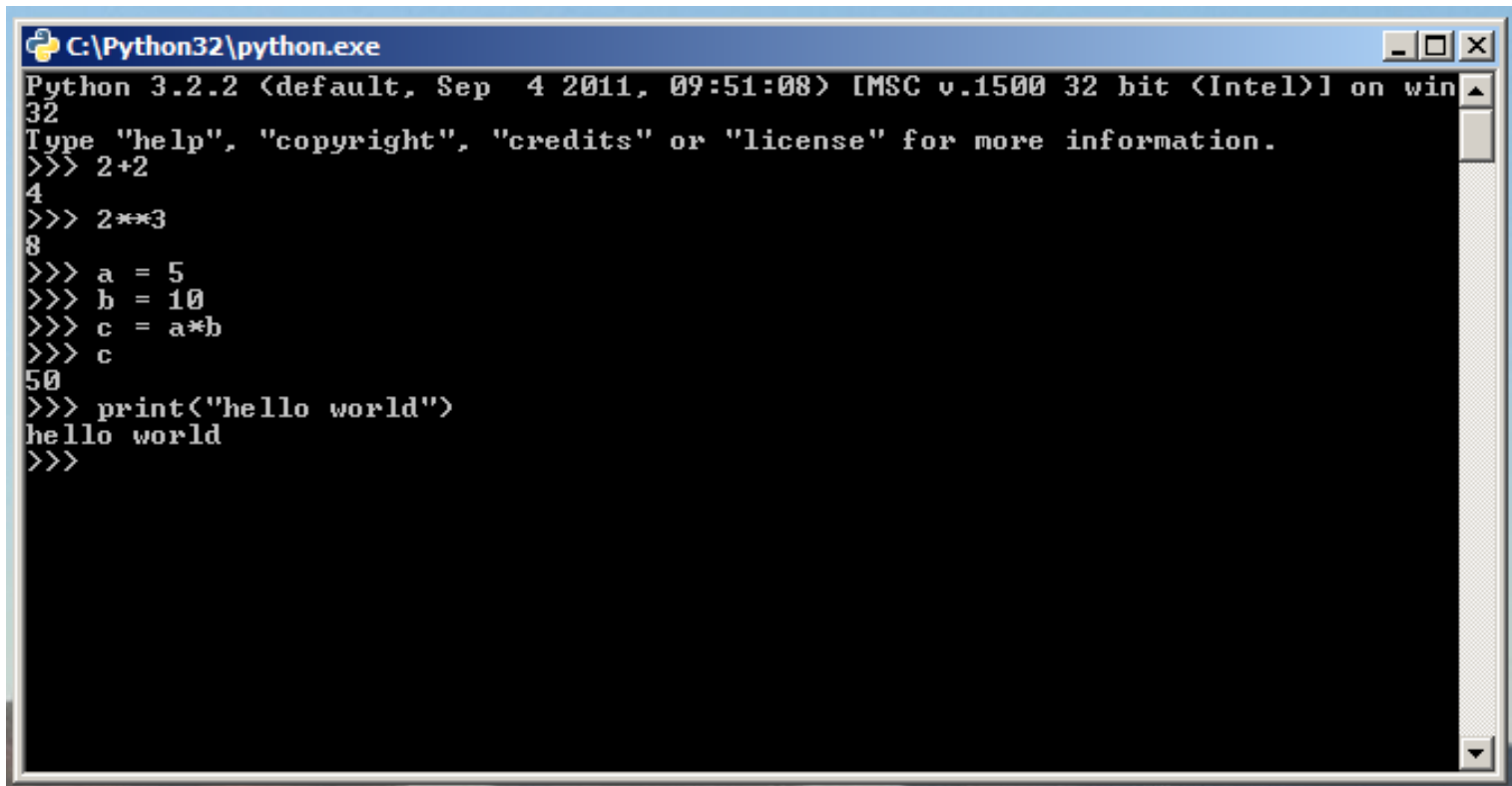
# Python Basics

# The Interactive Shell

5

Two ways to load the interactive shell:

- 1) by typing *python* at the terminal
- 2) Start > Programs > Python > Python (Command Line)

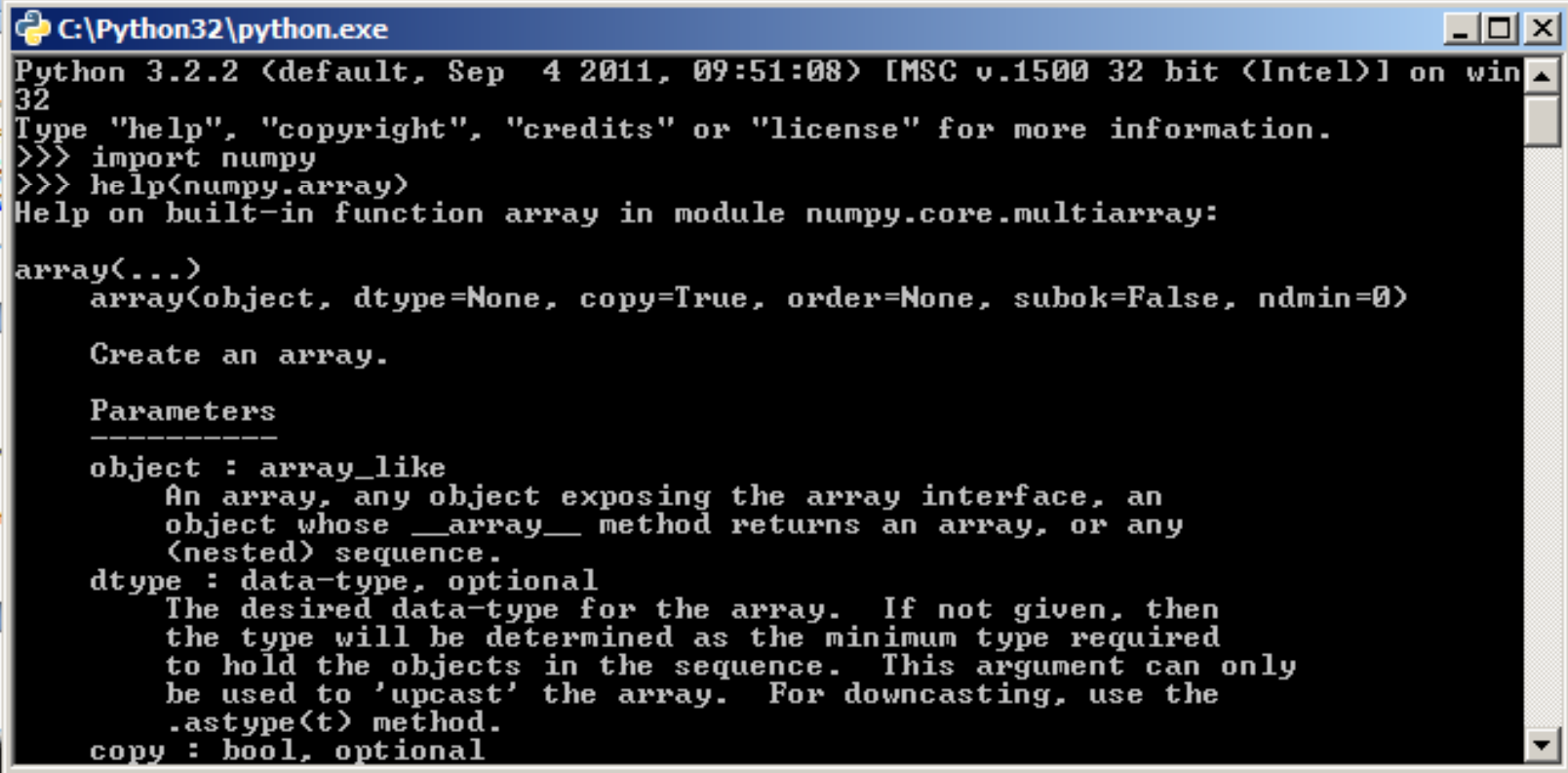


```
C:\Python32\python.exe
Python 3.2.2 <default, Sep 4 2011, 09:51:08> [MSC v.1500 32 bit <Intel>] on win
32
Type "help", "copyright", "credits" or "license" for more information.
>>> 2+2
4
>>> 2**3
8
>>> a = 5
>>> b = 10
>>> c = a*b
>>> c
50
>>> print("hello world")
hello world
>>>
```

# The Interactive Shell

6

- Within the shell, you can import any modules you might use in a script
- Basic commands include:  
    `help()`, `exit()`, `print()`, `import MODULE`



```
C:\Python32\python.exe
Python 3.2.2 (default, Sep  4 2011, 09:51:08) [MSC v.1500 32 bit (Intel)] on win
32
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy
>>> help(numpy.array)
Help on built-in function array in module numpy.core.multiarray:

array(...)
    array(object, dtype=None, copy=True, order=None, subok=False, ndmin=0)

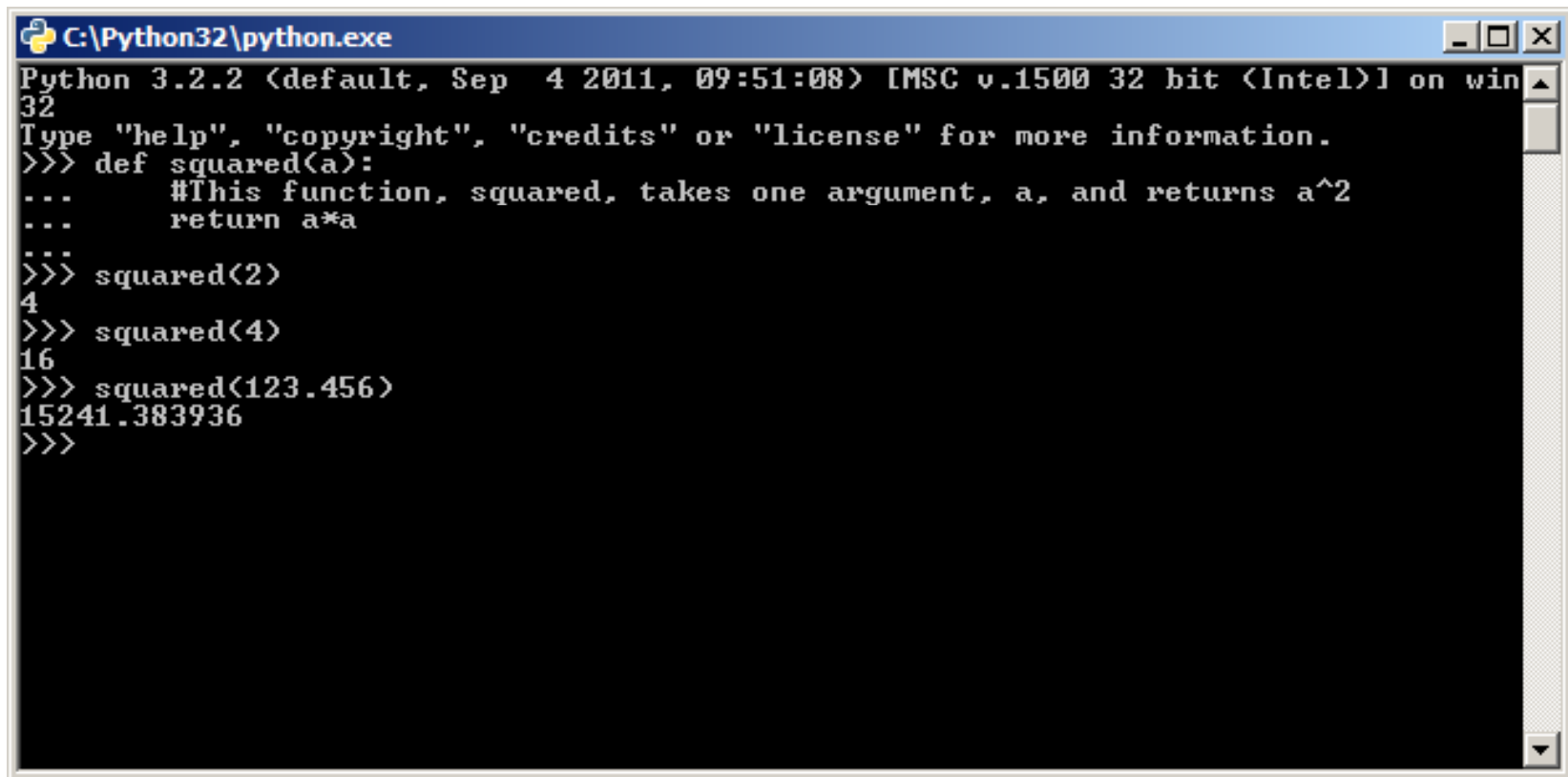
    Create an array.

    Parameters
    -----
    object : array_like
        An array, any object exposing the array interface, an
        object whose __array__ method returns an array, or any
        (nested) sequence.
    dtype : data-type, optional
        The desired data-type for the array.  If not given, then
        the type will be determined as the minimum type required
        to hold the objects in the sequence.  This argument can only
        be used to 'upcast' the array.  For downcasting, use the
        .astype(t) method.
    copy : bool, optional
```

# The Interactive Shell

7

- Within the shell, you can also create functions and input logic and loops just like in a script setting



```
C:\Python32\python.exe
Python 3.2.2 (default, Sep  4 2011, 09:51:08) [MSC v.1500 32 bit (Intel)] on win
32
Type "help", "copyright", "credits" or "license" for more information.
>>> def squared(a):
...     #This function, squared, takes one argument, a, and returns a^2
...     return a*a
...
>>> squared(2)
4
>>> squared(4)
16
>>> squared(123.456)
15241.383936
>>>
```

# Windows Command Line

8

- Programming via the operating system's command line is generally preferred for larger projects and is a valuable skill to have for its speed and extensibility
- Python at the command line is similar to any other programming language you may implement at the command line
- For non-scriptable languages, you might begin by creating a human-readable file containing your code, then compiling this using a language compiler (gcc, intel, etc.) in order to create an executable (binary; .exe, .out, etc.)
- However, since Python is a scripting language, no compilation is necessary. Python interprets a script when executed, allowing for quick testing and implementation.



# Running a Python Script

9

- First you create the script using your favorite text editor (gedit, vim, notepad, etc.), which we will save as *script.py*  
(<http://projects.gnome.org/gedit/>)
- Then open the Windows terminal, and change to the directory containing your *script.py* file.
- Run the script by typing: *python script.py*  
(make sure C:\Python32\ is in your PATH)
- What we put in our *script.py* is coming next.

# Writing a Script

10

- A first script, highlighting the separate regions of the file:

```
1 #This line is a comment, as designated by the leading "#"  
2  
3 #First import any modules you may need  
4 #In this example, we will import the math module and we  
5 #will rename it m for convenience.  
6  
7 import math as m  
8  
9 #Then we will write our program. Here, we will do some  
10 #elementary math using the math library.  
11  
12 a = 2.0  
13 b = 4.0  
14  
15 c = m.exp(a)  
16 d = m.exp(b)  
17  
18 print('exp(',a,') is equal to',c,'\nexp(',b,') is equal to',d)
```

# Implementation of the Golden Section Method

11

- We are going to learn the specific syntax for basic logic and variable manipulation by jumping right in to our implementation of the Golden Section method.
- Our goal is to write a python script, named `golden_numpy.py`, in which we:
  - ▣ Import the *numpy* module
  - ▣ Construct a function titled *fcn*:
    - Input: *x* (scalar or vector)
    - Returns: value of user-defined function
  - ▣ Construct a function titled *gold*:
    - Input: *xlowin*, *xhighin*, *eps*, *xvec*, *svec*
    - Returns: location and value of minimum (or maximum)

# Constructing a Function

# Writing *fcn(x)*

13

- A function in python is very similar to a function in Matlab and other languages—it is designed to take an input and return an output.
- In python the keyword *def* is used to **define** a function.
- For example, we start our *fcn(x)* by writing:

```
def fcn(x):  
    syntax here  
    return output
```

- It is very important to include the tab spacing, since Python uses this formatting in place of *end* statements this is required by python for proper interpretation

# Writing *fcn(x)*

14

- Let's say we are interested in the following function of  $x$ :

$$f(x) = x^4 - 5.0x^3 + 10.0$$

- We can implement this, simply as:

```
def fcn(x):
```

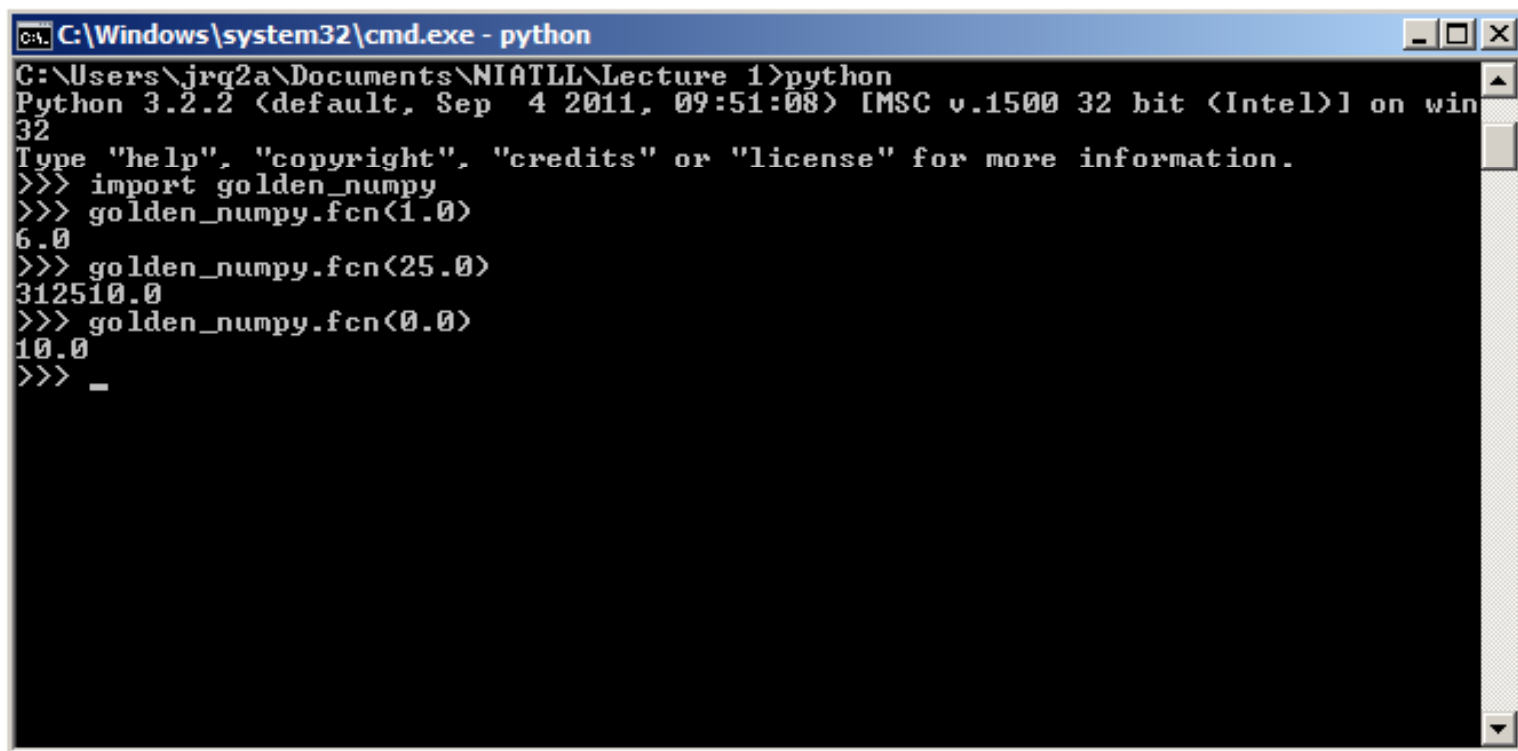
```
    return x**4.0-5.0*x**3.0+10.0
```

- Create and save this file as *golden\_numpy.py*, and we will now refer to this file as our *golden\_numpy* module

# Testing $fcn(x)$

15

- We can now import our new module into the interactive shell and test our  $fcn(x)$ .



```
C:\Windows\system32\cmd.exe - python
C:\Users\jrq2a\Documents\NIATLL\Lecture 1>python
Python 3.2.2 (default, Sep  4 2011, 09:51:08) [MSC v.1500 32 bit (Intel)] on win
32
Type "help", "copyright", "credits" or "license" for more information.
>>> import golden_numpy
>>> golden_numpy.fcn(1.0)
6.0
>>> golden_numpy.fcn(25.0)
312510.0
>>> golden_numpy.fcn(0.0)
10.0
>>> _
```

# Using Variables, Logic, and Modules



# Writing *gold*

17

- Now we are ready to implement our algorithm for the golden section method.
- We will write another function, called *gold*, which will take several input and will return the location of and the value of the minimum (of maximum) of our chosen function (*fcn*)
- Our gold function will have the standard form:  
*def gold(xlowin,xhighin,eps,xvec,svec):*  
*syntax here*  
*return output*

# Writing *gold* | Using Variables

18

- Defining variables requires no special descriptor, like with many programming languages. With Python, you simply use the “=” sign to associate values to variables.
- For example:
  - ▣ `Var1 = 0` [setting Var1 equal to the integer 0]
  - ▣ `Var2 = 1.0` [setting Var2 equal to the float 1.0]
  - ▣ `Var3 = 'var3'` [setting Var3 equal to the string “var3”]

# Writing *gold* | Logic

19

- Python has the same logic features as many programming languages (if, for, while, etc.)
- Python syntax is unlike many other languages
- The form of an *if* statement:

*if Var1 > Var2:*

*syntax*

*elif Var1 < Var2:*

*syntax*

*else:*

*syntax*

- *if* operators:  $<, <=, >, >=, ==, !=$

# Writing *gold* | Logic

20

- The form of a *for* loop:

*for i in [0,1,2,3]:*  
    *syntax*

OR

*for i in range(0,4):*  
    *syntax*

- The form of a *while* loop:

*while count < 100:*  
    *syntax*

- Note that indexing starts at 0 (not 1).

# Writing *gold* | Modules

21

- Modules are stand alone python files that contain functions, objects, classes, etc. that you can import into a script or use interactively.
- In the powell function, we will use the NumPy module, which is useful when working with arrays, but is useful for advanced scientific and engineering analysis
- SciPy (<http://www.scipy.org/>) is another very useful module for advanced scientific work, in addition to Matplotlib (<http://matplotlib.sourceforge.net/>) , which offers plotting tools that have a Matlab feel
- Python provides a few options for importing modules:
  - ▣ The most basic convention for importing a module is: *import MODULE*
  - ▣ Though you may like to import a module and rename it, like we saw earlier. This will save time for longer scripts. The form is: *import MODULE as NICKNAME*
  - ▣ You may also use the convention: *from MODULE import \**  
However, be careful doing this as this doesn't require you to specify the module when using its contents, and you could have imported functions of the same name from different modules.

# Writing *gold* | Modules

22

- Python has many built-in modules for you to call (Math, OS, etc) (<http://docs.python.org/tutorial/modules.html#tut-standardmodules>)
- Note that allocation of array space is not necessary with Python. We can easily append arrays dynamically. This is also true (and extremely useful) for lists—Python’s native “array” substitute (though lists are very different from an array).
- Note however that in the gold function, we only use lists (NumPy arrays will be used in the powell function). Lists are unique to Python, and are essentially heterogeneous arrays. Python doesn’t limit the contents of a list to the same form. Lists are very powerful; however, for applications heavy in computation and matrix math, (homogeneous) arrays are generally preferred (with the use of NumPy , Numeric, or another advanced math module)

# Writing *gold* | Code

23

1. Select initial bounds  $x_1$  and  $x_2$ , and compute  $F(x_1)$  and  $F(x_2)$
2. Initialize  $x_3$  and  $x_4$  using previous formulae, and compute  $F(x_3)$  and  $F(x_4)$
3. Based on required tolerance  $\varepsilon$ , determine total number of iterations  $N$ ; initialize counter  $K = 3$

$$x_3 = (1 - \tau)x_1 + \tau x_2 \quad \frac{x_3 - x_1}{x_2 - x_1} = \frac{x_2 - x_4}{x_2 - x_1} = \tau$$

$$x_4 = \tau x_1 + (1 - \tau)x_2 \quad x_3 - x_1 = x_2 - x_4$$

$$N = \frac{\ln(\varepsilon)}{\ln(1 - \tau)} + 3$$

$$\varphi = \frac{1 + \sqrt{5}}{2} \approx 1.618$$

$$\tau = 1 - (\varphi - 1) = 2 - \varphi \approx 0.382$$

# Writing *gold* | Code

24

## 4. Loop while $K < N$ :

### 1. If $F(x_3) > F(x_4)$

1.  $x_1 \leftarrow x_3$
2.  $F(x_1) \leftarrow F(x_3)$
3.  $x_3 \leftarrow x_4$
4.  $F(x_3) \leftarrow F(x_4)$
5.  $x_4 \leftarrow tx_1 + (1-t)x_2$
6. Compute  $F(x_4)$
7.  $K \leftarrow K+1$

### 2. Else

1.  $x_2 \leftarrow x_4$
2.  $F(x_2) \leftarrow F(x_4)$
3.  $x_4 \leftarrow x_3$
4.  $F(x_4) \leftarrow F(x_3)$
5.  $x_3 \leftarrow (1-t)x_1 + tx_2$
6. Compute  $F(x_3)$
7.  $K \leftarrow K+1$



# Writing *gold* | Initial Variables

25

```
8 #Import math library (functions like ceil, log, sqrt, etc)
9 import math
10 #Import NumPy (for efficient array processing)
11 import numpy as np
12
13 #The user-defined one-dimensional function
14 #def fcn(x):
15 #     return (x**4.0)-5.0*x**3.0+10.0
16
17 #The user-defined multidimensional function
18 def fcn(x):
19     #return
20     #return x[0]+x[1]*x[2]+10.0
21     #return math.sin(x[0])+math.cos(x[1])
22     return x**2.0+1.0
23
24 #This function solves fcn(x) for its local minimum using golden section method
25 def gold(xlowin,xhighin,eps,xvec,svec):
26     #xlen = len(xvec)
27     tau = (3.0-(5.0)**(0.5))/2.0
28     K = 2
29     N = math.ceil(math.log(eps)/math.log(1.0-tau))+3)
30     [flow,fhigh,xlow,xhigh] = [0.0,0.0,0.0,0.0]
31     [f1,f2,x1,x2] = [0.0,0.0,0.0,0.0]
32     [xlow,xhigh] = [xlowin,xhighin]
33     [flow,fhigh] = [fcn(xvec+xlow*svec),fcn(xvec+xhigh*svec)]
34     [x1,x2] = [(1.0-tau)*xlow+tau*xhigh,tau*xlow+(1.0-tau)*xhigh]
35     [f1,f2] = [fcn(xvec+x1*svec),fcn(xvec+x2*svec)]
```

Copyright 2012-20000 Garmann

```

24 #This function solves fcn(x) for its local minimum using golden section
25 def gold(xlowin,xhighin,eps,xvec,svec):
26     #xlen = len(xvec)
27     tau = (3.0-(5.0)**(0.5))/2.0
28     K = 2
29     N = math.ceil(math.log(eps)/math.log(1.0-tau)+3)
30     [flow,fhigh,xlow,xhigh] = [0.0,0.0,0.0,0.0]
31     [f1,f2,x1,x2] = [0.0,0.0,0.0,0.0]
32     [xlow,xhigh] = [xlowin,xhighin]
33     [flow,fhigh] = [fcn(xvec+xlow*svec),fcn(xvec+xhigh*svec)]
34     [x1,x2] = [(1.0-tau)*xlow+tau*xhigh,tau*xlow+(1.0-tau)*xhigh]
35     [f1,f2] = [fcn(xvec+x1*svec),fcn(xvec+x2*svec)]
36     for i in range(K,N):
37         if f1 > f2:
38             [xlow,flow] = [x1,f1]
39             [x1,f1] = [x2,f2]
40             x2 = tau*xlow+(1.0-tau)*xhigh
41             f2 = fcn(xvec+x2*svec)
42         else:
43             [xhigh,fhigh] = [x2,f2]
44             [x2,f2] = [x1,f1]
45             x1 = (1.0-tau)*xlow+tau*xhigh
46             f1 = fcn(xvec+x1*svec)
47
48     return [0.5*(x1+x2),fcn(xvec+0.5*(x1+x2)*svec)]

```

# Testing *gold*

27

- When it comes time to testing your module, it is tempting to go straight to the interactive shell, import your module, and start feeding it input.
- The *smarter* approach would be to create a python script, called *test.py*, for example, in which you import your new module (*golden\_numpy.py*). Let's assume you rename it to *gn* when importing it.
- Now you can call your functions within your *test.py* script as many times as you like by referring to them as *gn.fcn()* and *gn.gold()*.
- To run the *test.py* script, you can go to the command line and simply type *python test.py*

# Testing *gold*

28

- Let's also be a little smarter about defining our function, *fcn(x)*. Currently, *fcn(x)* only includes the single function we used to define it.
- Instead, lets redefine *fcn(x)* to be *fcn(x,wch)* where *wch* is an integer that instructs *fcn* to return one of several functions.
- This can be implemented using a simple if,elif,else construct, as seen on the next slide.
- Let's input the following 1D functions of *x*:
  - ▣ *wch* = 0:  $f(x) = x^2 + 1$
  - ▣ *wch* = 1:  $f(x) = x^4 - 5.0x^3 + 10.0$
  - ▣ *wch* = 2:  $f(x) = x^2$

# Testing *gold*

29

- Keep in mind we also have to add the wch input to the gold function and to the fcn calls within gold.

```
12
13 #The user-defined multidimensional function
14 def fcn(x,wch):
15     if wch == 0:
16         out = x**2.0+1.0
17     elif wch == 1:
18         out = x**4.0-5.0*x**3.0+10.0
19     elif wch == 2:
20         out = x**2.0
21     return out
22
```

# Testing *gold*

30

- A sample test script might look like:

```
1 import golden_numpy_nd as gn
2
3 xlow1 = -10.0
4 xlow2 = -10.0
5 xhigh1 = 10.0
6 xhigh2 = 10.0
7 eps1 = 0.01
8 eps2 = 0.01
9 xvec1 = 1.0
10 xvec2 = 1.0
11 svec1 = 1.0
12 svec2 = 1.0
13
14 out1 = gn.gold(xlow1,xhigh1,eps1,xvec1,svec1,0)
15 out2 = gn.gold(xlow2,xhigh2,eps2,xvec2,svec2,1)
16 out3 = gn.gold(xlow1,xhigh1,eps1,xvec1,svec1,100)
17
18 print('Output for case 1 of gold:', out1, ' for function 0')
19 print('\nOutput for case 2 of gold:', out2, ' for function 1')
20 print('\nOutput for case 3 of gold:', out3, 'for function default')
21
```

# Implementing Powell's Method

# Writing *powell* | Using *gold*

32

- We can use the *gold* function, in addition to the *fcn* function, in our implementation of the *powell* method
  - ▣ Work smarter, not harder
- Let's write another function, at the end of our *golden\_numpy* module, called *powell* that uses the following:
  - ▣ Input: *x0*, *eps*
  - ▣ Returns: the location and value of the minimum (or maximum) of the function *fcn(x)*



# Writing *powell* | N-D Function

33

- Before we start writing *powell*, we should augment the capabilities of our *fcn(x)* so that we can represent multi-dimensional functions.
- For this exercise, we will implement a few functions that we specify with an input parameter.
- We will use a simple *if/elif/else* construct to choose which function we want our routine to use.

# Writing *powell* | N-D Function

34

- We want the following functions available to us:
  - ▣  $X^2 + 1.0$
  - ▣  $X^4 - 5x^3 + 10$
  - ▣  $\text{Sin}(x) + \text{cos}(y)$
  - ▣  $X^2$
- So our code might look something like this when we are done:

# Writing *powell* | N-D Function

35

```
13 #The user-defined multidimensional function
14 def fcn(x,wch):
15     if wch == 0:
16         out = x**2.0+1.0
17     elif wch == 1:
18         out = x**4.0-5.0*x**3.0+10.0
19     elif wch == 2:
20         out = math.sin(x[0])+math.cos(x[1])
21     else:
22         out = x**2.0
23     return out
24
```

- Now make sure to update the gold function to accept the additional input parameter, *wch*, which we use to specify the function.

# Writing *powell* | Structure

36

- Just like before, we start by defining the *powell* function, using:

```
def powell(x0,eps):  
    syntax  
    return
```

- Now we will make use of our imported NumPy module by defining and manipulating arrays

# Writing *powell* | Arrays with NumPy

37

- With NumPy, we can initialize arrays by creating arrays (or vectors) of zeros easily by typing:

*array1 = np.zeros((10,10))*

in which array1 is a 10x10 array of zeros.

A vector of zeros of length 10 is simply:

*array2 = np.zeros((10))*

- Similarly, we can initialize an identity matrix using:

*array3 = np.eye((10,10))*

in which array3 is a 10x10 identity matrix.

# Writing *powell* | Arrays with NumPy

38

- Calling and manipulating elements in a NumPy array is similar to Matlab.

Examples are below:

*Array1[0,0] = 100.0*

*Array1[0,1] = 101.0*

*Array1[1,0] = 101.0*

*Array[:,0] = 102.0*

*Array[5,:] = 105.0*

- Now with the pseudocode, you should be prepared to implement the *powell* function on your own.

# Writing *powell* | Code

39

1. Initialize  $\mathbf{x}_0$ ,  $\mathbf{y}$ , and  $n$ -by- $n$  identity matrix  $\mathbf{H}$
2. Outer Loop (while outer counter  $<$  max iterations)
  1. Set  $q = 1$
  2. Inner Loop (for  $q = 1$  to  $n$ )
    1. Line search with Golden Section in direction  $\mathbf{H}(:,q)$
    2.  $\mathbf{x} = \mathbf{x} + \alpha^* \mathbf{H}(:,q)$
  3.  $\mathbf{S}_{\text{new}} = \mathbf{x} - \mathbf{y}$
  4. Line search with Golden Section in direction  $\mathbf{S}_{\text{new}}$
  5. Check convergence, break if  $\text{abs}(\text{norm}(\mathbf{S}_{\text{new}})) <$  tolerance
  6. Append  $\mathbf{S}_{\text{new}}$  to  $\mathbf{H}$  and delete  $\mathbf{H}(:,1)$

# Last Comments

40

- Python is easy to use, lightweight, and extensible. Support is easy to find, and for future reference, this online tutorial covers the many capabilities of Python:  
<http://docs.python.org/tutorial/>
- Slides will be posted on the web after class, along with the source code presented here.  
<http://people.virginia.edu/~jrq2a/niatll/index.html>
- A short survey will be emailed out after class. Your answers will help to guide the evolution of the NIATLL.



# Backup

# Writing *powell*:

42

- Allow time for students to work independently on *powell*. Provide source code and test script, in addition to slides, after class is over.

# Completed *powell* function

43

```
50 #This function solves fcn(x,y,...) for its local minimum using Powell's method
51 def powell(x0,eps):
52     N = len(x0)
53     [x,y] = [np.zeros((N)),np.zeros((N))]
54     for i in range(0,N):
55         [x[i],y[i]] = [x0[i],x0[i]]
56     [out,cnt] = [0,0]
57     H = np.eye(N,N)
58     while out == 0:
59         cnt = cnt + 1
60         if cnt > 100:
61             print('Maximum number of iterations surpassed')
62             break
63         for i in range(0,N):
64             GSout = gold(-10,10,eps,x,H[:,i])
65             x[i] = x[i] + GSout[0]*H[i,i]
66         S_new = x - y
67         GSout = gold(-10,10,eps,x,S_new)
68         x = x + GSout[0]*S_new
69         norm = 0.0
70         for i in range(0,N):
71             norm = norm+(x[i]-y[i])**2.0
72             if i < N-1:
73                 H[:,i] = H[:,i+1]
74             else:
75                 H[:,i] = S_new
76         if norm < eps:
77             out = 1
78         y = x
79
80     return [x,fcn(x)]
```

# Testing *powell*

44

```
1 import golden_numpy_nd as gn
2
3 xlow1 = -10.0
4 xlow2 = -10.0
5 xhigh1 = 10.0
6 xhigh2 = 10.0
7 eps1 = 0.01
8 eps2 = 0.01
9 xvec1 = 1.0
10 xvec2 = 1.0
11 svec1 = 1.0
12 svec2 = 1.0
13
14 out1 = gn.gold(xlow1,xhigh1,eps1,xvec1,svec1,0)
15 out2 = gn.gold(xlow2,xhigh2,eps2,xvec2,svec2,1)
16 out3 = gn.gold(xlow1,xhigh1,eps1,xvec1,svec1,100)
17
18 print('Output for case 1 of gold:', out1,' for function 0')
19 print('\nOutput for case 2 of gold:', out2,' for function 1')
20 print('\nOutput for case 3 of gold:', out3,'for function default')
21
22 x01 = [0,0]
23 eps1p = 0.01
24
25 out1p = gn.powell(x01,eps1p,2)
26
27 print('\nOutput for case 1 of powell:', out1p,'for function 2')
```

Copyright 2012 Jesse Quinlan