SIAM Workshop
Introduction to Python for mathematicians and scientists

Mike Sussman

`sussmanm@math.pitt.edu`

`http://www.math.pitt.edu/%7esussmanm`

October 18, 2014

Topics

Who am I?

- Part-time faculty in Math Dept.
- Experience at Bettis lab
- Administer 2070/2071 Numerical Analysis lab
- Interested in numerical applications associated with fluid flow
- Interested in large-scale scientific computing

Objectives

- Introduce Python programming
- Focus on use in scientific work

References

- Recent Python and NumPy/SciPy books from **oreilly.com**
- Python Reference:
  https://docs.python.org/2/reference/index.html
- The Python Tutorial
  https://docs.python.org/2/tutorial
- 10-minute Python tutorial
  http://www.stavros.io/tutorials/python/
- Tentative NumPy Tutorial
  http://wiki.scipy.org/Tentative_NumPy_Tutorial
- Wonderful scientific Python blog by Greg von Winckel
  http://www.scientificpython.net/

Getting Python

1. Recommend using WinPython on MS-Windows

`http://sourceforge.net/p/winpython/wiki/Installation`

2. Download version for Python 2.7
3. Run the installer
4. Do not "register" it
5. Navigate to **`Downloads\WinPython...`**
6. Run **`Spyder`** (not light)

Topics

What is Python?

- Computer programming language
- Interpreted
- Object-oriented
- Extended using “modules” and “packages”

Python and modules

- Core Python: bare-bones
  `https://docs.python.org/2/reference/index.html`
- “Standard Library”
  `https://docs.python.org/2/library/index.html`
- “Python package index” (50,000 packages)
  `https://pypi.python.org`

Python for scientific use

- `numpy`
- `scipy`
- `matplotlib.pylab`
- `sympy`
- `SAGE`

Topics

Running Python

- Use Spyder IDE
- Run `python` in a Cygwin command window

File structure and line syntax

- No mandatory statement termination character.
- Blocks are determined by indentation
- Statements requiring a following block end with a colon (`:`)
- Comments start with octothorpe (`#`), end at end of line
- Multiline comments are surrounded by triple double quotes (`"""`)
- Continue lines with `\`

Topics

Python basics

`example1.py`

1. Debugger
2. x/3, -x/3
3. float(x)/3
4. conjugate(z), abs(w), w*w
5. y0==y1
6. 2**100 (answer is long)

Basic data types

- Integers: `0, -5, 100`
- Floating-point numbers: `3.14159, 6.02e23`
- Complex numbers: `1.5 + 0.5j`
- Strings: `"A string"`
  - Can use single quotes
- Long (integers of arbitrary length)
- Logical or Boolean: `True`, `False`
- `None`

Basic operations

- `+, -, *, /`
- `**` (raise to power)
- `%` (remainder)
- `and, or, not`
- `>, <, >=, <=, ==, !=` (logical comparison)

Python array-type data types

- List: `[0,"string",`*another list* `]`
- Tuple: immutable list, surrounded by `()`
- Dictionary (dict): `{"key1":"value1",2:3,"pi":3.14}`

Getting help

```
>>> help(complex)
class complex(object)
 |  complex(real[, imag]) -> complex number
 |
 |  Create a complex number from a real part and an optional imaginary part.
 |  This is equivalent to (real + imag*1j) where imag defaults to 0.
 |
 |  Methods defined here:
 |
 |  __abs__(...)
 |      x.__abs__() <==> abs(x)
 |
 |  __add__(...)
 |      x.__add__(y) <==> x+y
 |
 |  __div__(...)
 |      x.__div__(y) <==> x/y
 |
 |  conjugate(...)
 |      complex.conjugate() -> complex
 |
 |      Return the complex conjugate of its argument. (3-4j).conjugate() == 3+4j
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  imag
 |      the imaginary part of a complex number
 |
 |  real
 |      the real part of a complex number
 |
```

Topics

Functions, flow control, and import

`example2.py`

1. Debugger
2. i/10
3. n, term, partialSum out of workspace after return!

Functions

- Functions begin with **`def`**
- The **`def`** line ends with a colon
- Function bodies are indented
- Functions use **`return`** to return values

Flow control

- **`if ... elif ... else`**
- **`for`**
- **`while`**
- Bodies are indented
- **`range(N)`** generates `0, 1, ... , (N-1)`

Importing and naming

- Include external libraries using **`import`**
- **`import numpy`**
  Imports all numpy functions, call as **`numpy.sin(x)`**
- **`import numpy as np`**
  Imports all numpy functions, call as **`np.sin(x)`**
- **`from numpy import *`**
  Imports all numpy functions, call as **`sin(x)`**
- **`from numpy import sin`**
  Imports only **`sin()`**

Pylab in Spyder

Automatically does following imports

```
from pylab import *
from numpy import *
from scipy import *
```

You *must* do your own importing when writing code in files

I strongly suggest using names.

```
import numpy as np
import scipy.linalg as la
import matplotlib.pyplot as plt
```

Topics

Subscripts

- `x = [ 'a', 'b', 'c', 'd' ]`

Subscripts

- `x = [ 'a', 'b', 'c', 'd' ]`
- `x[0]` is `'a'`

Subscripts

- `x = [ 'a', 'b', 'c', 'd' ]`
- `x[0]` is `'a'`
- `x[3]` is `'d'`

Subscripts

- `x = [ 'a', 'b', 'c', 'd' ]`
- `x[0]` is `'a'`
- `x[3]` is `'d'`
- `x[0:2]` is `[ 'a', 'b' ]`

Subscripts

- `x = [ 'a', 'b', 'c', 'd' ]`
- `x[0]` is `'a'`
- `x[3]` is `'d'`
- `x[0:2]` is `[ 'a', 'b' ]`
- `x[-1]` is `'d'`

Equals, Copies, and Deep Copies

```
>>> import copy as cp
```

Equals, Copies, and Deep Copies

```
>>> import copy as cp

>>> x=[1,2]
>>> y=[3,4,x]
>>> z=y
>>> print "x=",x,"   y=",y,"   z=",z
x= [1, 2]   y= [3, 4, [1, 2]]   z= [3, 4, [1, 2]]
```

Equals, Copies, and Deep Copies

```
>>> import copy as cp

>>> x=[1,2]
>>> y=[3,4,x]
>>> z=y
>>> print "x=",x,"   y=",y,"   z=",z
x= [1, 2]   y= [3, 4, [1, 2]]   z= [3, 4, [1, 2]]

>>> c=cp.copy(y)
>>> d=cp.deepcopy(y)
>>> print "y=",y," z=",z," c=",c," d=",d
y= [3, 4, [1, 2]]  z= [3, 4, [1, 2]]  c= [3, 4, [1, 2]]  d= [3, 4, [1, 2]]
```

Equals, Copies, and Deep Copies

```
>>> import copy as cp

>>> x=[1,2]
>>> y=[3,4,x]
>>> z=y
>>> print "x=",x,"   y=",y,"   z=",z
x= [1, 2]   y= [3, 4, [1, 2]]   z= [3, 4, [1, 2]]

>>> c=cp.copy(y)
>>> d=cp.deepcopy(y)
>>> print "y=",y," z=",z," c=",c," d=",d
y= [3, 4, [1, 2]]  z= [3, 4, [1, 2]]  c= [3, 4, [1, 2]]  d= [3, 4, [1, 2]]

>>> y[0]="*"
>>> print "y=",y," z=",z," c=",c," d=",d
```

Equals, Copies, and Deep Copies

```
>>> import copy as cp

>>> x=[1,2]
>>> y=[3,4,x]
>>> z=y
>>> print "x=",x,"   y=",y,"   z=",z
x= [1, 2]   y= [3, 4, [1, 2]]   z= [3, 4, [1, 2]]

>>> c=cp.copy(y)
>>> d=cp.deepcopy(y)
>>> print "y=",y," z=",z," c=",c," d=",d
y= [3, 4, [1, 2]]  z= [3, 4, [1, 2]]  c= [3, 4, [1, 2]]  d= [3, 4, [1, 2]]

>>> y[0]="*"
>>> print "y=",y," z=",z," c=",c," d=",d
y= ["*", 4, [1, 2]]  z= ["*", 4, [1, 2]]  c= [3, 4, [1, 2]]  d= [3, 4, [1, 2]]
```

Equals, Copies, and Deep Copies

```
>>> import copy as cp

>>> x=[1,2]
>>> y=[3,4,x]
>>> z=y
>>> print "x=",x,"   y=",y,"   z=",z
x= [1, 2]   y= [3, 4, [1, 2]]   z= [3, 4, [1, 2]]

>>> c=cp.copy(y)
>>> d=cp.deepcopy(y)
>>> print "y=",y," z=",z," c=",c," d=",d
y= [3, 4, [1, 2]]  z= [3, 4, [1, 2]]  c= [3, 4, [1, 2]]  d= [3, 4, [1, 2]]

>>> y[0]="*"
>>> print "y=",y," z=",z," c=",c," d=",d
y= ["*", 4, [1, 2]]  z= ["*", 4, [1, 2]]  c= [3, 4, [1, 2]]  d= [3, 4, [1, 2]]

>>> z[2][0]=9
>>> print "y=",y," z=",z," c=",c," d=",d
```

Equals, Copies, and Deep Copies

```
>>> import copy as cp

>>> x=[1,2]
>>> y=[3,4,x]
>>> z=y
>>> print "x=",x,"   y=",y,"   z=",z
x= [1, 2]   y= [3, 4, [1, 2]]   z= [3, 4, [1, 2]]

>>> c=cp.copy(y)
>>> d=cp.deepcopy(y)
>>> print "y=",y," z=",z," c=",c," d=",d
y= [3, 4, [1, 2]]  z= [3, 4, [1, 2]]  c= [3, 4, [1, 2]]  d= [3, 4, [1, 2]]

>>> y[0]="*"
>>> print "y=",y," z=",z," c=",c," d=",d
y= ["*", 4, [1, 2]]  z= ["*", 4, [1, 2]]  c= [3, 4, [1, 2]]  d= [3, 4, [1, 2]]

>>> z[2][0]=9
>>> print "y=",y," z=",z," c=",c," d=",d
y= ["*", 4, [9, 2]]  z= ["*", 4, [9, 2]]  c= [3, 4, [9, 2]]  d= [3, 4, [1, 2]]
```

Equals, Copies, and Deep Copies

```
>>> import copy as cp

>>> x=[1,2]
>>> y=[3,4,x]
>>> z=y
>>> print "x=",x,"   y=",y,"   z=",z
x= [1, 2]   y= [3, 4, [1, 2]]   z= [3, 4, [1, 2]]

>>> c=cp.copy(y)
>>> d=cp.deepcopy(y)
>>> print "y=",y," z=",z," c=",c," d=",d
y= [3, 4, [1, 2]]  z= [3, 4, [1, 2]]  c= [3, 4, [1, 2]]  d= [3, 4, [1, 2]]

>>> y[0]="*"
>>> print "y=",y," z=",z," c=",c," d=",d
y= ["*", 4, [1, 2]]  z= ["*", 4, [1, 2]]  c= [3, 4, [1, 2]]  d= [3, 4, [1, 2]]

>>> z[2][0]=9
>>> print "y=",y," z=",z," c=",c," d=",d
y= ["*", 4, [9, 2]]  z= ["*", 4, [9, 2]]  c= [3, 4, [9, 2]]  d= [3, 4, [1, 2]]

>>> c[2][1]='c'
>>> print "y=",y," z=",z," c=",c," d=",d
```

Equals, Copies, and Deep Copies

```
>>> import copy as cp

>>> x=[1,2]
>>> y=[3,4,x]
>>> z=y
>>> print "x=",x,"   y=",y,"   z=",z
x= [1, 2]   y= [3, 4, [1, 2]]   z= [3, 4, [1, 2]]

>>> c=cp.copy(y)
>>> d=cp.deepcopy(y)
>>> print "y=",y," z=",z," c=",c," d=",d
y= [3, 4, [1, 2]]  z= [3, 4, [1, 2]]  c= [3, 4, [1, 2]]  d= [3, 4, [1, 2]]

>>> y[0]="*"
>>> print "y=",y," z=",z," c=",c," d=",d
y= ["*", 4, [1, 2]]  z= ["*", 4, [1, 2]]  c= [3, 4, [1, 2]]  d= [3, 4, [1, 2]]

>>> z[2][0]=9
>>> print "y=",y," z=",z," c=",c," d=",d
y= ["*", 4, [9, 2]]  z= ["*", 4, [9, 2]]  c= [3, 4, [9, 2]]  d= [3, 4, [1, 2]]

>>> c[2][1]='c'
>>> print "y=",y," z=",z," c=",c," d=",d
y= ["*", 4, [9, c]]  z= ["*", 4, [9, c]]  c= [3, 4, [9, c]]  d= [3, 4, [1, 2]]

>>> x
[9, c]
```

Equals, Copies, and Deep Copies

```
>>> import copy as cp

>>> x=[1,2]
>>> y=[3,4,x]
>>> z=y
>>> print "x=",x,"   y=",y,"   z=",z
x= [1, 2]   y= [3, 4, [1, 2]]   z= [3, 4, [1, 2]]

>>> c=cp.copy(y)
>>> d=cp.deepcopy(y)
>>> print "y=",y," z=",z," c=",c," d=",d
y= [3, 4, [1, 2]]  z= [3, 4, [1, 2]]  c= [3, 4, [1, 2]]  d= [3, 4, [1, 2]]

>>> y[0]="*"
>>> print "y=",y," z=",z," c=",c," d=",d
y= ["*", 4, [1, 2]]  z= ["*", 4, [1, 2]]  c= [3, 4, [1, 2]]  d= [3, 4, [1, 2]]

>>> z[2][0]=9
>>> print "y=",y," z=",z," c=",c," d=",d
y= ["*", 4, [9, 2]]  z= ["*", 4, [9, 2]]  c= [3, 4, [9, 2]]  d= [3, 4, [1, 2]]

>>> c[2][1]='c'
>>> print "y=",y," z=",z," c=",c," d=",d
y= ["*", 4, [9, c]]  z= ["*", 4, [9, c]]  c= [3, 4, [9, c]]  d= [3, 4, [1, 2]]

>>> x
[9, c]
```

Equals, Copies, and Deep Copies

```
>>> import copy as cp

>>> x=[1,2]
>>> y=[3,4,x]
>>> z=y
>>> print "x=",x,"   y=",y,"   z=",z
x= [1, 2]   y= [3, 4, [1, 2]]   z= [3, 4, [1, 2]]

>>> c=cp.copy(y)
>>> d=cp.deepcopy(y)
>>> print "y=",y," z=",z," c=",c," d=",d
y= [3, 4, [1, 2]]  z= [3, 4, [1, 2]]  c= [3, 4, [1, 2]]  d= [3, 4, [1, 2]]

>>> y[0]="*"
>>> print "y=",y," z=",z," c=",c," d=",d
y= ["*", 4, [1, 2]]  z= ["*", 4, [1, 2]]  c= [3, 4, [1, 2]]  d= [3, 4, [1, 2]]

>>> z[2][0]=9
>>> print "y=",y," z=",z," c=",c," d=",d
y= ["*", 4, [9, 2]]  z= ["*", 4, [9, 2]]  c= [3, 4, [9, 2]]  d= [3, 4, [1, 2]]

>>> c[2][1]='c'
>>> print "y=",y," z=",z," c=",c," d=",d
y= ["*", 4, [9, c]]  z= ["*", 4, [9, c]]  c= [3, 4, [9, c]]  d= [3, 4, [1, 2]]

>>> x
[9, c]
```

Equals, Copies, and Deep Copies

```
>>> import copy as cp

>>> x=[1,2]
>>> y=[3,4,x]
>>> z=y
>>> print "x=",x,"   y=",y,"   z=",z
x= [1, 2]   y= [3, 4, [1, 2]]   z= [3, 4, [1, 2]]

>>> c=cp.copy(y)
>>> d=cp.deepcopy(y)
>>> print "y=",y," z=",z," c=",c," d=",d
y= [3, 4, [1, 2]]  z= [3, 4, [1, 2]]  c= [3, 4, [1, 2]]  d= [3, 4, [1, 2]]

>>> y[0]="*"
>>> print "y=",y," z=",z," c=",c," d=",d
y= ["*", 4, [1, 2]]  z= ["*", 4, [1, 2]]  c= [3, 4, [1, 2]]  d= [3, 4, [1, 2]]

>>> z[2][0]=9
>>> print "y=",y," z=",z," c=",c," d=",d
y= ["*", 4, [9, 2]]  z= ["*", 4, [9, 2]]  c= [3, 4, [9, 2]]  d= [3, 4, [1, 2]]

>>> c[2][1]='c'
>>> print "y=",y," z=",z," c=",c," d=",d
y= ["*", 4, [9, c]]  z= ["*", 4, [9, c]]  c= [3, 4, [9, c]]  d= [3, 4, [1, 2]]

>>> x
[9, c]
```

Moral: Only **`deepcopy`** does it right!

Topics

A Class is a generalized data type

- **`numpy`** defines a class called **`ndarray`**
- Define variable **`x`** of type **`ndarray`**, a one-dimensional array of length 10:

```
import numpy as np
x=np.ndarray([10])
```

- Varibles of type **`ndarray`** are usually just called “array”.

Classes define members' "attributes"

- Attributes can be data
  - Usually, data attributes are "hidden"
  - Names start with double-underscore
  - Programmers are trusted not to access such data
- Attributes can be functions
  - Functions are provided to access "hidden" data

Examples of attributes

One way to generate a **`numpy`** array is:

```
import numpy as np
x=np.array( [0, 0.1, 0.2, 0.4, 0.9, 3.14] )
```

- (data attribute) **`x.size`** is **`6`**.
- (data attribute) **`x.dtype`** is **`"float64"`** (quotes mean “string”)
- (function attribute) **`x.item(2)`** is **`0.2`** (parentheses mean “function”)
- Copy function is provided by numpy:
  `y = x.copy()` *or*
  `y = np.copy(x)`

Operators can be overridden

- Multiplication and division are pre-defined (overridden)

```
>>> 3*x
array([ 0.   ,  0.3 ,  0.6 ,  1.2 ,  2.7 ,  9.42])
```

- Brackets can be overridden to make things look “normal”

```
>>> x[2] # bracket overridden
0.2
```

Topics

Topics

Gauß integration

- Integrate $Q = \int_0^1 f(\xi) d\xi$
- Approximate it as $Q \approx \sum_{i=1}^{3} w_i f(g_i)$
- w_i and g_i come from reference materials.

Example 3

`example3.py`

- Function of a vector returns a vector
- **`np.not`**
- Extensive testing!
- **`y=0.0*x+1.0`** $\Longleftrightarrow$ **`y=np.zeros_like(x)`** $\Longleftrightarrow$ **`y=np.zeros( shape(x) )`**
- **`append()`** is a List attribute (function)

Topics

A boundary value problem by the finite element method

$$
\begin{aligned}
-u'' + u &= f \quad \text{on } [0, L] \\
u'(0) &= 0 = u'(L)
\end{aligned}
$$

1. Pick a basis of functions $\phi_i(x)$
2. Write $u(x) \approx \sum_i u_i \phi_i(x)$
3. Plug into equation
4. Multiply equation through by $\phi_j(x)$ and integrate
5. Solve system of equations for u_i

FEM formulation

$$-u'' + u = f(x) \qquad u'(0) = u'(L) = 0$$

Multiply through by a function v and integrate

$$\int_0^L u'(x)v'(x)dx + \Big[u'(x)v(x)\Big]_0^L + \int_0^L u(x)v(x)dx = \int_0^L f(x)v(x)dx$$

The bracketed term drops out because of boundary values.
Assume that an approximate solution can be written as

$$u(x) = \sum_{j=1}^{N} u_j \phi_j(x)$$

Choosing $v(x) = \phi_i(x)$ yields

$$\sum_{j=1}^{N} \underbrace{\left(\int_0^L \phi_i'(x)\phi_j'(x)dx + \int_0^L \phi_i(x)\phi_j(x)dx \right)}_{a_{ij}} u_j = \underbrace{\int_0^L f(x)\phi_i(x)dx}_{f_i}.$$

$$\mathbf{AU} = \mathbf{F}.$$

Do integrations elementwise

- Break $[0, L]$ into N uniform subintervals $e_k : k = 0, \dots, N-1$, each of width Δx
- $\int_0^L \phi_i(x) f(x)\, dx = \sum_k \int_{e_k} \phi_i(x) f(x)\, dx$
- $\int_{e_k} \phi_i(x) f(x)\, dx = \int_0^1 \phi_i(\xi) f(\xi) \Delta x\, d\xi$
- Inside e_k, define

$$\begin{aligned} \phi^0(\xi) &= 2.0(\xi - 0.5)(\xi - 1.0), \\ \phi^1(\xi) &= 4.0\xi(1.0 - \xi), \\ \phi^2(\xi) &= 2.0\xi(\xi - 0.5) \end{aligned}$$

Topics

What do the shape functions look like?

- Suppose $\Delta x = L/N$ so that $e_k = [k\Delta x, (k+1)\Delta x]$
- Define $x_i = i\Delta x/2$ for $i = 0, 1, \ldots, 2N$
- $x_{2N} = L$
- $e_k = [x_{2k}, x_{2K+2}]$ for $k = 0, 1, \ldots, (N-1)$
- Map $e_k \to [0, 1]$ is given by $x = (k + \xi)\Delta x$
- Inside e_k,

$$
\begin{aligned}
\xi &= (x - k\Delta x)/\Delta x \\
\phi_k^0(x) &= 2.0(\xi - 0.5)(\xi - 1.0), \\
\phi_k^1(x) &= 4.0\xi(1.0 - \xi), \\
\phi_k^2(x) &= 2.0\xi(\xi - 0.5)
\end{aligned}
$$

- Outside e_k, $\phi_k^i = 0$

Shape functions are a "partition of unity"

- Each ϕ_k^i is 1 at $\xi_i \in e_k$ and 0 elsewhere
- Each ϕ is piecewise quadratic
- The unique piecewise quadratic function satisfying $\phi_i(x_j) = \delta_{ij}$ agrees with one of the ϕ_k^i when $x_j \in e_k$.
- $\sum_i \phi_i(x) = 1$
- ϕ_i form a basis of the space of piecewise quadratic functions with breaks at x_i.
- u piecewise quadratic $\Rightarrow u(x) = \sum_i u_i \phi_i(x)$
- u_i are called "degrees of freedom."

Topics

Example 4

`example4.py`

- ϕ_1^2 and ϕ_2^0 together make ϕ_4
- 2D subscripting: **(**Amat1[m, n])
- **`plt.plot`** and **`plt.hold`** are like Matlab
- **`np.linspace`** is like Matlab

Topics

Example 5

`example5.py`

- Solves $-u'' + u = f$ in weak form $\int u'v' + \int uv = \int fv$ with Neumann boundary conditions on $[0, 5]$
- Two tests: $f_0(x) = 1$ and $f_1(x) = x$
- Two exact solutions: $u_0(x) = x$ and $u_1(x) = \frac{\cosh(5)-1}{\sinh(5)} \cosh(x) - \sinh(x) + x$

Convergence results

Convergence results		
N	error	ratio
5	0.000142449953558	
10	8.60661737944e-06	16.55121254702143
20	5.21196552761e-07	16.51318937903001
40	3.19766785929e-08	16.29927108429344
80	1.97897364593e-09	16.1582134550725
160	1.23080146312e-10	16.0787397905218

Fourth-order convergence is too high, a consequence of “superconvergence.”

Topics

Some differences with C++

- Indentation
- `**`, `and`, `or`
- `long`
- `(=)` and copying
- Interpreted *vs.* compiled
- No private variables
  - Programmer must pretend not to see variables starting with __
- No `const`
- Cannot have two functions with same name
  - Allowed in C++ if signatures different
- Automatic garbage collection
- Variable types are implicit
- Constructor syntax
- Extra parameter `self`

Some versions/variants of Python

- **`CPython`**: reference implementation of Python, written in C
- **`Cython`**: superset of Python. Easy to write integrated C or C++ and Python code
- **`Jython`**: version of Python written in Java, can easily call Java methods

Topics

FEniCS

FEniCS is a package for solving partial differential equations expressed in weak form.

1. You write a script in high-level Python
   - Uses UFL form language
   - Can use numpy, scipy, matplotlib.pyplot, *etc.*
   - Can use Viper for plotting
2. DOLFIN interprets the script
3. UFL is passed to FFC for compilation
4. Instant turns it into C++ callable from Python ("swig")
5. Linear algebra is passed to PETSc or UMFPACK

DOLFIN classes

- Sparse matrices and vectors *via* PETSc
- Solvers *via* PETSc can run in parallel
- Eigenvalues *via* SLEPc
- Newton solver for nonlinear equations
- Connected to ParaView for plotting solutions

FEniCS example

```
from dolfin import *

# Create mesh and define function space
mesh = UnitSquareMesh(6, 4)
V = FunctionSpace(mesh, 'Lagrange', 1)

# Define boundary conditions
u0 = Expression('1 + x[0]*x[0] + 2*x[1]*x[1]')

def u0_boundary(x, on_boundary):
    return on_boundary

bc = DirichletBC(V, u0, u0_boundary)

# Define variational problem
u = TrialFunction(V)
v = TestFunction(V)
f = Constant(-6.0)
a = inner(nabla_grad(u), nabla_grad(v))*dx
L = f*v*dx

# Compute solution
u = Function(V)
solve(a == L, u, bc)

# Plot solution and mesh
plot(u,interactive=True)
plot(mesh,interactive=True)
```

Always start with this

FEniCS example

```
from dolfin import *

# Create mesh and define function space
mesh = UnitSquareMesh(6, 4)
V = FunctionSpace(mesh, 'Lagrange', 1)

# Define boundary conditions
u0 = Expression('1 + x[0]*x[0] + 2*x[1]*x[1]')

def u0_boundary(x, on_boundary):
    return on_boundary

bc = DirichletBC(V, u0, u0_boundary)

# Define variational problem
u = TrialFunction(V)
v = TestFunction(V)
f = Constant(-6.0)
a = inner(nabla_grad(u), nabla_grad(v))*dx
L = f*v*dx

# Compute solution
u = Function(V)
solve(a == L, u, bc)

# Plot solution and mesh
plot(u,interactive=True)
plot(mesh,interactive=True)
```

- Mesh on $[0,1] \times [0,1]$
- Uniform 6 cells in x_0, 4 in x_1

FEniCS example

```
from dolfin import *

# Create mesh and define function space
mesh = UnitSquareMesh(6, 4)
V = FunctionSpace(mesh, 'Lagrange', 1)

# Define boundary conditions
u0 = Expression('1 + x[0]*x[0] + 2*x[1]*x[1]')

def u0_boundary(x, on_boundary):
    return on_boundary

bc = DirichletBC(V, u0, u0_boundary)

# Define variational problem
u = TrialFunction(V)
v = TestFunction(V)
f = Constant(-6.0)
a = inner(nabla_grad(u), nabla_grad(v))*dx
L = f*v*dx

# Compute solution
u = Function(V)
solve(a == L, u, bc)

# Plot solution and mesh
plot(u,interactive=True)
plot(mesh,interactive=True)
```

Linear Lagrange shape functions

FEniCS example

```
from dolfin import *

# Create mesh and define function space
mesh = UnitSquareMesh(6, 4)
V = FunctionSpace(mesh, 'Lagrange', 1)

# Define boundary conditions
u0 = Expression('1 + x[0]*x[0] + 2*x[1]*x[1]')

def u0_boundary(x, on_boundary):
    return on_boundary

bc = DirichletBC(V, u0, u0_boundary)

# Define variational problem
u = TrialFunction(V)
v = TestFunction(V)
f = Constant(-6.0)
a = inner(nabla_grad(u), nabla_grad(v))*dx
L = f*v*dx

# Compute solution
u = Function(V)
solve(a == L, u, bc)

# Plot solution and mesh
plot(u,interactive=True)
plot(mesh,interactive=True)
```

- "Expression" causes a compilation
- **x** is a "global variable"

FEniCS example

```
from dolfin import *

# Create mesh and define function space
mesh = UnitSquareMesh(6, 4)
V = FunctionSpace(mesh, 'Lagrange', 1)

# Define boundary conditions
u0 = Expression('1 + x[0]*x[0] + 2*x[1]*x[1]')

def u0_boundary(x, on_boundary):
    return on_boundary

bc = DirichletBC(V, u0, u0_boundary)

# Define variational problem
u = TrialFunction(V)
v = TestFunction(V)
f = Constant(-6.0)
a = inner(nabla_grad(u), nabla_grad(v))*dx
L = f*v*dx

# Compute solution
u = Function(V)
solve(a == L, u, bc)

# Plot solution and mesh
plot(u,interactive=True)
plot(mesh,interactive=True)
```

- `on_boundary` is a "global" variable

FEniCS example

```
from dolfin import *

# Create mesh and define function space
mesh = UnitSquareMesh(6, 4)
V = FunctionSpace(mesh, 'Lagrange', 1)

# Define boundary conditions
u0 = Expression('1 + x[0]*x[0] + 2*x[1]*x[1]')

def u0_boundary(x, on_boundary):
    return on_boundary

bc = DirichletBC(V, u0, u0_boundary)

# Define variational problem
u = TrialFunction(V)
v = TestFunction(V)
f = Constant(-6.0)
a = inner(nabla_grad(u), nabla_grad(v))*dx
L = f*v*dx

# Compute solution
u = Function(V)
solve(a == L, u, bc)

# Plot solution and mesh
plot(u,interactive=True)
plot(mesh,interactive=True)
```

- Set Dirichlet b.c.
- Can be more than one boundary
- `u0_boundary` is used

FEniCS example

```
from dolfin import *

# Create mesh and define function space
mesh = UnitSquareMesh(6, 4)
V = FunctionSpace(mesh, 'Lagrange', 1)

# Define boundary conditions
u0 = Expression('1 + x[0]*x[0] + 2*x[1]*x[1]')

def u0_boundary(x, on_boundary):
    return on_boundary

bc = DirichletBC(V, u0, u0_boundary)

# Define variational problem
u = TrialFunction(V)
v = TestFunction(V)
f = Constant(-6.0)
a = inner(nabla_grad(u), nabla_grad(v))*dx
L = f*v*dx

# Compute solution
u = Function(V)
solve(a == L, u, bc)

# Plot solution and mesh
plot(u,interactive=True)
plot(mesh,interactive=True)
```

- This is UFL
- Specify weak form
- $-\Delta u = f \iff \int \nabla u \nabla v \, dx = \int f \, v \, dx$

FEniCS example

```
from dolfin import *

# Create mesh and define function space
mesh = UnitSquareMesh(6, 4)
V = FunctionSpace(mesh, 'Lagrange', 1)

# Define boundary conditions
u0 = Expression('1 + x[0]*x[0] + 2*x[1]*x[1]')

def u0_boundary(x, on_boundary):
    return on_boundary

bc = DirichletBC(V, u0, u0_boundary)

# Define variational problem
u = TrialFunction(V)
v = TestFunction(V)
f = Constant(-6.0)
a = inner(nabla_grad(u), nabla_grad(v))*dx
L = f*v*dx

# Compute solution
u = Function(V)
solve(a == L, u, bc)

# Plot solution and mesh
plot(u,interactive=True)
plot(mesh,interactive=True)
```

Define trial and test function spaces

FEniCS example

```
from dolfin import *

# Create mesh and define function space
mesh = UnitSquareMesh(6, 4)
V = FunctionSpace(mesh, 'Lagrange', 1)

# Define boundary conditions
u0 = Expression('1 + x[0]*x[0] + 2*x[1]*x[1]')

def u0_boundary(x, on_boundary):
    return on_boundary

bc = DirichletBC(V, u0, u0_boundary)

# Define variational problem
u = TrialFunction(V)
v = TestFunction(V)
f = Constant(-6.0)
a = inner(nabla_grad(u), nabla_grad(v))*dx
L = f*v*dx

# Compute solution
u = Function(V)
solve(a == L, u, bc)

# Plot solution and mesh
plot(u,interactive=True)
plot(mesh,interactive=True)
```

Define $L(v) = \int f(x)v(x)\,dx$ with $f = -6$

FEniCS example

```
from dolfin import *

# Create mesh and define function space
mesh = UnitSquareMesh(6, 4)
V = FunctionSpace(mesh, 'Lagrange', 1)

# Define boundary conditions
u0 = Expression('1 + x[0]*x[0] + 2*x[1]*x[1]')

def u0_boundary(x, on_boundary):
    return on_boundary

bc = DirichletBC(V, u0, u0_boundary)

# Define variational problem
u = TrialFunction(V)
v = TestFunction(V)
f = Constant(-6.0)
a = inner(nabla_grad(u), nabla_grad(v))*dx
L = f*v*dx

# Compute solution
u = Function(V)
solve(a == L, u, bc)

# Plot solution and mesh
plot(u,interactive=True)
plot(mesh,interactive=True)
```

Define $a(u, v) = \int \nabla u \cdot \nabla v \, dx$

FEniCS example

```
from dolfin import *

# Create mesh and define function space
mesh = UnitSquareMesh(6, 4)
V = FunctionSpace(mesh, 'Lagrange', 1)

# Define boundary conditions
u0 = Expression('1 + x[0]*x[0] + 2*x[1]*x[1]')

def u0_boundary(x, on_boundary):
    return on_boundary

bc = DirichletBC(V, u0, u0_boundary)

# Define variational problem
u = TrialFunction(V)
v = TestFunction(V)
f = Constant(-6.0)
a = inner(nabla_grad(u), nabla_grad(v))*dx
L = f*v*dx

# Compute solution
u = Function(V)
solve(a == L, u, bc)

# Plot solution and mesh
plot(u,interactive=True)
plot(mesh,interactive=True)
```

`u` is *redefined* as a **`Function`** instead of **`TrialFunction`**

FEniCS example

```
from dolfin import *

# Create mesh and define function space
mesh = UnitSquareMesh(6, 4)
V = FunctionSpace(mesh, 'Lagrange', 1)

# Define boundary conditions
u0 = Expression('1 + x[0]*x[0] + 2*x[1]*x[1]')

def u0_boundary(x, on_boundary):
    return on_boundary

bc = DirichletBC(V, u0, u0_boundary)

# Define variational problem
u = TrialFunction(V)
v = TestFunction(V)
f = Constant(-6.0)
a = inner(nabla_grad(u), nabla_grad(v))*dx
L = f*v*dx

# Compute solution
u = Function(V)
solve(a == L, u, bc)

# Plot solution and mesh
plot(u,interactive=True)
plot(mesh,interactive=True)
```

Solve the system $L(v) = a(u, v)\ \forall v$ subject to boundary conditions.

FEniCS example

```
from dolfin import *

# Create mesh and define function space
mesh = UnitSquareMesh(6, 4)
V = FunctionSpace(mesh, 'Lagrange', 1)

# Define boundary conditions
u0 = Expression('1 + x[0]*x[0] + 2*x[1]*x[1]')

def u0_boundary(x, on_boundary):
    return on_boundary

bc = DirichletBC(V, u0, u0_boundary)

# Define variational problem
u = TrialFunction(V)
v = TestFunction(V)
f = Constant(-6.0)
a = inner(nabla_grad(u), nabla_grad(v))*dx
L = f*v*dx

# Compute solution
u = Function(V)
solve(a == L, u, bc)

# Plot solution and mesh
plot(u,interactive=True)
plot(mesh,interactive=True)
```

- Plot u and mesh in two frames.
- `interactive=True` causes the plot to remain displayed until destroyed by mouse.
- Can also put `interactive()` at the end.