Com S // Cpr E // Math 5250

Numerical Analysis of High-Performance Computing

Instructor: Songting Luo

Lecture 5: NumPy arrays

Outline

1. Summary from last time

2. NumPy arrays

Summary from last time

Good coding practices demo

```
def sqrt(x,printshow=0,initial_guess=3.0,kmax=100,tol=1.0e-14):
    """
    Newton's method for solving f(s) = s^2 - x = 0
    Input:
    x: float number
    printshow: switch on/off printing intermediate steps. 0 off(defaul)
    initial_guess: float number, initial guess for the iteration, defau
    kmax: integer, maximum number for iterations, default = 100.
    tol: float number, tolerance for two successive iterations, default
    Output: square root of x
    """

    # convert input to float
    x = 1.0*x

    # check if input makes sense
    if x==0.0:
        return 0.0
    elif x<0.0:
        print( "** ERROR: input x must be non-negative **" )
        return -1.0
```

Good coding practices demo

```
# main loop
s = initial_guess
for k in range(kmax):
    if printshow==1:
        print( "Before iteration %d, s = %20.15f" % (k,s) )

    sold = s
    s = 0.5*(s + x/s)

    if (abs((s-sold)/x)<tol):
        break

if printshow==1:
    print( "After %d iterations, s = %20.15f" % (k+1,s) )

return s
```

Good coding practices demo

```
def test_sqrt():
    from numpy import sqrt as numpy_sqrt

    xvalues = [0.0, 2.0, 100.0, 10000.0, 1.0e-4]

    for x in xvalues:
        print( " Testing with x = %20.15e" % x )
        s_mine = sqrt(x)
        s_numpy = numpy_sqrt(x)

        print("s = %20.15e, numpy_sqrt =  %20.15e" % (s_mine,s_numpy))

        assert abs(s_mine - s_numpy) < 1.0e-14, \
          "Disagree with numpy for x = %20.15e" % x
```

Good coding practices demo

```
def test_sqrt():
    from numpy import sqrt as numpy_sqrt

    xvalues = [0.0, 2.0, 100.0, 10000.0, 1.0e-4]

    for x in xvalues:
        print( " Testing with x = %20.15e" % x )
        s_mine = sqrt(x)
        s_numpy = numpy_sqrt(x)

        print("s = %20.15e, numpy_sqrt =  %20.15e" % (s_mine,s_numpy))

        assert abs(s_mine - s_numpy) < 1.0e-14, \
          "Disagree with numpy for x = %20.15e" % x
```

Demo:

```
$ cd $ISUHPC/lectures/lecture4
$ python3
>>> import myfuncs as my
>>> my.test_sqrt()
```

Python modules

When you start Python it has a few basic built-in types and functions.

To do something fancier, you will probably import modules.

Example: square root, π, sin, cos, and exp from NumPy

```
from numpy import sqrt
>>> sqrt(23.0)
4.795831523312719

>>> from numpy import pi, sin, cos
>>> sin(pi/3.0)
0.8660254037844386
>>> cos(pi/3.0)
0.5000000000000011

>>> from numpy import exp
>>> exp(7.11)
1224.1475460917379
```

Python modules

When you type `import modname`, Python looks on its search path for the file modname.py.

You can add more directories using the Unix environment variable `$PYTHONPATH`.

Or, in Python, you can add to the Python path using the sys module:

```
>>> import sys
>>> sys.path  # returns list of directories in path
['', '/Users/luos', ... ]

>>> sys.path.append('newdirectory')
```

Note: The empty string '' in the search path means it looks first in the current directory.

There are many different ways to import . . .

```
>>> # only import a single function
>>> from numpy import sqrt
>>> sqrt(23.0)
4.795831523312719

>>> # import all functions from numpy
>>> from numpy import *
>>> sqrt(23.0)
4.795831523312719

>>> # import all functions from numpy with namespace numpy
>>> import numpy
>>> numpy.sqrt(23.0)
4.795831523312719

>>> # import all functions from numpy with namespace np
>>> import numpy as np
>>> np.sqrt(23.0)
4.795831523312719
```

NumPy arrays

Lists are not good for linear algebra

Lists in Python are quite general, can have arbitrary objects as elements.

Addition and scalar multiplication are defined for lists, but not what we want for numerical computation:

Multiplication appends to list via duplication:

```
>>> x=[-5.0,1.07]
>>> 3*x
[-5.0, 1.07, -5.0, 1.07, -5.0, 1.07]
```

Addition concatenates

```
>>> y=[4.14, -2.101]
>>> x+y
[-5.0, 1.07, 4.14, -2.101]
```

NumPy module

Instead, use NumPy arrays:

```
>>> import numpy as np

>>> x = np.array([-5.0,1.07])
>>> 3*x
array([-15.  ,   3.21])
```

Other operations also apply component-wise:

```
>>> np.sqrt(np.abs(x)) * np.cos(x) * (x**3)
array([-79.28599117,   0.6084107 ])
```

Note: * is component-wise multiply

Note: ** is component-wise raise to a power

NumPy arrays

Unlike lists, all elements of an `np.array` have the same type

```
>>> # all integers
>>> x = np.array([-3,0,4,11])
>>> print( x )
[-3  0  4 11]

>>> # if we have one float, we get all floats
>>> y = np.array([-3.0,0,4,11])
>>> print( y )
[ -3.   0.   4.  11.]
```

Can explicitly state desired data type:

```
>>> x = np.array([-3,0,4,11], dtype=complex)
>>> print( x )
[ -3.+0.j   0.+0.j   4.+0.j  11.+0.j]
```

Can do complex vector arithmetic:

```
>>> (x + 1.j) * 2.j
array([-2. -6.j, -2. +0.j, -2. +8.j, -2.+22.j])
```

NumPy arrays for vectors and matrices

```
>>> A = np.array([[1.1,2.2],[-2.3,7.5],[2.24,-3.06]])
>>> print( A )
[[ 1.1   2.2 ]
 [-2.3   7.5 ]
 [ 2.24 -3.06]]
>>> A.shape
(3, 2)

>>> At = A.T
>>> print( At )
[[ 1.1  -2.3   2.24]
 [ 2.2   7.5  -3.06]]
>>> At.shape
(2, 3)

>>> x = np.array([1.2,-4.1])
>>> xt = x.T
>>> x.shape; xt.shape;
(2,)
(2,)
```

NumPy arrays for vectors and matrices

```
>>> print( A )
[[ 1.1   2.2 ]
 [-2.3   7.5 ]
 [ 2.24 -3.06]]

>>> print( x )
[ 1.2 -4.1]

>>> np.dot(x,x)   # simple vector dot product
18.25

>>> np.dot(A,x)   # matrix-vector product
array([ -7.7  , -33.51 ,  15.234])

>>> np.dot(At,A)  # matrix-matrix product
array([[ 11.5176, -21.6844],
       [-21.6844,  70.4536]])
```

Number of indeces of an array

The attribute ndim tells you how many indeces the array has

```
>>> A = np.ones((4,4))
>>> print(A)
[[ 1.  1.  1.  1.]
 [ 1.  1.  1.  1.]
 [ 1.  1.  1.  1.]
 [ 1.  1.  1.  1.]]
>>> A.ndim
2
```

Warning: ndim has nothing to do with the dimension of the column space, row space, or null space. Nor does it have anything to do with the number of rows or columns. Use shape to get the number of rows and columns.

```
>>> A.shape
(4, 4)
>>> num_rows = A.shape[0]
>>> num_cols = A.shape[1]
>>> print( 'num_rows = ',num_rows, ' num_cols = ',num_cols )
num_rows =  4  ,  num_cols =  4
```

Number of indeces of an array

Scalars have ndim 0:

```
>>> z = np.array(8.1); z.ndim
0
```

Vectors have ndim 1:

```
>>> q = np.array([2.5, 3.6, 4.7]); q.ndim
1
```

Arrays of any dimension are supported (e.g., ndim = 3)

```
>>> T = np.ones((2,3,3))
>>> print( T )
[[[ 1.  1.  1.]
  [ 1.  1.  1.]
  [ 1.  1.  1.]]

 [[ 1.  1.  1.]
  [ 1.  1.  1.]
  [ 1.  1.  1.]]]
>>> T.ndim
3
```

NumPy arrays for vectors and matrices

```
>>> A = np.array([[1.1,2.2],[-2.3,7.5],[2.24,-3.06]])
>>> A.shape
(3, 2)
>>> print( A )
[[ 1.1   2.2 ]
 [-2.3   7.5 ]
 [ 2.24 -3.06]]
>>> x = np.array([1.2,-4.1])
>>> x.shape
>>> print( x )
[ 1.2 -4.1]

>>> np.dot(x,x)   # simple vector dot product
18.25

>>> np.dot(A,x)   # matrix-vector product
array([ -7.7  , -33.51 ,  15.234])

>>> np.dot(A.T,A)  # matrix-matrix product
array([[ 11.5176, -21.6844],
       [-21.6844,  70.4536]])
```

Linear algebra with NumPy

Set up linear system:

```
>>> import numpy as np

>>> A = np.array([[-4.3, 7.1],[-0.2, 5.6]])
>>> print( A )
[[-4.3  7.1]
 [-0.2  5.6]]

>>> b = np.dot(A, np.array([2.0, -5.0]))
>>> print( b )
[-44.1 -28.4]
```

Now solve $A\vec{x} = \vec{b}$ (via Gaussian Elimination):

```
>>> from numpy.linalg import solve

>>> x = solve(A,b)
>>> print( x )
[ 2. -5.]
```

Eigenvalues

```
>>> from numpy.linalg import eig

>>> # unpacks tuple of numpy arrays (evals,evecs)
>>> evals, evecs  = eig(A)

>>> print( evals )
[-4.15442504  5.45442504]

>>> print( evecs )
[[-0.99978987 -0.58849002]
 [-0.02049921 -0.80850448]]
```

Eigenvalues

```
>>> from numpy.linalg import eig

>>> # unpacks tuple of numpy arrays (evals,evecs)
>>> evals, evecs  = eig(A)

>>> print( evals )
[-4.15442504  5.45442504]

>>> print( evecs )
[[-0.99978987 -0.58849002]
 [-0.02049921 -0.80850448]]
```

Spectral decomposition: $A = R\Lambda R^{-1}$

```
>>> from numpy.linalg import inv
>>> evecs_inv = inv(evecs)
>>> ediag = np.diag(evals)
>>> B = np.dot(np.dot(evecs,ediag),evecs_inv)
>>> print( B )
[[-4.3  7.1]
 [-0.2  5.6]]
```

Quadrature (numerical integration)

$$\int_0^2 x^2\,dx = \left[\frac{x^3}{3}\right]_0^2 = \frac{8}{3}$$

```
>>> import scipy.integrate as scint
>>> def f(x):
...     return x**2
...
>>> scint.quad(f, 0.0, 2.0)
(2.666666666666667, 2.960594732333751e-14)
```

Returns (value, error estimate).

Other keyword arguments to set error tolerance, for example.

Lambda functions

In the last example, *f* is so simple we might want to just include its definition directly in the call to quad.

We can do this with a lambda function:

```
>>> f = lambda x: x**2    # Python's version of inline functions
>>> f(3)
9
>>> scint.quad(f, 0.0, 2.0)
(2.666666666666667, 2.960594732333751e-14)
```

This defines the same *f* as before. But instead we could do:

```
>>> scint.quad(lambda x: x**2, 0.0, 2.0)
(2.666666666666667, 2.960594732333751e-14)
```

More quadrature (numerical integration)

$$\int_{-10}^{10} \int_{-10}^{10} e^{-(xy)^2} \, dx \, dy$$

```
>>> import scipy.integrate as scint
>>> import numpy as np
>>> def g(x,y):
...     return np.exp(-(x*y)**2)
...
>>> a=-10; b=10;
>>> def c(x):
...     return -10.0
...
>>> def d(x):
...     return 10.0
...
>>> scint.dblquad(g,a,b,c,d)
(19.805134167372994, 4.97009930617193e-12)
```

numpy.copy

Remember: Python variables are pointers to objects

```
>>> import numpy as np
>>> A = np.array([[1,2,3],[4,5,6]],dtype=float)
>>> U = A            # U and A point to the same object
>>> V = np.copy(A)   # V and A point to different objects
```

Lab assignment

- Develop a function **GaussElimination(A,b)** for solving the linear system

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

  by Gaussian Elimination, with A and **b** the input and **x** the output.

- Find the cubic polynomial $p(x) = ax^3 + bx^2 + cx + d$ that interpolates $f(x) = \cos(x)$ at the following four points $(-0.1, \cos(-0.1)),\ (-0.02, \cos(-0.02)), (0.02, \cos(0.02)),\ (0.1, \cos(0.1))$ by
  - Set up a 4×4 linear system for computing (a, b, c, d)
  - Solve the linear system using the above function **GaussElimination(A,b)**
  - Measure the maximum error between f and p at the four points.
- Develop a script **demo_polynomial_interpolation.py** for the above tasks.
- Update Git repository.
- Submit both codes and results (screenshot)