Scientific Computation

Spring, 2019

Lecture 20

Notes

- No labs this week
- Extra office hours: Wednesday 10-11, Thursday 1-2, both in MLC
- Please ensure you have read all clarifications for HW3

- HW4 for M4/M5 students has been posted

- Feedback for HW2 should be posted Wednesday evening on Blackboard
  - All marks are provisional and subject to rescaling by an exam committee in June
  - After the HW2 marking is done, I will start going through outstanding HW1 marking queries

- HW2 marking queries will be considered in April
- HW3 marking will be completed some time in May

Today

- Brief discussions of:
  - Classes and object-oriented programming
  - Compiling Python code with Numba
  - Parallel loops with joblib
  - Libraries

- Main takeaways from class

How should we *structure* code?

- Key idea in programming (and general problem solving): Break complex problems into smaller and simpler parts

- Can/should the code structure reflect this?

Modular programming

- In Python, it is natural to distribute portions of code to functions. Collections of functions can be collected in modules (as in coursework).

- This is ok for small problems, what about large collaborative projects
  - Can be difficult to efficiently develop large collection of functions and modules

Classes

Objects and classes: a different approach

Basic idea: package functions (*methods)* and variables (*attributes)* together into *objects*

Methods and data can easily be passed (*inherited)* from one class to another

Arrays as objects

A numpy array can be though of as an object with associated attributes and methods:

```
In [16]: x = np.array([1,2,5,4])

In [17]: x.size
Out[17]: 4

In [18]: x.mean()
Out[18]: 3.0
```

Here, size **is an attribute, and** mean() **is a method**

Classes: trivial example

```
In [5]: class circle:
   ...:     """Simple class for working with circles
   ...:     """
   ...:     def __init__(self,r=1.0):
   ...:         """initialize object"""
   ...:         self.r = r
   ...:     def area(self):
   ...:         """compute area of circle"""
   ...:         from math import pi
   ...:         return pi*self.r*self.r
   ...:     def circumference(self):
   ...:         """compute circumference of circle"""
   ...:         from math import pi
   ...:         return 2*pi*self.r
   ...:
```

Three methods and one attribute (radius, r**) packaged together in** circle **class:**

1) __init__: **Special "constructor" function used to initialize (*instantiate*) an object**

2) area **and** circumference **are (hopefully) self-explanatory**

Note the self **variable – better understood via an example…**

OOP – very brief intro

- Object-oriented programming is (almost) universally used in large software projects

- Particularly important for collaborative development of software

- More important outside of scientific computing
  - Within scientific computing, a mathematical model/numerical method often provides a natural structure for code

OOP – very brief intro

- Object-oriented programming is (almost) universally used in large software projects
- Particularly important for collaborative development of software
- More important outside of scientific computing
  - Within scientific computing, a mathematical model/numerical method often provides a natural structure for code
- Basic idea: organize code into class hierarchies
  - Classes form parent-child relationships
  - Child inherits all methods from parents, grandparents, etc…
  - Allows for organized development of code: once a class is "mature" further growth occurs via development of sub-classes
  - Useful methods/functions can be easily accessed throughout a large project – make sure they are at (or near) the top of the class "family tree"

OOP: simple example

```
In [5]: class circle:
   ...:     """Simple class for working with circles
   ...:     """
   ...:     def __init__(self,r=1.0):
   ...:         """initialize object"""
   ...:         self.r = r
   ...:     def area(self):
   ...:         """compute area of circle"""
   ...:         from math import pi
   ...:         return pi*self.r*self.r
   ...:     def circumference(self):
   ...:         """compute circumference of circle"""
   ...:         from math import pi
   ...:         return 2*pi*self.r
   ...:
```

Previously: discussed circle class

Consider cylinder **class which inherits methods from** circle

OOP: simple example

```
In [128]: class cylinder(circle):
     ...:     def __init__(self,r=1.0,h=1.0):
     ...:         self.r=r
     ...:         self.h=h
     ...:     def volume(self):
     ...:         return self.area()*self.h
     ...:     def surface_area(self):
     ...:         return self.circumference()*self.h + 2*self.area()
```

Previously: discussed circle class

Consider cylinder **class which inherits methods from** circle

- **Note the inclusion of** circle **in the header**
- **The** area **and** circumference **methods are available within** cylinder
- **And used within the new** volume **and** circumference **methods**

```
In [129]: Cyl = cylinder(2,3)

In [130]: Cyl.surface_area()
Out[130]: 62.83185307179586
```

OOP: simple example

```
In [128]: class cylinder(circle):
     ...:     def __init__(self,r=1.0,h=1.0):
     ...:         self.r=r
     ...:         self.h=h
     ...:     def volume(self):
     ...:         return self.area()*self.h
     ...:     def surface_area(self):
     ...:         return self.circumference()*self.h + 2*self.area()
```

Previously: discussed circle class

Consider cylinder **class which inherits methods from** circle

- **Note the inclusion of** circle **in the header**
- **The** area **and** circumference **methods are available within** cylinder
- **And used within the new** volume **and** circumference **methods**
- **What would happen if we had** area **instead of** surface_area **in** cylinder?
  - **Then the method in the child would take precedence for cylinder objects**
  - **This is *overloading* – we have already seen *operator overloading***
- **A cylinder object is also a circle object, can use** isinstance **function to check this**

Compiling Python code

- Python is an *interpreted* language
  - Interpreter goes through code line-by-line

- Compiled languages use a compiler to optimize blocks of code
  - Code development is slower, but execution time can be much shorter (especially if code contains loops)
  - Python built-in functions usually utilize compiled routines

- Numba allows Python code to be compiled
  - See: http://numba.pydata.org/numba-doc/0.12.2/tutorial_firststeps.html

Compiling Python code

- **A simple example – insertion sort:**

```
def isort(L):
    """Simple insertion sort code
    """
    for j in range(1,len(L)):
        #compare L[j] with L[i]
        i = j-1
        key = L[j]

        while i>=0:

            if key<L[i]: #shift from i to i-1
                i=i-1
                if i<0:
                    L[i+2:j+1] = L[i+1:j]
                    L[i+1] = key

            else: #insert key at i+1
                L[i+2:j+1] = L[i+1:j]
                L[i+1] = key
                i = -1
        #print("j,L",j,L)

    return L
```

Compiling Python code

- **A simple example – insertion sort:**

```
In [2]: import numba

In [3]: from isort_numba import isort

In [4]: isort_jit = numba.jit("void(i4[:])")(isort)
```

- **The first argument to .jit tells the compiler to expect an array of 4-byte integers as input into the function**

Compiling Python code

- **A simple example – insertion sort:**

```
In [2]: import numba

In [3]: from isort_numba import isort

In [4]: isort_jit = numba.jit("void(i4[:])")(isort)
```

- **The first argument to .jit tells the compiler to expect an array of 4-byte integers as input into the function**

```
In [9]: L = np.random.randint(0,10000,5000)

In [14]: timeit out1=isort(L)
6.94 ms ± 165 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

In [15]: timeit out2=isort_jit(L)
210 µs ± 11.7 µs per loop (mean ± std. dev. of 7 runs, 10000 loops
each)

In [16]: out1[:5]
Out[16]: array([1, 5, 5, 8, 9])

In [17]: out2[:5]
Out[17]: array([1, 5, 5, 8, 9])
```

- **Compiled version is about 30 times faster!**

Parallel computing with Python

- **Several tools exist for parallelizing Python code**
- **The Joblib module allows loops to be parallelized**
  - **Use** `Parallel(n_jobs=num_cores)` **to distribute loops across num_cores processors**

One processor:

```
>>> from math import sqrt
>>> [sqrt(i ** 2) for i in range(10)]
[0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0]
```

Two processors:

```
>>> from math import sqrt
>>> from joblib import Parallel, delayed
>>> Parallel(n_jobs=2)(delayed(sqrt)(i ** 2) for i in range(10))
[0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0]
```

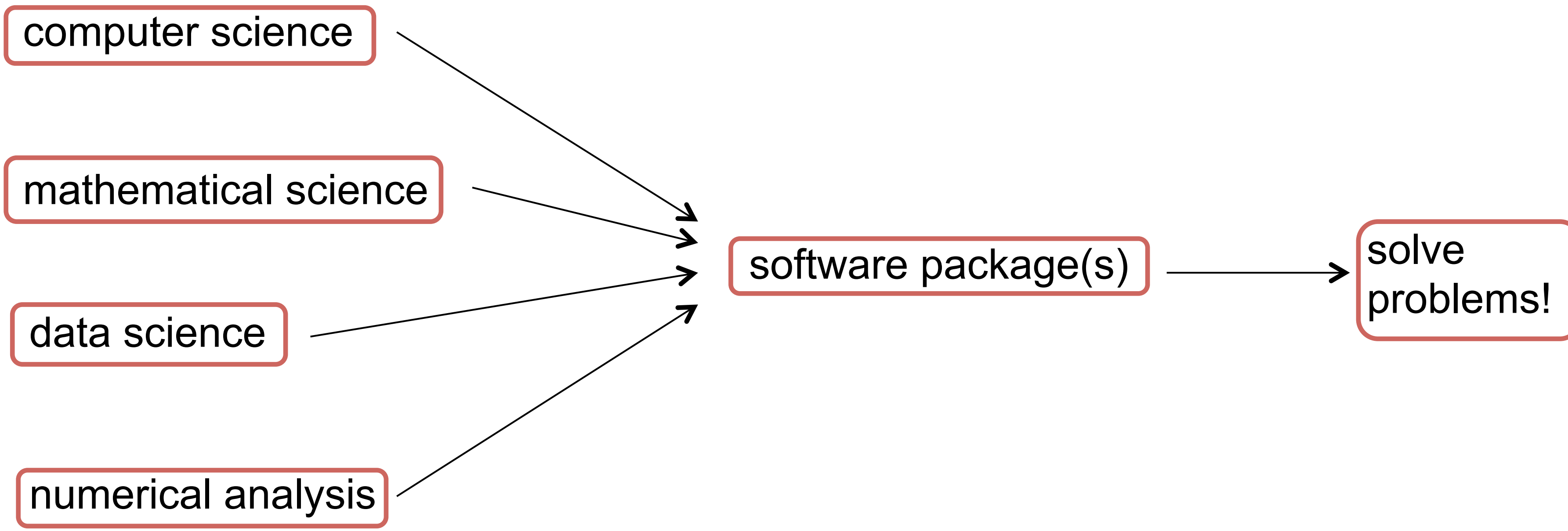
- **The** MPI4PY **module allows distributed-memory computing with Python+MPI**

Scientific libraries

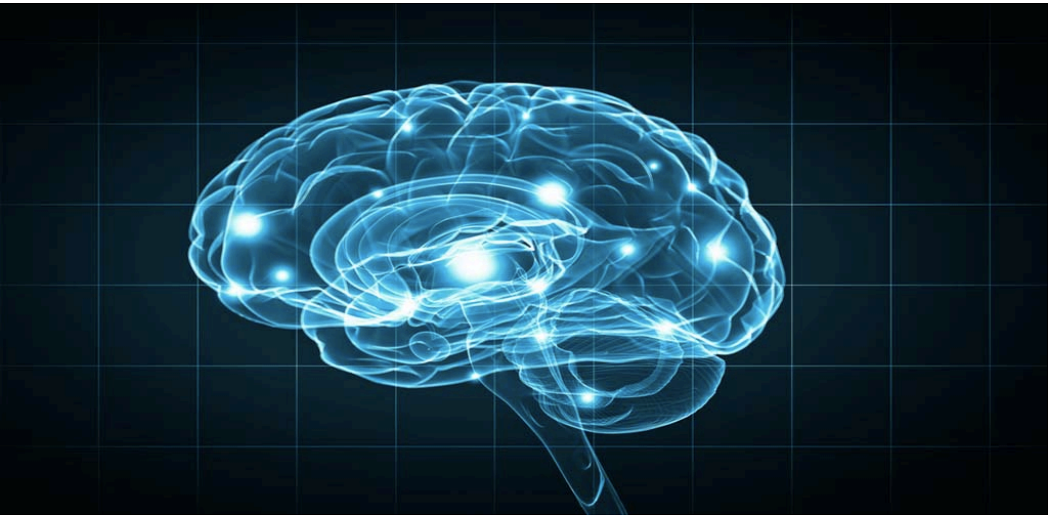
- It is important to have good programming skills
- It is (arguably) more important to know how to use libraries and analyzed results produced by them
- Avoid writing your own code whenever possible!
- Many powerful libraries are readily available in Python
  - fft, networkx, scipy.optimize, numpy.linalg, and many more
  - Parallel libraries (with Python interfaces) also exist, e.g. Petsc
    - But these are not easy to use!

Scientific computation

- **What is "*Scientific computation*"?**
  - **No single, standard definition (that I'm aware of!)**



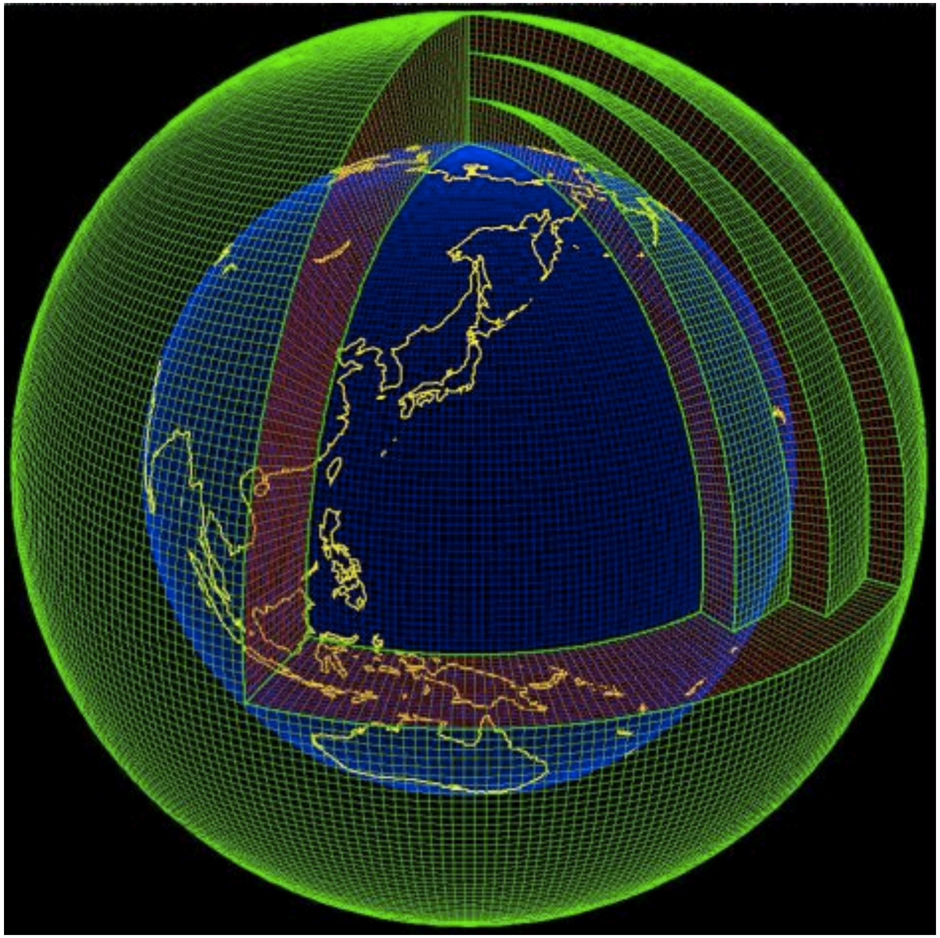
Scientific computation



D.S. Bassett, *How You Think: Structural Network Mechanisms of Human Brain Function*

- **Example 1: Brain dynamics**
- **Graph partitioning**
- **Biochemistry**
- **Nonlinear “differential” equations**

Scientific computation



https://www.jma.go.jp/jma/jma-eng/jma-center/nwp

- **Example 2: Numerical weather prediction**
- **Partial differential equations**
- **Data assimilation**
- **Large sparse linear systems**

Scientific computation

- Example 3: Self-driving car
- Rapid image processing (images are matrices)
- Numerical linear algebra
- Machine learning
- Mathematical optimization

Scientific computation

- The aim of this class is to provide a foundation for further study of these kinds of topics

- Further work in scientific computation can go in many directions:
    - Algorithms
    - Numerical analysis and methods
    - Programming languages
    - Parallel computing

- There is no shortage of important problems waiting to be solved!