Com S // Cpr E // Math 5250

Numerical Analysis of High-Performance Computing

Instructor: Songting Luo

Lecture 3: Floating Point Numbers and More Python

Outline

1. Announcements
2. Floating Point Numbers
3. More Python

Announcements

Installing relevant software on a personal computer

- All software used in this class is open-source (with the exception of software on the HPC-Class cluster)
- Therefore, it is possible for you to setup your own personal machine
- On a Linux machine, installation of gcc, git, python, IPython, matplotlib, pylab is either automatic or easy
- On a Windows machine: Install VirtualBox:
  `https://www.virtualbox.org/`

  Or Windows Subsystem for Linux (WSL):
  `https://docs.microsoft.com/en-us/windows/wsl/`

  This will let you run Linux inside of Windows.
- On a Mac OS X machine:
  - Install Devloper's Tools (Xcode, etc. . .) from `http://www.apple.com`
  - Install gcc and git via homebrew (`http://brew.sh/`) or Fink (`http://www.finkproject.org/`)
  - Install python, IPython, matplotlib, pylab

The power of using Git

- No matter what computer you are on (Carver 449 Lab, your own laptop, another computer lab, etc. . .), you can always work on your Math 5250 Labs by cloning your repository to your current machine:

```
git clone {repository on GitHub}
```

- git push from local machine to GitHub

```
$ git add -A .
$ git commit -m "whatever message to keep track of your files"
$ git push
```

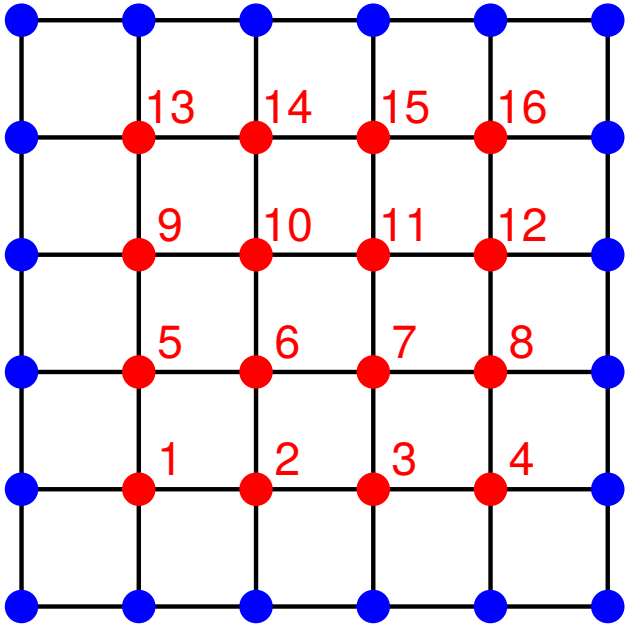
- git pull from GitHub to local machine

```
$ git pull
```

Floating Point Numbers

Recall: steady-state heat conduction

- Discretize an $N \times N$ grid with N^2 unknowns



- Assume temperature is fixed (and known) at each boundary point
- neighboring value = node that shares an edge with current node
- Every interior point has **exactly** 4 neighbors
- For each interior point, the steady state value is approximately the average of the 4 neighboring values

Steady-state heat conduction

System size: If $N = 100 \implies 100 \times 100$ grid $\implies$ linear system with $10,000$ equations, $A\vec{U} = \vec{b}$, where A is of size $10,000 \times 10,000$.

Question: How much disk space is required to store a $10,000 \times 10,000$ matrix of real numbers?

Steady-state heat conduction

System size: If $N = 100 \implies 100 \times 100$ grid $\implies$ linear system with $10,000$ equations, $A\vec{U} = \vec{b}$, where A is of size $10,000 \times 10,000$.

Question: How much disk space is required to store a $10,000 \times 10,000$ matrix of real numbers?

Answer: That depends on how many bytes are used for each real number.
1 byte = 8 bits, bit = “binary digit”

Steady-state heat conduction

System size: If $N = 100 \implies 100 \times 100$ grid $\implies$ linear system with $10,000$ equations, $A\vec{U} = \vec{b}$, where A is of size $10,000 \times 10,000$.

Question: How much disk space is required to store a $10,000 \times 10,000$ matrix of real numbers?

Answer: That depends on how many bytes are used for each real number.
1 byte = 8 bits, bit = "binary digit"

Assuming 8 bytes (64 bits) per value:

$$10,000 \times 10,000 \text{ matrix} \implies 10^8 \text{ entries matrix}$$

$$\implies 8 \times 10^8 \text{ bytes} = 8 \times 10^5 \text{ kB} = 8 \times 10^2 \text{ MB} = 800 \text{ MB}$$

Steady-state heat conduction

System size: If $N = 100 \implies 100 \times 100$ grid $\implies$ linear system with $10,000$ equations, $A\vec{U} = \vec{b}$, where A is of size $10,000 \times 10,000$.

Question: How much disk space is required to store a $10,000 \times 10,000$ matrix of real numbers?

Answer: That depends on how many bytes are used for each real number.
1 byte = 8 bits, bit = "binary digit"

Assuming 8 bytes (64 bits) per value:

$$10,000 \times 10,000 \text{ matrix} \implies 10^8 \text{ entries matrix}$$

$$\implies 8 \times 10^8 \text{ bytes} = 8 \times 10^5 \text{ kB} = 8 \times 10^2 \text{ MB} = 800 \text{ MB}$$

Note: less than $50,000$ values are non-zero $\implies$ 99.95% are 0.

Therefore, if we can get away with only storing non-zeros:

$$800\,\text{MB} \times 0.0005 = 0.4\,\text{MB} = 400\,\text{kB}$$

Measuring size and speed

Kilo	=	thousand $\left(10^3\right)$
Mega	=	million $\left(10^6\right)$
Giga	=	billion $\left(10^9\right)$
Tera	=	trillion $\left(10^{12}\right)$
Peta	=	10^{15}
Exa	=	10^{18}

Binary representation

Memory is divided into bytes, consisting of 8 bits each.

1 byte can hold $2^8 = 256$ distinct numbers:

```
00000000    = 0
00000001    = 1
00000010    = 2
...
11111111    = 255
```

Binary representation

Memory is divided into bytes, consisting of 8 bits each.

1 byte can hold $2^8 = 256$ distinct numbers:

```
00000000    = 0
00000001    = 1
00000010    = 2
...
11111111    = 255
```

8-bit representation: $(a_7)\,(a_6)\,(a_5)\,(a_4)\,(a_3)\,(a_2)\,(a_1)\,(a_0)$

Converting a binary representation to an integer: $x = \sum_{n=0}^{7} a_n 2^n$

$$11111111: \quad x = \sum_{n=0}^{7} 2^n = 1 + 2 + 4 + 8 + 16 + 32 + 64 + 128 = 255$$

A byte might represent integers, characters, colors, . . .

Modern computers: 4 bytes (32 bits) or 8 bytes (64 bits) used to represent integers and real numbers.

Integers

To store integers, need one bit for the sign (+ or -). In one byte, this leaves 7 bits.

Two's complement representation:

```
10000000 = -128
10000001 = -127
10000010 = -126
...
11111110 = -2
11111111 = -1
00000000 = 0
00000001 = 1
00000010 = 2
...
01111111 = 127
```

$$x = -a_7 2^7 + \sum_{n=0}^{6} a_n 2^n$$

Integers

To store integers, need one bit for the sign (+ or -). In one byte, this leaves 7 bits.

Two's complement representation:

```
10000000 = -128
10000001 = -127
10000010 = -126
...
11111110 = -2
11111111 = -1
00000000 = 0
00000001 = 1
00000010 = 2
...
01111111 = 127
```

$$x = -a_7\,2^7 + \sum_{n=0}^{6} a_n\,2^n$$

Advantage: binary addition works directly (e.g., $10 + (-6) = 4$)

	0	0	0	0	1	0	1	0
+	1	1	1	1	1	0	1	0
[1]	0	0	0	0	0	1	0	0

$$\Longrightarrow \quad 00000100 = 2^2\,(\text{base } 10) = 4\,(\text{base } 10)$$

Integers

Integers are typically stored in 8 bytes (64 bits). Values roughly between

$$-2^{63} \approx -10^{19} \qquad \text{and} \qquad 2^{63} \approx 10^{19}$$

can be stored.

In Python, larger integers will automatically be stored using more bytes.

```
$ python

>> 2**30
1073741824

>>> 2**62
4611686018427387904

>>> 2**63
9223372036854775808L

>>> 2**100
1267650600228229401496703205376L
```

Fixed point numbers

Idea: Use 64 bits for a real number but always assume *N* bits in integer part and *M* bits in fractional part.

Example: 5 digits for integer part and 6 digits in fractional part:

```
00003.141592     (pi)
00000.000314     (pi / 10000)
31415.926535     (pi * 10000)
```

Fixed point numbers

Idea: Use 64 bits for a real number but always assume N bits in integer part and M bits in fractional part.

Example: 5 digits for integer part and 6 digits in fractional part:

```
00003.141592     (pi)
00000.000314     (pi / 10000)
31415.926535     (pi * 10000)
```

Disadvantages:

- Precision depends on size of number
- Often many wasted bits (leading 0's)
- Limited range: often scientific problems involve very large or small numbers

Floating point numbers

Base 10 scientific notation:

$$\texttt{0.31278e-8} = 0.31278 \times 10^{-8} = 0.0000000031278$$

Mantissa: 0.31278, Exponent: -8

Floating point numbers

Base 10 scientific notation:

$$\texttt{0.31278e-8} = 0.31278 \times 10^{-8} = 0.0000000031278$$

Mantissa: 0.31278, Exponent: -8

Binary floating point numbers:

Example:

Mantissa: 0.101101, Exponent: -11011

means

$$\begin{aligned} 0.101101 &= 1 \times \left(2^{-1}\right) + 0 \times \left(2^{-2}\right) + 1 \times \left(2^{-3}\right) + 1 \times \left(2^{-4}\right) + 0 \times \left(2^{-5}\right) + 1 \times \left(2 \right. \\ &= 0.703125 \text{ (base 10)} \\ -11011 &= -\left\{1 \times \left(2^{4}\right) + 1 \times \left(2^{3}\right) + 0 \times \left(2^{2}\right) + 1 \times \left(2^{1}\right) + 1 \times \left(2^{0}\right)\right\} \\ &= -27 \text{ (base 10)} \end{aligned}$$

So the number is

$$0.703125 \times 2^{-27} \approx 5.238689482212067 \times 10^{-9}$$

Floating point numbers

- Python float is 8 bytes with IEEE standard representation
- 53 bits for mantissa and 11 bits for exponent (64 bits = 8 bytes)
- We can store 52 binay bits of precision:

$$\varepsilon_{\text{machine}} = 2^{-52} \approx 2.22 \times 10^{-16}$$

- Therefore, roughly 15 digits of precision
- Machine epsilon ($\varepsilon_{\text{machine}}$) is the relative distance between any two consecutive numbers on the floating point number system
- Therefore, simply representing a real number in a floating point number system produces an error: round-off error

Floating point numbers

- Since $2^{-52} \approx 2.22 \times 10^{-16}$, this corresponds to roughly 15 digits of precision

- For example:

```
>>> from numpy import pi

>>> pi
3.141592653589793

>>> 1000*pi
3141.592653589793

>>> pi/1000
0.003141592653589793
```

- Note: storage and arithmetic is done in base 2; converted to base 10 only when printed!

More Python

Python overview

Python is an object oriented general-purpose language

Advantages:

- Can be used interactively from a Python shell (similar to Matlab)
- Can also write scripts to execute from Unix shell
- Little overhead to start programming
- Powerful modern language
- Many modules are available for specialized work
- Good graphics and visualization modules (e.g., Matplotlib)
- Easy to combine with other languages (e.g., C and Fortran)
- Open source and runs on all platforms

Python overview

Disadvantage: Can be slow to do certain things, such as looping over arrays

This is because Python code is interpreted rather than compiled

Additional features to improve performance:

- Need to use suitable modules (e.g., NumPy, Pandas) for speed
- Can easily create custom modules from compiled code written in Fortran, C, etc. . .
- Can also use extensions such as Cython that makes it easier to mix Python with C code that will be compiled
- Python is often used for high-level scripts that e.g., download data from the web, run a set of experiments, collate and plot results

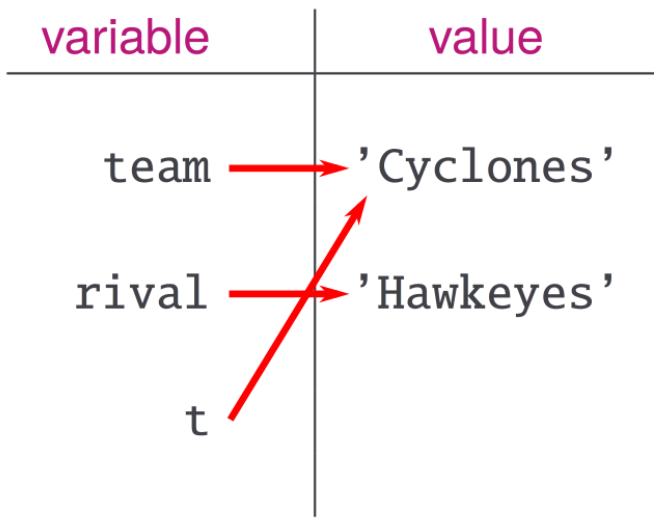
Python is an object-oriented language

- Nearly everything in Python is an object of some class
- The class description tells what data the object holds (attributes) and what operations (methods or functions) are defined to interact with the object
- Every "variable" is really just a pointer to some object. You can reset it to point to some other object at will
- So variables don't have "type" (e.g., integer, float, string) (but the objects they currently point to do)

Python is an object-oriented language

- Nearly everything in Python is an object of some class
- The class description tells what data the object holds (attributes) and what operations (methods or functions) are defined to interact with the object
- Every "variable" is really just a pointer to some object. You can reset it to point to some other object at will
- So variables don't have "type" (e.g., integer, float, string) (but the objects they currently point to do)

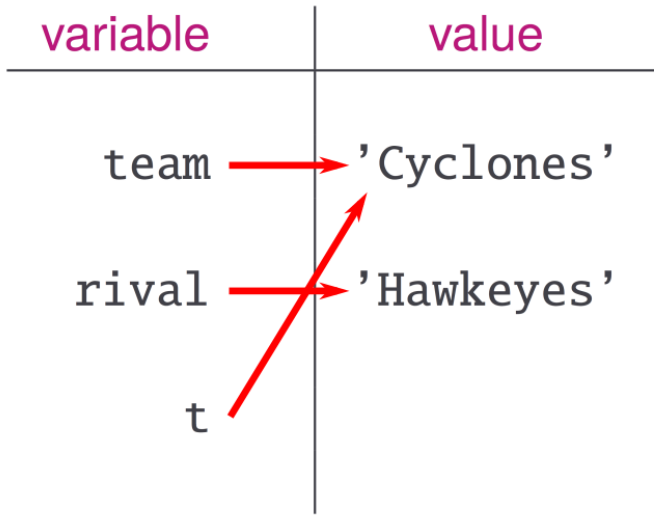
```
>>> team = 'Cyclones'
>>> print(team)
Cyclones
>>> rival = 'Hawkeyes'
>>> t = team
>>> print(t)
Cyclones
```



Python is an object-oriented language

- Nearly everything in Python is an object of some class
- The class description tells what data the object holds (attributes) and what operations (methods or functions) are defined to interact with the object
- Every "variable" is really just a pointer to some object. You can reset it to point to some other object at will
- So variables don't have "type" (e.g., integer, float, string) (but the objects they currently point to do)

```
>>> team = 'Cyclones'
>>> print(team)
Cyclones
>>> rival = 'Hawkeyes'
>>> t = team
>>> print(t)
Cyclones
>>> t = 4.0**3
>>> print(t)
64.0
```



Python is an object-oriented language

```
>>> x = -72.05
>>> print(id(x), type(x))
140485917044336 <class 'float'>

>>> x = 11.83
>>> print( id(x), type(x) )
140485886550992 <class 'float'>

>>> x = [-2,0,2]
>>> print( id(x), type(x) )
140485916373568 <class 'list'>

>>> x = [7,3,-1]
>>> print( id(x), type(x) )
140485915342720 <class 'list'>
```

Python is an object-oriented language

```
>>> x = [7,3,-1]

>>> print( id(x), type(x) )
140485915342720 <class 'list'>

>>> x.append(-17)

>>> x
[7, 3, -1, -17]

>>> print( id(x), type(x) )
140485915342720 <class 'list'>
```

Note: Object of type 'list' has a method 'append' that changes the object

A list is a mutable object (i.e., can be changed)

Python is an object-oriented language

```
>>> x = [4,-5,7]

>>> print( id(x), x )
140485917375872 [4, -5, 7]

>>> y = x

>>> print( id(y), y )
140485917375872 [4, -5, 7]

>>> y.append(-11)

>>> y
[4, -5, 7, -11]

>>> x
[4, -5, 7, -11]
```

Note: `x` and `y` point to the same object !!!

Make a copy of a list

```
>>> x = [5,6,7]

>>> print( id(x), x )
140485915342720 [5, 6, 7]

>>> y = list(x)   # creates a new list object (contrast y = x)

>>> print( id(y), y )
140485917393920 [5, 6, 7]

>>> y.append(-3)

>>> y
[5, 6, 7, -3]

>>> x
[5, 6, 7]
```

Integers and floats are immutable

```
>>> x = 8.45

>>> print( id(x), x )
140485886550544 8.45

>>> y = x

>>> print( id(y), y )
140485886550544 8.45

>>> y = y + 1

>>> print( id(y), y )
140485886550992 9.45

>>> print( id(x), x )
140485886550544 8.45
```

Note: in order to change `y`, Python created a new object and left the old object intact (`x` still points to that object)

Lists

Note: The elements of a list can be any objects (need not be same type):

```
>>> L = [3, 4.5, 'abc', [1,2]]

>>> L[0]
3

>>> L[1]
4.5

>>> L[2]
'abc'

>>> L[3]
[1, 2]

>>> L[3][0]   # element 0 of L[3]
1
```

Lists

Lists have several built-in methods (append, insert, sort, pop, reverse, remove, . . .)

```
>>> L = [3, 4.5, 'abc', [1,2]]

>>> L2 = L.pop(2)

>>> L2
'abc'

>>> L
[3, 4.5, [1, 2]]
```

Note: L still points to the same object, but it has changed.

Lists and tuples

```
>>> L = [3, 4.5, 'abc']

>>> L[0] = 'xy'

>>> L
['xy', 4.5, 'abc']

>>> T = (3, 4.5, 'abc')

>>> T[0]
3

>>> T[0] = 'xy'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Note: a tuple is like a list but is immutable

Lab assignment

- Practice operations on a Python list: append(), clear(), copy(), count(), extend(), index(), insert(), pop(), remove(), reverse(), sort().
- Generate a random list L of length N compare the performance of the following operations by measuring times with increasing N: N, $2N$, $4N$, $8N$, ..., explain what you observe.
  1. Cost of removing elements from lists:
     `L.pop()`
     `L.pop(0)`
  2. Slicing vs explicit copy of a list:
     `A = L[:]`
     `B = list(L)`
  3. In-place vs out-of-place modification:
     `L.reverse()`
     `R = L[::-1]`
- Update your git repository by creating a folder of lecture 03, adding files and pushing to GitHub (GitLab, Bitbucket, ...)
- Submit both code and screen shots