HIGH-PERFORMANCE SCIENTIFIC COMPUTING (HPSC) ME522

Instructor: Gaurav Bhutani

SMME, IIT Mandi

Feb-Jun 2023 semester

Venue: to be announced

Online: https://meet.google.com/xyn-osvu-yys

Lecture plan

- Course essentials
- Goals and learning outcomes
- Pre-requisites and software requirements
- Short demonstration

Course essentials

Notes

- Class: 4 hours per week (2 sessions).
- Venue to be announced.
- One hr lecture, followed by 1 hour lab / hands-on session. Use your own laptop.
- Teaching supporter: Ayush Sahu (SMME Metch
  Help with labs, installing software, obtaining material, announcements
- Announcements: Course mailing list / Google chat group
- Learn from peers
- Moodle – all course slides and link to material
- Codes on Github as we create
- Virtual machines on IIT Mandi Cloud

Evaluation

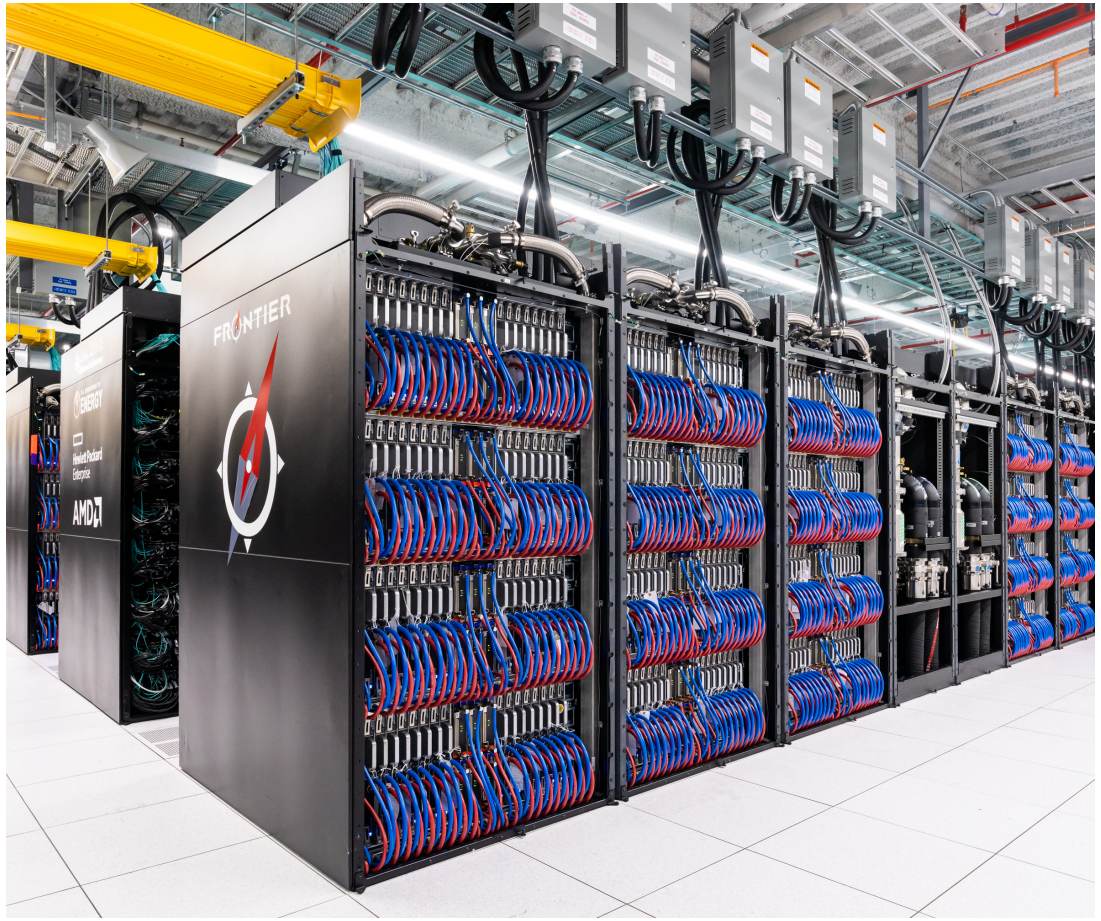
- Lab/class attendance and random lab viva (20)
- Midsem (30)
  - Written (10)
  - Lab exam (20)
- Endsem (50)
  - Written (20)
  - Lab exam (30)

Goals

HPC

- **High Performance Computing (HPC)** often means heavy-duty computing on clusters or supercomputers with 100s of thousands of cores.
- *"World's fastest computer"*
- #1. Frontier – HPE Cray (Oak Ridge National Lab, USA)
  8.7M cores ≈ 1100 Petaflops; 21MW power
- #3. Leonardo – Atos (CINECA, Italy)
  1.4M cores ≈ 174 Petaflops; 5.6MW
- Param Himalaya – Atos (IIT Mandi, India)
  800 Teraflops; 150kW

See http://top500.org for current list.

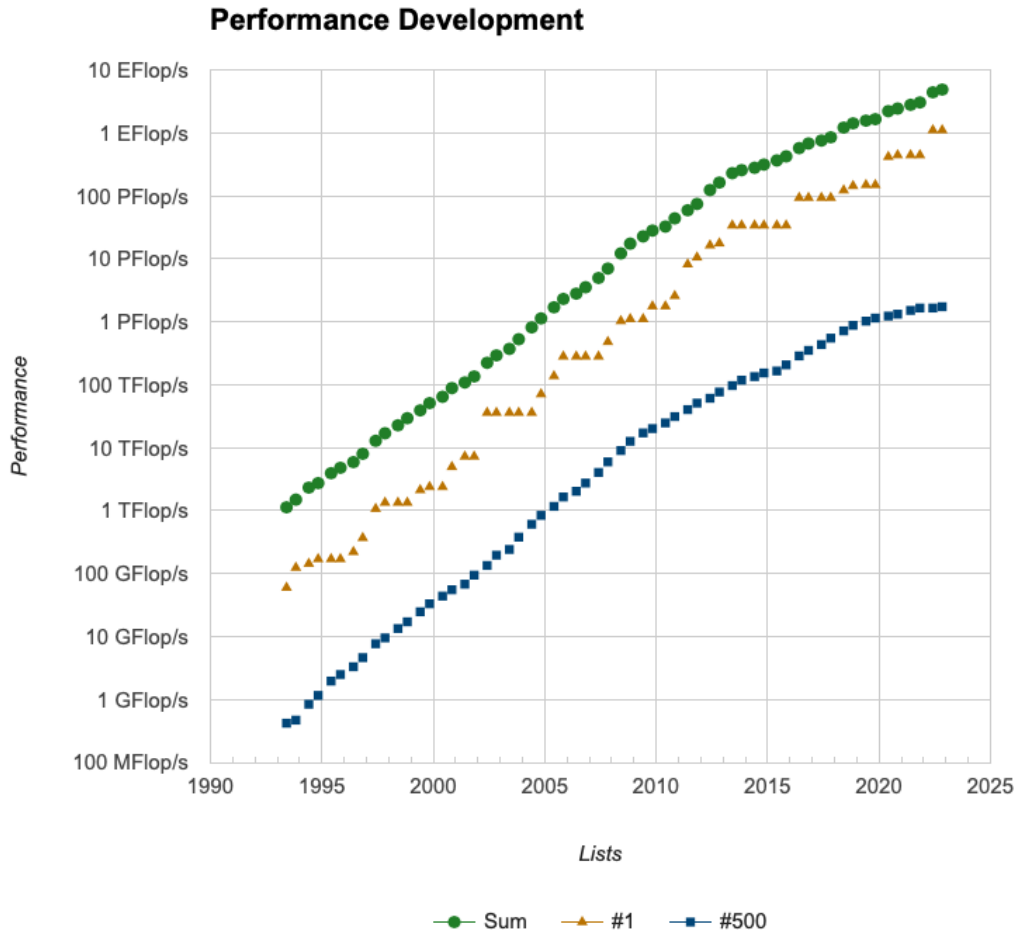


Increasing speed

- Moore's Law: Processor speed doubles every 18 months.
  ⇒ factor of 1024 in 15 years.
- Going forward: Number of cores doubles every 18 months

- Top: Total computing power of top 500 computers
- Middle: #1 computer
- Bottom: #500 computer

http://www.top500.org



Our focus

- Our focus is more modest, but we will cover material that is:
  - Essential to know if you eventually want to work on supercomputers
  - Extremely useful for any scientific computing project, even on a laptop.
- Focus on scientific computing as opposed to other computationally demanding domains, for which somewhat different tools might be best.

Learning outcomes

Efficient processing

- Basic computer architecture, e.g. floating point arithmetic, cache hierarchies
- Using Unix (or Linux)
- Language issues, e.g. compiled vs. interpreted, object oriented, etc.
- Specific languages: Python (for scripting), Fortran 90/95 (for fast processing)
- Parallel computing with OpenMP, MPI
- Not included: GPU-based parallelisation

Good software practices

- Version control using Git, github
- Makefiles
- Debuggers, code testing
- Reproducability
- Use of high-performance computing (HPC) clusters

Strategy

- So much material, so little time…
- Concentrate on basics, simple motivating examples.
- Get enough hands-on experience to be comfortable experimenting further and learning much more on your own.
- Learn what's out there to help select what's best for your needs. New languages are introduced with time – similar ideas though.
- Teach many things "by example" as we go along.
- You'll be expected to read supplementary notes when they are provided.
- No specific book for the course. Internet search for help will be useful.

Pre-requisites & software requirements

Pre-requisites

- Some programming experience in some language, e.g., Matlab, C.
- You should be comfortable:
  - editing a file containing a program and executing it,
  - using basic structures like loops, if-then-else, input-output,
  - writing subroutines or functions in some language
- You are not expected to know Python or Fortran.
- Some basic knowledge of linear algebra, e.g.:
  - what vectors and matrices are and how to multiply them
  - How to go about solving a linear system of equations
- Comfort level for learning new software.

Software requirements

- You will need access to a computer with a number of things on it. All open-source software.
- Note: Unix is often required for scientific computing.
- Windows: Many tools we'll use can be used with Windows, but learning Unix is part of this class.
- Options:
- Install everything you'll need on your own computer.
- Install VirtualBox and use the Virtual Machine (VM) created for this class.
- Use Amazon Web Services with an Amazon Machine Image (AMI) which will be created for this class. Other cloud computing services are also available.

Know the class

- A short intro by various class members
  - Name and program enrolled in
  - What are your academic interests and what research projects you are working on? For PG students please mention your research group at IIT Mandi.
  - Why interested in this course? What are your expectations from the course?

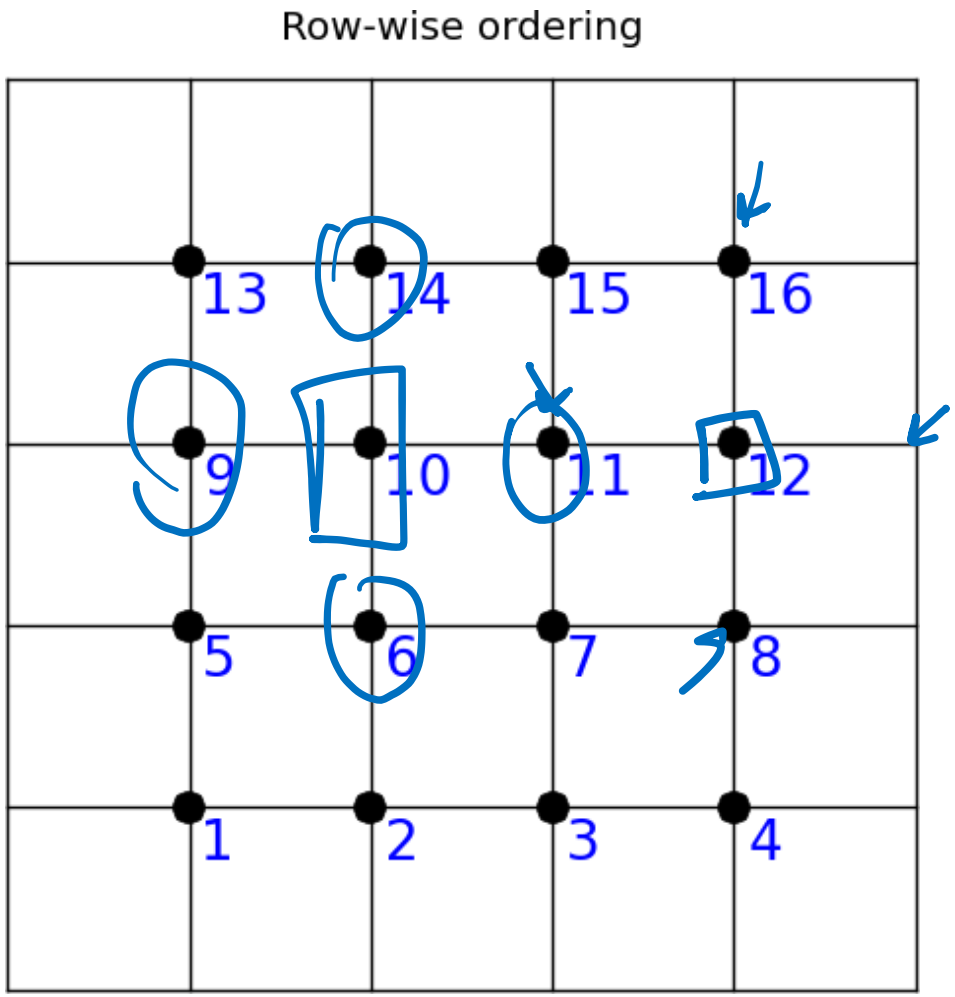
Short demonstration

Demo code

```
$ cd <folder name>
$ export HPSC=$PWD
$ cd $HPSC/lecture1

$ make plots

$ display *.png
```

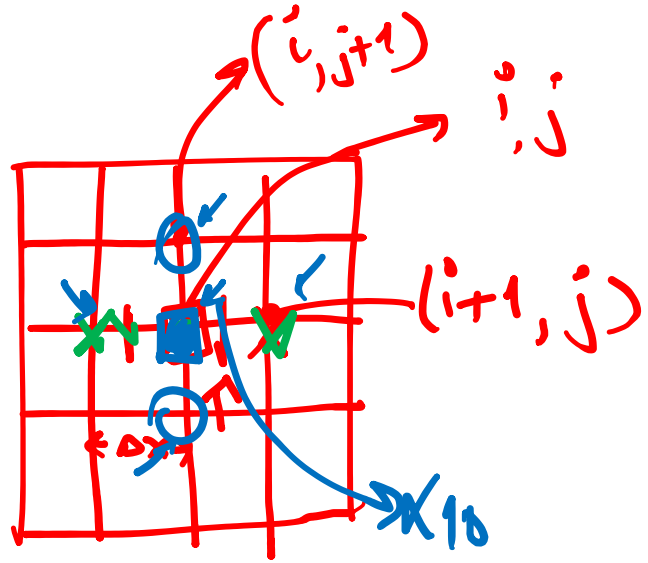
Steady-state heat conduction

- Discretize on an $N \times N$ grid with N^2 unknowns
- Assume temperature is fixed (and known) at each point on boundary.
- At interior points, the steady state value is (approximately) the average of the 4 neighbouring values.



$$\nabla^2 u = 0$$

2-D $\quad \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$



$$[A]_{N^2 \times N^2} \{x\} = \{b\}$$

$$\begin{bmatrix} x_8 \\ x_9 \\ x_{10} \\ x_{11} \\ x_{12} \end{bmatrix}$$

N^2 120×120

144 00

$$\frac{\partial^2 u}{\partial x^2} = \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{(\Delta x)^2}$$

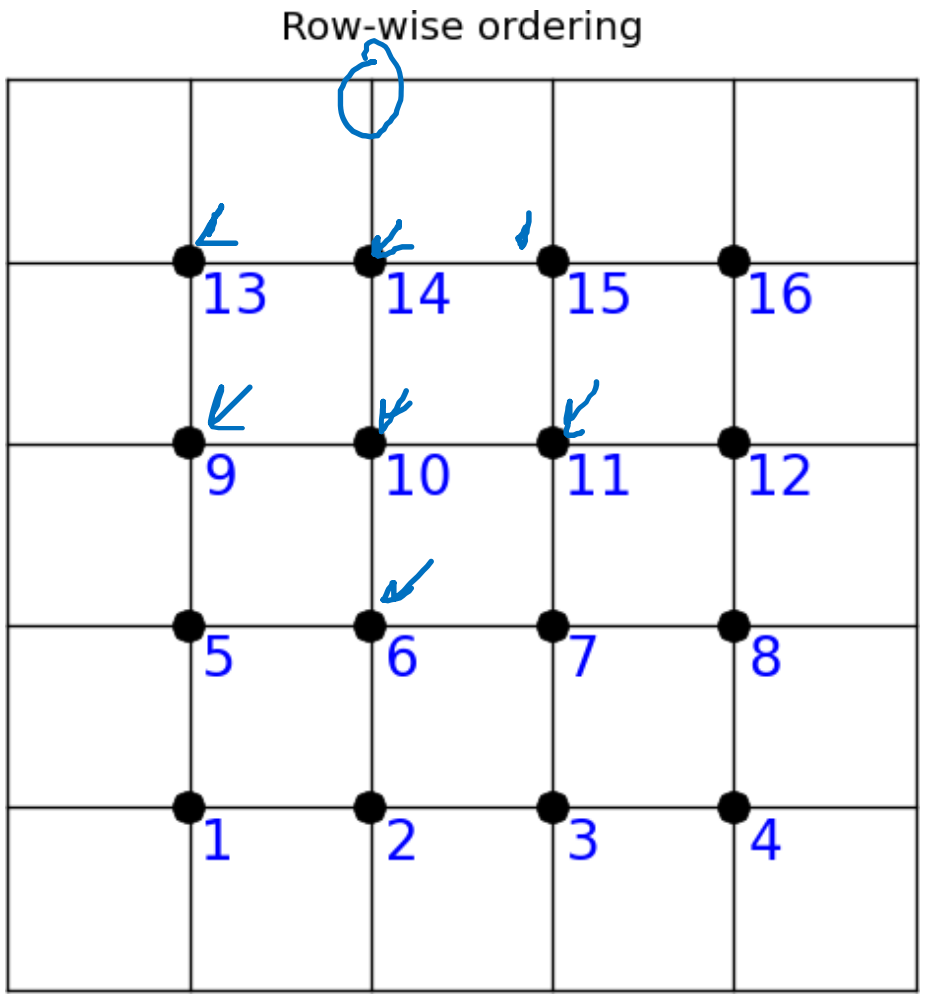
$$u_{i,j} = f(u_{i-1,j}, u_{i+1,j}, u_{i,j-1}, u_{i,j+1})$$

Steady-state heat conduction

$$u_{i,j} = \frac{1}{4}\left(u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1}\right)$$

- Holds for i, j = 1, 2, . . . , N with $u_{0,j}$ known on boundary. Gives a linear system Au = b, with N^2 equations N^2 unknowns. Matrix A is $N^2 \times N^2$, for N = 120, N^2 = 14400.
- Very sparse: each row of matrix A has at most 5 nonzeros. Gaussian elimination is not the best approach.
- Jacobi method (not the best method)

$$u_{(i,j)}^{[k+1]} = \frac{1}{4}\left(u_{(i-1,j)}^{[k]} + u_{(i+1,j)}^{[k]} + u_{(i,j-1)}^{[k]} + u_{(i,j+1)}^{[k]}\right)$$



Speedup of linear solvers

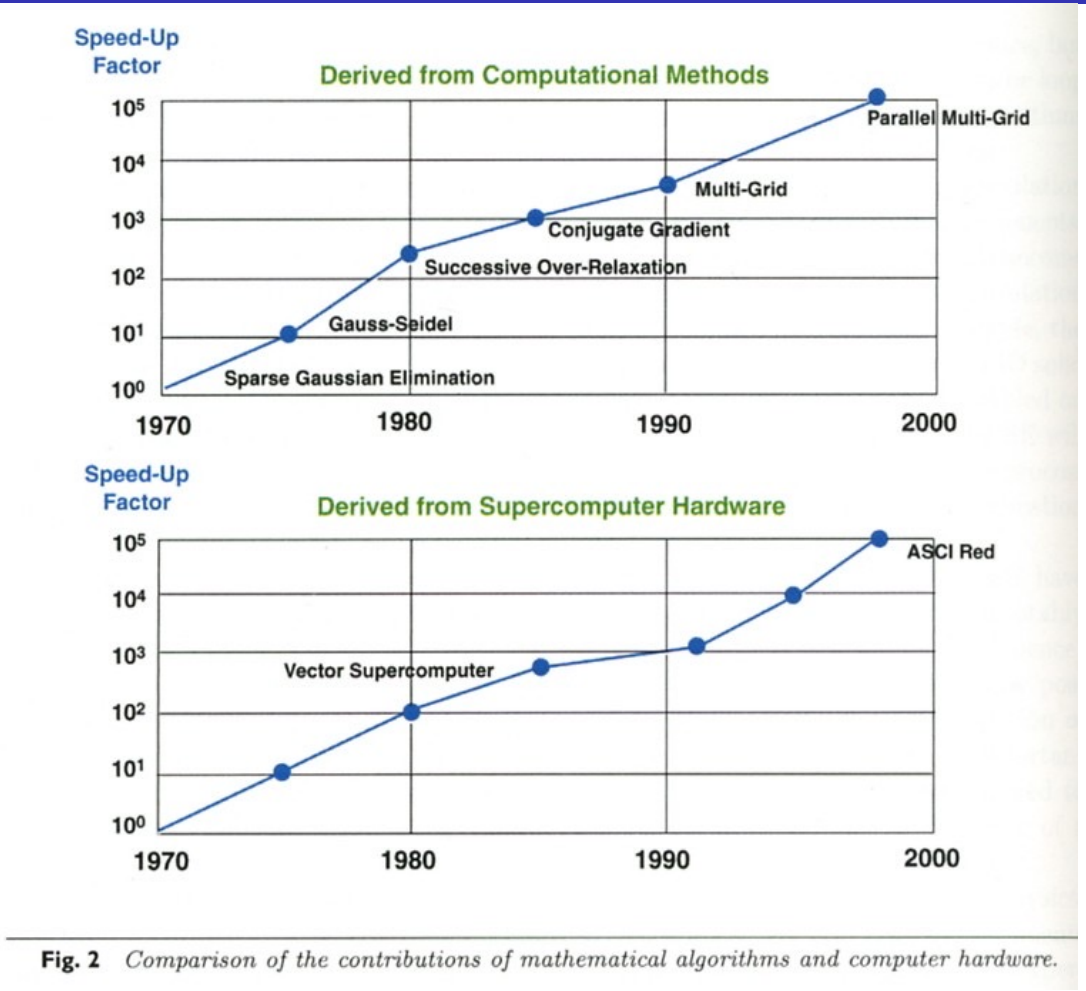


Fig. 2 *Comparison of the contributions of mathematical algorithms and computer hardware.*

Source: SIAM Review 43(2001), p. 168.

- Exponential increase in the speed of numerical algorithms.
- Exponential increase in the computer hardware performance with time.

Class virtual machine

- Available on Google Drive - `https://drive.google.com/drive/folders/1mnMJoJNqiOpcuFzVZeW8EfoxqlqHWWVG?usp=sharing`
- Username: hpsc; password: me522
- This file is large! About 5 GB compressed. After unzipping, about 10 GB.

References

- Slides are adapted from HPSC, AM483, Uni of Washington by Randall J Leveque (https://faculty.washington.edu/rjl/teaching.html)
  Refer to his page for more useful material and notes.

HPSC 101 — Lecture 2

Outline:

- Binary storage, floating point numbers
- Version control — main ideas
- Distributed version control, e.g., git

Reading:

- Storing information in binary
- version control & git
- Github

Outline of course

Some topics to be covered:

- Unix
- Version control (git)
- Python
- Compiled vs. interpreted languages
- Fortran 90
- Makefiles
- Parallel computing
- OpenMP
- MPI (message passing interface)
- Graphics / visualization

Unix (and Linux, Mac OS X, etc.)

See the Software Carpentry Unix training material

Unix commands will be introduced as needed and mostly discussed in the context of other things.

Some important ones...

- cd, pwd, ls, mkdir
- mv, cp

Commands are typed into a terminal window shell,

We will use bash. Prompt will be denoted $, e.g.

```
$ cd ..
```

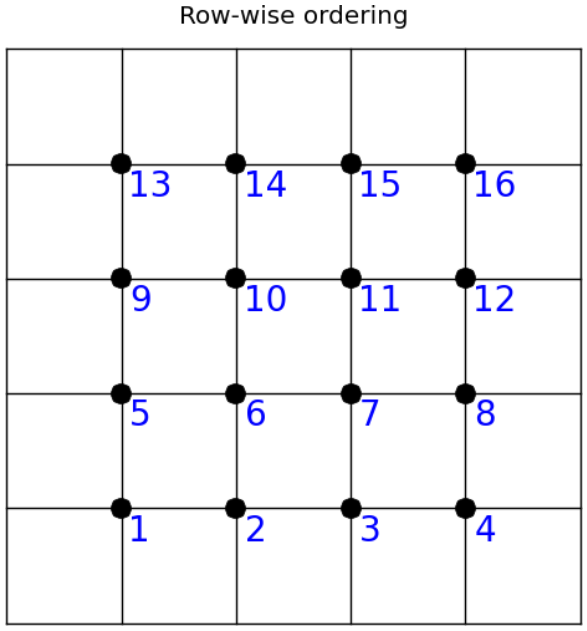
Other references and sources

- Wikipedia often has good intros and summaries.
- Software Carpentry, particularly these videos
  (also see their YouTube playlist for lessons)

- Other courses at universities or supercomputer centers.

Steady state heat conduction

Discretize on an $N \times N$ grid with N^2 unknowns:



Assume temperature is fixed (and known) at each point on boundary.

At interior points, the steady state value is (approximately) the average of the 4 neighboring values.

Storing a big matrix

Recall: Approximating the heat equation on a 100×100 grid gives a linear system with $10,000$ equations, $Au = b$ where the matrix A is $10,000 \times 10,000$.

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

Storing a big matrix

Recall: Approximating the heat equation on a 100×100 grid gives a linear system with $10,000$ equations, $Au = b$ where the matrix A is $10,000 \times 10,000$.

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

It depends on how many bytes are used for each real number.

1 byte = 8 bits, bit = "binary digit"

Storing a big matrix

Recall: Approximating the heat equation on a 100×100 grid gives a linear system with $10,000$ equations, $Au = b$ where the matrix A is $10,000 \times 10,000$.

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

It 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:

A $10,000 \times 10,000$ matrix has 10^8 elements,

so this requires 8×10^8 bytes = 800 MB.

Storing a big matrix

Recall: Approximating the heat equation on a 100×100 grid gives a linear system with $10,000$ equations, $Au = b$ where the matrix A is $10,000 \times 10,000$.

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

It 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:

A $10,000 \times 10,000$ matrix has 10^8 elements,

so this requires 8×10^8 bytes = 800 MB.

And less than $50,000$ values are nonzero, so 99.95% are 0.

Measuring size and speed

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

Computer memory

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

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

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

Might represent integers, characters, colors, etc.

Usually programs involve integers and real numbers that require more than 1 byte to store.

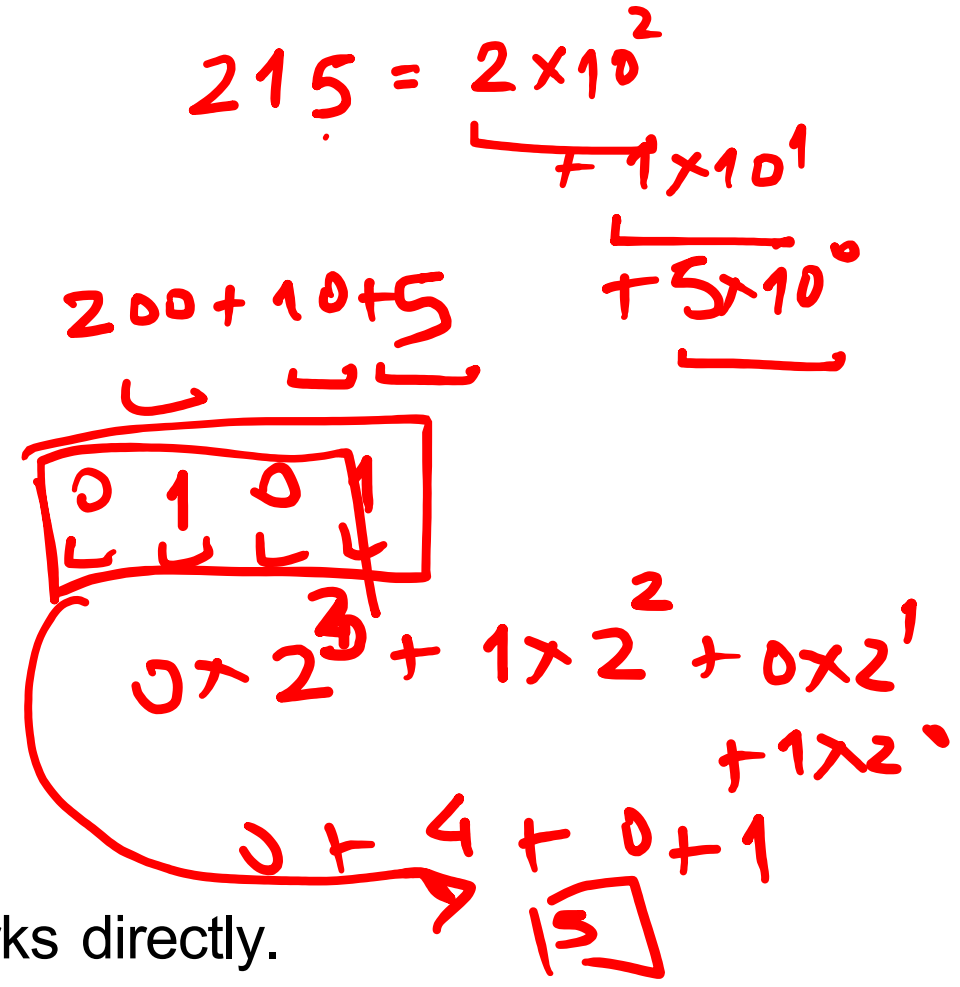
Often 4 bytes (32 bits) or 8 bytes (64 bits) used for each.

Integers

To store integers, need one bit for the sign (+ or −)
In one byte this would leave 7 bits for binary digits.

Two-complements representation used:

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



Advantage: Binary addition works directly.

Integers

Integers are typically stored in 4 bytes (32 bits). Values between roughly -2^{31} and 2^{31} can be stored.

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

Note: special software for arithmetic, may be slower!

```
$ python

>>> 2**30
1073741824

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

Note L on end!

Fixed point notation

Use, e.g. 64 bits for a real number but always assume N bits in integer part and M bits in fractional part.

Analog in decimal arithmetic, e.g.:

5 digits for integer part and
6 digits in fractional part

Could represent, e.g.:

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

Fixed point notation

Use, e.g. 64 bits for a real number but always assume N bits in integer part and M bits in fractional part.

Analog in decimal arithmetic, e.g.:

5 digits for integer part and
6 digits in fractional part

Could represent, e.g.:

```
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 real numbers

Base 10 scientific notation:

$$\texttt{0.2345e-18} = 0.2345 \times 10^{-18} = 0.000000000000000002345$$

Mantissa: 0.2345, Exponent: -18

Floating point real numbers

Base 10 scientific notation:

$$\texttt{0.2345e-18} = 0.2345 \times 10^{-18} = 0.000000000000000002345$$

Mantissa: 0.2345, Exponent: -18

Binary floating point numbers:

Example: Mantissa: 0.101101, Exponent: -11011 means:

$$\begin{aligned} 0.101101 &= 1(2^{-1}) + 0(2^{-2}) + 1(2^{-3}) + 1(2^{-4}) + 0(2^{-5}) + 1(2^{-6}) \\ &= 0.703125 \text{ (base 10)} \\ -11011 &= -1(2^{4}) + 1(2^{3}) + 0(2^{2}) + 1(2^{1}) + 1(2^{0}) = -27 \text{ (base 10)} \end{aligned}$$

So the number is

$$0.703125 \times 2^{-27} \approx 5.2386894822120667 \times 10^{-9}$$

Floating point real numbers

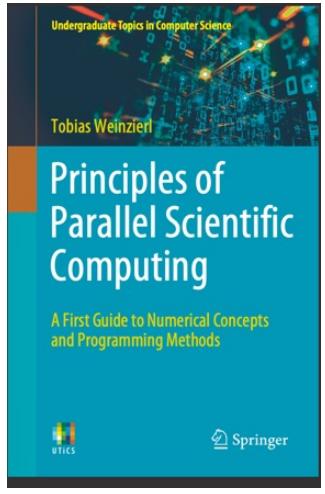
Python float is 8 bytes with IEEE standard (754) representation.

53 bits for mantissa and 11 bits for exponent (64 bits = 8 bytes).

We can store 52 binary bits of precision.

$2^{-52} \approx 2.2 \times 10^{-16} \Longrightarrow$ roughly 15 digits of precision.

Read Chapter 4: Floating Point Numbers from Principles of Parallel Scientific Computing by Tobias Weinzierl



Floating point real numbers

Since $2^{-52} \approx 2.2 \times 10^{-16}$ this corresponds to roughly 15 digits of precision.

For example:

```
>>> from numpy import pi
>>> pi
3.1415926535897931

>>> 1000 * pi
3141.5926535897929

>>> pi/1000
0.0031415926535897933
```

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

Version control systems

Originally developed for large software projects with many developers.

Also useful for single user, e.g. to:

- Keep track of history and changes to files,
- Be able to revert to previous versions,
- Keep many different versions of code well organized,
- Easily archive exactly the version used for results in publications,
- Keep work in sync on multiple computers.

Server-client model:

Original style, still widely used (e.g. CVS, Subversion)

One central repository on server.

Developers' workflow (simplified!):

- Check out a working copy,
- Make changes, test and debug,
- Check in (commit) changes to repository (with comments). This creates new version number.
- Run an update on working copy to bring in others' changes.

The system keeps track of diffs from one version to the next (and info on who made the changes, when, etc.)

A changeset is a collection of diffs from one commit.

Server-client model:

Only the server has the full history.

The working copy has:

- Latest version from repository (from last checkout, commit, or update)
- Your local changes that are not yet committed.

Server-client model:

Only the server has the full history.

The working copy has:

- Latest version from repository (from last checkout, commit, or update)
- Your local changes that are not yet committed.

Note:

- You can retrieve older versions from the server.
- Can only *commit* or *update* when connected to server.
- When you *commit*, it will be seen by anyone else who does an *update* from the repository.

Often there are trunk and branches subdirectories.

Distributed version control

Git uses a distributed model:

When you clone a repository you get all the history too,

All stored in .git subdirectory of top directory.

Usually don't want to mess with this!

Ex: (backslash is continuation character in shell)

```
$ git clone \
   https://bitbucket.org/rjleveque/uwhpsc \
   mydirname
```

will make a complete copy of the class repository (from uwhpsc) and call it mydirname. If mydirname is omitted, it will be called uwhpsc.

This directory has a subdirectory .git with complete history.

Distributed version control

Git uses a distributed model:

- git commit commits to your clone's .git directory.
- git push sends your recent changesets to another clone
  by default: the one you cloned from (e.g. bitbucket),
  but you can push to any other clone (with write permission).
- git fetch pulls changesets from another clone
  by default: the one you cloned from (e.g. bitbucket)
- git merge applies changesets to your working copy

Next lecture: simpler example of using git in a single directory.

Distributed version control

Advantages of distributed model:

- You can commit changes, revert to earlier versions, examine history, etc. without being connected to server.
- Also without affecting anyone else's version if you're working collaboratively. *Can commit often while debugging.*

Distributed version control

Advantages of distributed model:

- You can commit changes, revert to earlier versions, examine history, etc. without being connected to server.
- Also without affecting anyone else's version if you're working collaboratively. *Can commit often while debugging.*
- No problem if server dies, every clone has full history.

Distributed version control

Advantages of distributed model:

- You can commit changes, revert to earlier versions, examine history, etc. without being connected to server.
- Also without affecting anyone else's version if you're working collaboratively. *Can commit often while debugging.*
- No problem if server dies, every clone has full history.

For collaboration will still need to push or fetch changes eventually and git merge may become more complicated.

Bitbucket

You can uwhpsc examine class repository at:
`https://bitbucket.org/rjleveque/uwhpsc`

Experiment with the “Source”, “Commits” tabs...

See also Software Carpentry for more git references and tutorials.

Github

https://github.com

Another repository for hosting git repositories.

Many open sources projects use it, including Linux kernel.

(Git was developed by Linus Torvalds for this purpose!)

Github

https://github.com

Another repository for hosting git repositories.

Many open sources projects use it, including Linux kernal.

(Git was developed by Linus Torvalds for this purpose!)

Many open source scientific computing projects use github, e.g.

- Ipython, Jupyter notebook
- NumPy, Scipy, matplotlib

Other tools for git

The gitkraken tool is useful for working with git in GUI.

This is not installed on the VM, but you can install it from their website by downloading the .deb or

```
sudo snap install gitkraken
```

Demo...

Hands-on exercise

1. Explore the Software Carpentry Github repo page for Unix Shell training
2. Modify a file in the SC Unix repo using gitkraken, then commit the change, then checkout the previous version again to access the version of the file you cloned.

3. Apply for HPC account over the IIT Mandi HPC website

https://sites.google.com/iitmandi.ac.in/hpc-iit-mandi

Hands-on exercise - bash

1. List the largest file in the folder `shell-novice/shell-lesson-data/exercise-data/proteins`, print its name, then copy it a new folder called `largest`.
2. What does: ls pe?tane.* give? And cat pe?tane.* ?

3. Count the number of files and directories in a given folder.

   Generate a new file with the text: “the number of files in the folder is xx”.
4. Write a bash script/loop to create a file that contains the second last line of all pdb files available in a folder
5. in a folder containing 10 files, write a bash script to prepend the name of each file with the year it was created.

6. What does this command do: cut -d, -f 2 animals.csv | sort | uniq –c? animals.csv is in `shell-novice/shell-lesson-data/exercise-data/animal-counts`

Answers

```
echo cp $(ls -S | head -1) ./largest

Ls | wc -w
echo The number of the files is $(ls | wc -w) >
blah.txt

for filename in *.pdb
do
tail -2 $filename | head -1 >> tt.txt
Done

for filename in *.pdb; do echo mv $filename
$(ls -lT $filename | cut -w -f 9)-$filename;
done
```

Hands-on exercise - bash

1. You have a set of 20 text files and would like to concatenate them into one super text file.
2. Write a script that prints the name of the file with the longest name in a given directory.

Answer:

```
for filename in *.pdb
do
  echo ${filename}
  cat ${filename} >> total.pdb
Done
```

```
for line in $(ls); do echo ${#line} ${line} >>
namelen.txt; done
```

HPSC 101 — Lecture 3

This lecture:

- computing square roots
- Python demo
- git demo

Computing square roots

Hardware arithmetic units can add, subtract, multiply, divide.

Other mathematical functions usually take some software.

Computing square roots

Hardware arithmetic units can add, subtract, multiply, divide.

Other mathematical functions usually take some software.

Example: Compute $\sqrt{2} \approx 1.4142135623730951$

In most languages, sqrt(2) computes this.

```
>>> from numpy import sqrt
>>> sqrt(2.)
```

One possible algorithm to approximate $s = \sqrt{x}$

```
s = 1.     # or some better initial guess
for k in range(kmax):
    s = 0.5 * (s + x/s)
```

where `kmax` is some maximum number of iterations.

Note: In Python, range(N) is $[0, 1, 2, , \ldots, N-1]$.

One possible algorithm to approximate $s = \sqrt{x}$

```
s = 1.     # or some better initial guess
for k in range(kmax):
    s = 0.5 * (s + x/s)
```

where `kmax` is some maximum number of iterations.

Note: In Python, range(N) is $[0, 1, 2, , \ldots, N - 1]$.

Why this works...

If $s < \sqrt{x}$ then $x/s > \sqrt{x}$

If $s > \sqrt{x}$ then $x/s < \sqrt{x}$

One possible algorithm to approximate $s = \sqrt{x}$

```
s = 1.     # or some better initial guess
for k in range(kmax):
    s = 0.5 * (s + x/s)
```

where `kmax` is some maximum number of iterations.

Note: In Python, range(N) is $[0, 1, 2, , \ldots, N - 1]$.

Why this works...

If $s < \sqrt{x}$ then $x/s > \sqrt{x}$

If $s > \sqrt{x}$ then $x/s < \sqrt{x}$

In fact this is Newton's method to find root of $s^2 - x = 0$.

Newton's method

Problem: Find a solution of $f(s) = 0$ (zero or root of f)

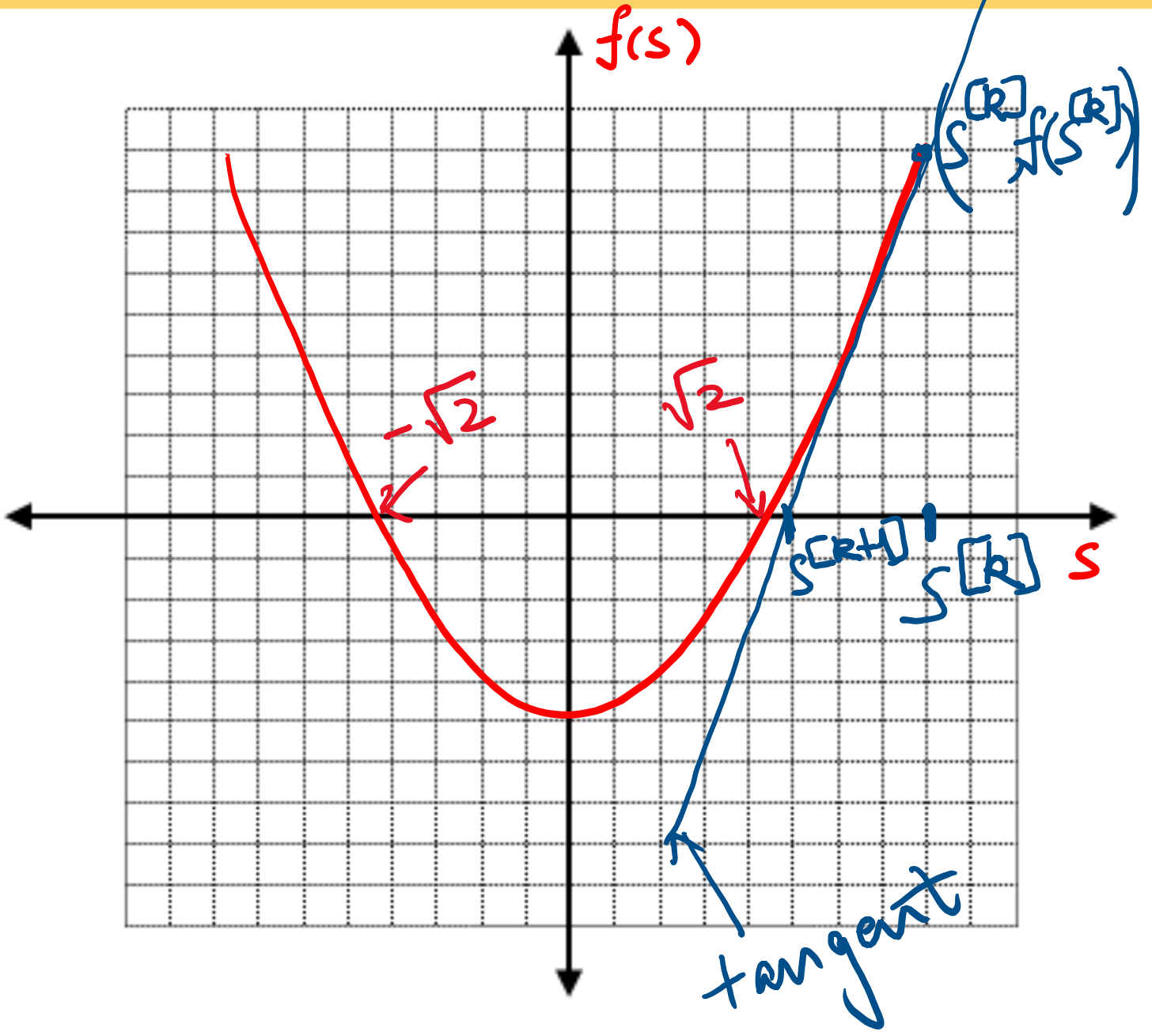
Idea: Given approximation $s^{[k]}$,

approximate $f(s)$ by a linear function,
the tangent line at $(s^{[k]}, f(s^{[k]}))$.

Find unique zero of this function and use as $s^{[k+1]}$.

Newton's method

$f(s) = s^2 - x$



Newton's method

Problem: Find a solution of $f(s) = 0$ (zero or root of f)

Idea: Given approximation $s^{[k]}$,
approximate $f(s)$ by a linear function,
the tangent line at $(s^{[k]}, f(s^{[k]}))$.

Find unique zero of this function and use as $s^{[k+1]}$.

Updating formula:

$$s^{[k+1]} = s^{[k]} - \frac{f(s^{[k]})}{f'(s^{[k]})}$$

$$s = s - \frac{s^2 - x}{2s} = \frac{1}{2}\left(s + \frac{x}{s}\right)$$

Demo...

Goals:

- Develop our own version of `sqrt` function.
- Start simple and add complexity in stages.
- Illustrate some Python programming.
- Illustrate use of git to track our development

HPSC 101 — Lecture 4

This lecture:

- Continue demo: computing square roots
- More about Python, IPython
- More about git, Unix

Installing new software on VM

Some useful additions...

```
$ sudo apt-get install xxdiff
(venv) $ pip install sympy

(venv) $ pip install pytest
```

Demo

Demo

Lab / HW

Python and git (*30 minutes*)

- Fork the hpsc_2023 repository into your own Githib account
- Create mycubrt() (be careful with negative x values here). This file must sit in code*/python folder*
- Create unit tests and run the tests using nose. Make sure all the tests pass.
- Add and commit this work
- Push this work to your Github repository

HPSC101 — Lecture 5

This lecture:

- Python concepts and objects
- Data types, lists, tuples
- Modules
- Demo — plotting and IPython notebook

Python

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
- Easy to combine with other languages (e.g. Fortran)
- Open source and runs on all platforms

Python

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

Code is interpreted rather than compiled

Need to use suitable modules (e.g. NumPy) 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.

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.

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.)

Object-oriented language

```
>>> x = 3.4
>>> print id(x), type(x)   # id() returns memory
8645588 <type 'float'>     address

>>> x = 5
>>> print id(x), type(x)
8401752 <type 'int'>

>>> x = [4,5,6]
>>> print id(x), type(x)
1819752 <type 'list'>

>>> x = [7,8,9]
>>> print id(x), type(x)
1843808 <type 'list'>
```

Object-oriented language

```
>>> x = [7,8,9]
>>> print id(x), type(x)
1843808 <type 'list'>

>>> x.append(10)
>>> x
[7, 8, 9, 10]
>>> print id(x), type(x)
1843808 <type 'list'>
```

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

A list is a mutable object.

Object-oriented language — gotcha

```
>>> x = [1,2,3]
>>> print id(x), x
1845768 [1, 2, 3]

>>> y = x
>>> print id(y), y
1845768 [1, 2, 3]

>>> y.append(27)
>>> y
[1, 2, 3, 27]

>>> x
[1, 2, 3, 27]
```

Note: x and y point to the same object!

Making a copy

```
>>> x = [1,2,3]
>>> print id(x), x
1845768 [1, 2, 3]

>>> y = list(x)     # creates new list object
>>> print id(y), y
1846488 [1, 2, 3]

>>> y.append(27)

>>> y
[1, 2, 3, 27]

>>> x
[1, 2, 3]
```

integers and floats are immutable

If `type(x) in [int,float]`, then setting `y = x` creates a new object `y` pointing to a new location.

```
>>> x = 3.4
>>> print id(x), x
8645588 3.4

>>> y = x
>>> print id(y), y
8645588 3.4

>>> y = y+1

>>> print id(y), y
8463377 4.4

>>> print id(x), x
8645588 3.4
```

Lists

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

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

Indexing starts at 0:

```
>>> L[0]
3

>>> L[2]
'abc'

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

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

Lists

Lists have several built-in methods, e.g. append, insert, sort, pop, reverse, remove, etc.

```
>>> 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.

In IPython: Type L. followed by Tab to see all attributes and methods.

Lists and tuples

```
>>> L = [3, 4.5, 'abc']
>>> L[0] = 'xy'
>>> L
['xy', 4.5, 'abc']
```

A tuple is like a list but is immutable:

```
>>> 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
```

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: to use square root function:

```
>>> from numpy import sqrt
>>> sqrt(2.)
1.4142135623730951
```

Python modules

When 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, using the sys module:

```
>>> import sys
>>> sys.path   # returns list of directories
['', '/usr/bin', ....]

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

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

Python modules

Different ways to import:

```
>>> from numpy import sqrt
>>> sqrt(2.)
1.414213562373095l

>>> from numpy import *
>>> sqrt(2.)
1.414213562373095l

>>> import numpy
>>> numpy.sqrt(2.)
1.414213562373095l

>>> import numpy as np
>>> np.sqrt(2.)
1.414213562373095l
```

Graphics and Visualization

Many tools are available for plotting numerical results.

Some open source Python options:

- matplotlib for 1d plots and
  2d plots (e.g. pseudocolor, contour, quiver)
- Mayavi for 3d plots (curves, surfaces, vector fields)

Graphics and Visualization

Open source packages developed by National Labs...

- VisIt
- ParaView

Harder to get going, but designed for large-scale 3d plots, distributed data, adaptive mesh refinement results, etc.:

Each have stand-alone GUI and also Python scripting capabilities.

Based on VTK (Visualization Tool Kit).

HPSC 101 — Lecture 6

This lecture:

- NumPy arrays and functions
- “Pythonic” ways to do things

- Python: main programs and private variables
- Timing Python execution

Lists aren't good as numerical arrays

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, e.g.

Multiplication repeats:

```
>>> x = [2., 3.]
>>> 2*x
[2.0, 3.0, 2.0, 3.0]
```

Addition concatenates:

```
>>> y = [5., 6.]
>>> x+y
[2.0, 3.0, 5.0, 6.0]
```

NumPy module

Instead, use NumPy arrays:

```
>>> import numpy as np

>>> x = np.array([2., 3.])
>>> 2*x
array([ 4.,  6.])

Try x*y where both x and y are
arrays of the same size.
```

Other operations also apply component-wise:

```
>>> np.sqrt(x) * np.cos(x) * x**3
array([ -4.708164  , -46.29736719])
```

Note: `*` is component-wise multiply

NumPy arrays

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

```
>>> np.array([1, 2, 3])     # all integers
array([1, 2, 3])

>>> np.array([1, 2, 3.])    # one float
array([ 1.,  2.,  3.])      # they're all floats!
```

Can explicitly state desired data type:

```
>>> x = np.array([1, 2, 3], dtype=complex)
>>> print x
[ 1.+0.j,  2.+0.j,  3.+0.j]

>>> (x + 1.j) * 2.j
array([-2.+2.j, -2.+4.j, -2.+6.j])
```

NumPy arrays for vectors and matrices

```
>>> A = np.array([[1.,2], [3,4], [5,6]])
>>> A
array([[ 1.,  2.],
       [ 3.,  4.],
       [ 5.,  6.]])

>>> A.shape
(3, 2)

>>> A.T
array([[ 1.,  3.,  5.],
       [ 2.,  4.,  6.]])

>>> x = np.array([1., 1.])
>>> x.T
array([ 1.,  1.])
```

NumPy arrays for vectors and matrices

```
>>> A
array([[ 1.,  2.],
       [ 3.,  4.],
       [ 5.,  6.]])

>>> x
array([ 1.,  1.])

>>> np.dot(A,x)      # matrix-vector product
array([  3.,   7.,  11.])

>>> np.dot(A.T, A)    # matrix-matrix product
array([[ 35.,  44.],
       [ 44.,  56.]])
```

NumPy matrices for vectors and matrices

For Linear algebra, may instead want to use numpy.matrix:

```
>>> A = np.matrix([[1.,2], [3,4], [5,6]])
>>> A
matrix([[ 1.,  2.],
        [ 3.,  4.],
        [ 5.,  6.]])
```

Or, Matlab style (as a string that is converted):

```
>>> A = np.matrix("1.,2; 3,4; 5,6")
>>> A
matrix([[ 1.,  2.],
        [ 3.,  4.],
        [ 5.,  6.]])
```

NumPy matrices for vectors and matrices

Note: vectors are handled as matrices with 1 row or column:

```
>>> x = np.matrix("4.;5.")
>>> x
matrix([[ 4.],
        [ 5.]])
>>> x.T
matrix([[ 4.,  5.]])
>>> A*x
matrix([[ 14.],
        [ 32.],
        [ 50.]])
```

But note that indexing into *x* requires two indices:

```
>>> print x[0,0], x[1,0]
4.0 5.0
Try x[:]
```

Which to use, array or matrix?

For linear algebra matrix may be easier (and more like Matlab), but vectors need two subscripts!

For most other uses, arrays more natural, e.g.

```
>>> x = np.linspace(0., 3., 100)   # 100 points
>>> y = x**5 - 2.*sqrt(x)*cos(x)   # 100 values
>>> plot(x,y)
```

np.linspace returns an array, which is what is needed here.

We will always use arrays.

See **http://www.scipy.org/NumPy_for_Matlab_Users**

Rank of an array

The rank of an array is the number of subscripts it takes:

```
>>> A = np.ones((4,4))

>>> A
array([[ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.]])

>>> np.rank(A)
2
```

Warning: This is not the rank of the matrix in the linear algebra sense (dimension of the column space)!

Rank of an array

Scalars have rank 0:

```
>>> z = np.array(7.)
>>> z
array(7.0)
```

NumPy arrays of any dimension are supported, e.g. rank 3:

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

       [[ 1.,  1.],
        [ 1.,  1.]]])
>>> T[0,0,0]
1.0
```

Linear algebra with NumPy

```
>>> A = np.array([[1., 2.], [3, 4]])
>>> A
array([[ 1.,  2.],
       [ 3.,  4.]])

>>> b = np.dot(A, np.array([8., 9.]))
>>> b
array([ 26.,  60.])
```

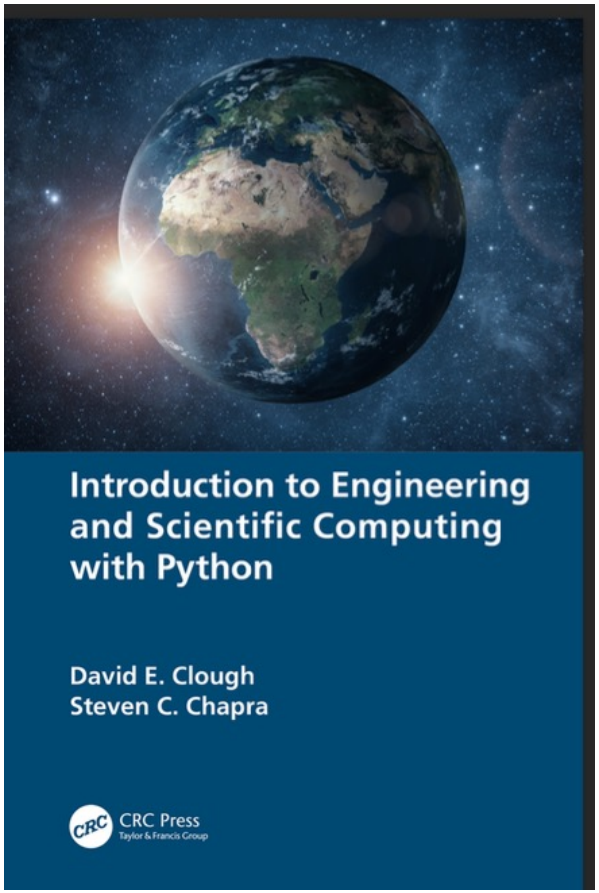
Now solve $Ax = b$:

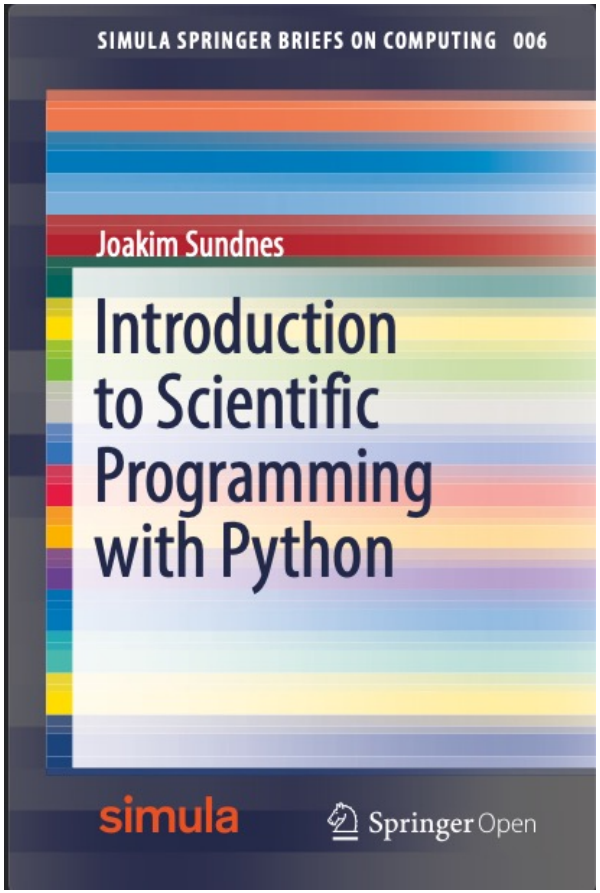
```
>>> from numpy.linalg import solve
>>> solve(A,b)
array([ 8.,  9.])
```

Eigenvalues

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

>>> eig(A)  # returns a tuple (evals,evecs)

(array([-0.37228132,  5.37228132]),
       array([[-0.82456484, -0.41597356],
              [ 0.56576746, -0.90937671]]))

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

>>> evals
array([-0.37228132,  5.37228132])

>>> evecs
array([[-0.82456484, -0.41597356],
       [ 0.56576746, -0.90937671]])
```

Reference





https://link.springer.com/book/10.1007/978-3-030-50356-7

Pythonic ways

- f-string format (since Python 3.6)

```
print("At iteration number %s, s= %20.15f" %(k,s))
print(f"At iteration number {k}, s= {s}")
print(f"At iteration number {k}, s= {s:10.15f}")

t = 1.234567
print(f"Default output gives t = {t}.")
print(f"We can set the precision: t = {t:.2}.")
print(f"Or control the number of decimals: t = {t:.2f}.")

Default output gives t = 1.234567.
We can set the precision: t = 1.2.
Or control the number of decimals: t = 1.23.
```

- Zip function

```
for low, high in zip(A_low, A_high):
    print(low, high)
```

Python ways - continued

- List slicing

```
>>> a = [2, 3.5, 8, 10]
>>> a[2:] # from index 2 to end of list [8, 10]
>>> a[1:3] # from index 1 up to, but not incl.,
index 3 [3.5, 8]
>>> a[:3] # from start up to, but not incl., index
3 [2, 3.5, 8]
>>> a[1:-1] # from index 1 to next last element
[3.5, 8]
>>> a[:] # the whole list [2, 3.5, 8, 10]

b = a[:] will make a copy of the entire list a, and
any subsequent changes to b will not change a
```

- Membership

```
List1=[1,2,3,”hello”]
2 in List1 – True
4 in List1 – False
”hello” in List1 - True
```

Quadrature (numerical integration)

Estimate $\int_0^2 x^2 \, \mathrm{d}x = \frac{8}{3}$:

```
>>> from scipy.integrate import quad

>>> def f(x):
...     return x**2
...
>>> quad(f, 0., 2.)
(2.6666666666666667, 2.9605947323337514e-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
>>> f(4)
16
```

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

```
>>> quad(lambda x: x**2, 0., 2.)
(2.666666666666667, 2.9605947323337514e-14)
```

“Main program” in a Python module

Python modules often end with a section that looks like:

```
if__name__== "__main__":

    # some code
```

This code is **not** executed if the file is imported as a module, only if it is run as a script, e.g. by...

```
$ python filename.py

>>> execfile("filename.py")

In[1]: run filename.py
```

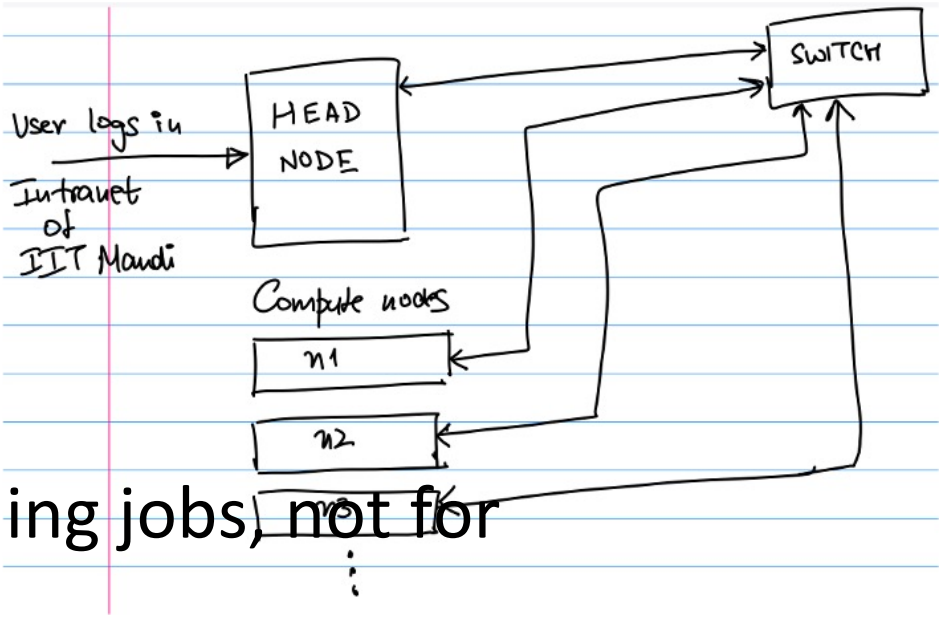
Exercise

- Write a function to calculate the exponential using the exponential series
  - The function should take the number of terms of the series as an argument, use default of 100
  - Check the convergence of the series
- Plot y=exp(x) for 1000 points between 0 and 100, using the built-in numpy exp() function and your_exp() function
- Compare the execution time of the two functions using the timeit command of Ipython
- All work should be done in Jupyter online and the notebook saved back into your local machine and pushed into your fork

This lecture:

- HPC introduction
- Python on HPC
- Hands-on
- Exercise

HPC - Introduction



HPC topology

Head node:
Suited to login and scheduling jobs, not for computation

Compute nodes:
Suited for heavy computation, typically has 2 processors per node (machine), each processor has up to 12 cores, 64+GB RAM, 2TB local storage

HPC IIT Mandi (website tour)

- https://sites.google.com/iitmandi.ac.in/hpc-iit-mandi/
- CPUHPC (10.8.1.19): CPU-based parallelism
- GPUHPC (10.8.1.20): GU-based parallelism
- 10G connectivity between nodes
- Filesystems
  - Home: 10GB
  - Working dir (wd): 2TB
- Software: basic software + *Singularity*
- Example PBS scripts
- Queue details
- Resources

Applications

Engineering

• Fluid flow and heat transfer (Open FOAM, Fluidity, ANSYS Fluent) • Solid mechanics (ANSYS)

• Deep learning (Python modules)

Physics

• Molecular dynamics (in-house codes)

Chemistry

• Computational chemistry (Gromacs)

Biology

• Gene sequencing

IIT Mandi HPC cluster: 2884 processing cores; 300+ users • 169 nodes; 2884 cores

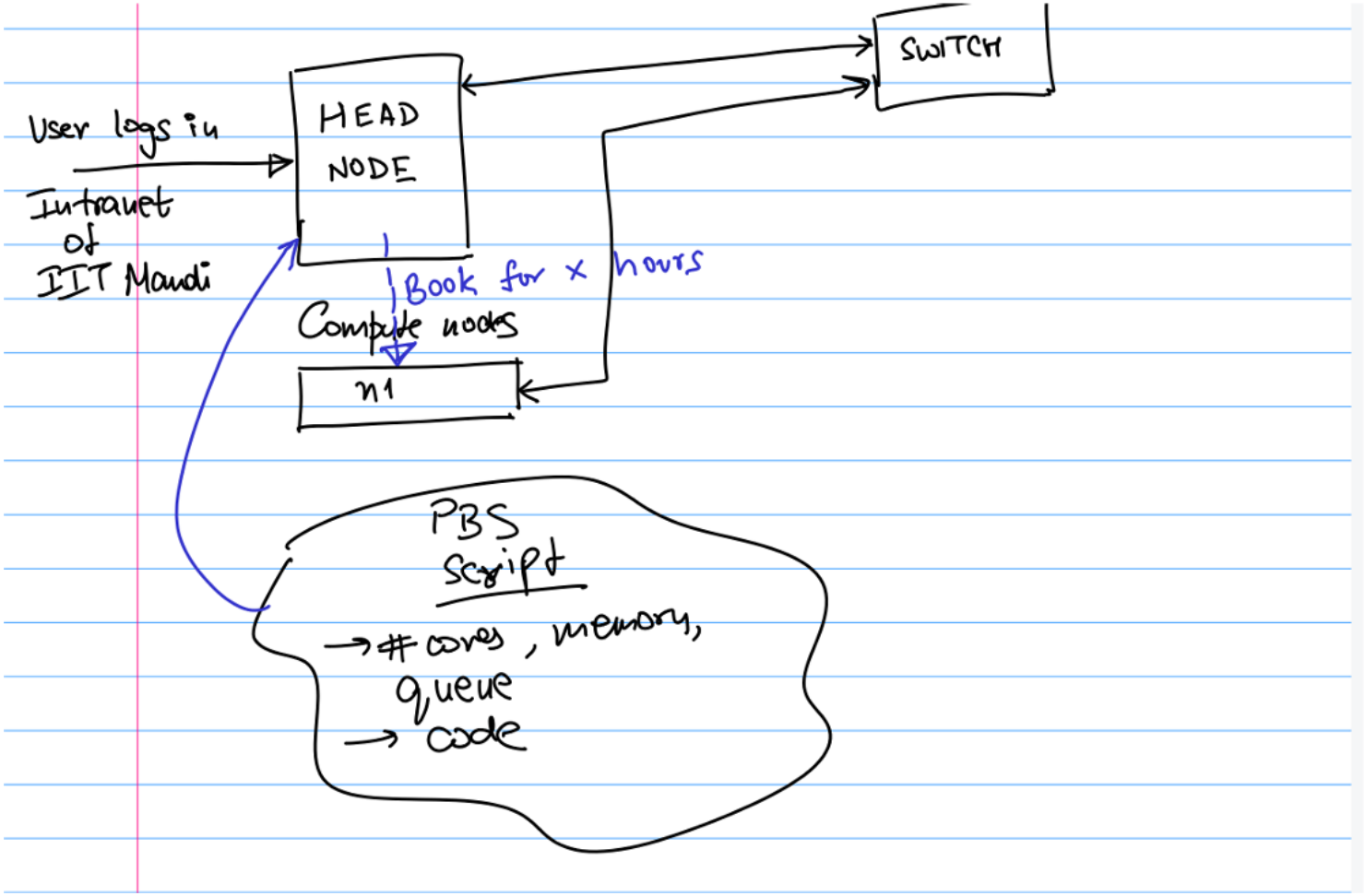
- Intel Xeon processors
- 11.5 TB memory
- 986 TB storage
- 33 Nvidia graphical processing units (GPUs) • 10Gb/s ethernet connectivity
- Cooling system

HPC – scheduler – Torque PBS

PBS: portable batch system

Queuing system for job submission which takes care of the number of jobs and excessive usage

Clusters
Filesystems


SWITCH
HEAD
NODE
User logs in
Intranet
of
IIT Mandi
Book for x hours
Compute nodes
n1
PBS
Script
→ # cores, memory,
queue
→ code

Hands-on

- Session screenshot to be recorded
- Logging in to the cluster(s) and directory structure
- Prepare virtualenv and install packages using pip
- PBS script for running a Python code
- Launching the PBS script using qsub
- Output and error files

Commands

scl —list - list all scl (Red Hat) packages available
scl enable rh-python36 bash (exit to exit from the scl)
pip3 install --user --upgrade --proxy=http://10.8.0.1:8080 pip
pip3 install --user --upgrade --proxy=http://10.8.0.1:8080 virtualenv
virtualenv ~/virtualenvs/testenv
source ./virtualenvs/testenv/bin/activate
pip3 install --upgrade --proxy=http://10.8.0.1:8080 pip
pip3 install --proxy=http://10.8.0.1:8080 numpy ipython jupyter
pip3 list - to check installed packages

qsub <pbs_script.sh>

PBS script – detailed discussion

- PBS directives used in the script
- man qsub
  - discuss environment variables
- https://docs.adaptivecomputing.com/torque/4-0-2/Content/topics/commands/qsub.htm

Qstat

qstat -an
-a - all jobs
-n - display nodes allocated to jobs
qstat -q
qstat <jobid> -f - full detail

qdel jobid
qdel all

Interactive PBS – Ipython

Qsub –I <pbs_script.sh>

Exercise 1

1. Login to the cluster
2. Copy the mysqrt.py file to wd
3. Copy a sample serial PBS script from the HPC website and edit it to calculate the square root of 2.0 with the debug mode on. The output should not be redirected into a file. See where it goes.
4. Launch the script on CPUHPC. Also, try with a different queue.
5. Launch the script on GPUHPC. Take care that the queues are different on the two clusters.
6. Also, try using the interactive mode in qsub
7. Try qstat options, and pbsnodes to check node mapping.
8. Copy the output file and error files back to your PC
9. Added challenge: can you write a bash conditional in your PBS script that runs the code only if the job is sent to n121. Use –'-l nodes=n121.cluster.iitmandi.ac.in' to test it.

Exercise 2 - advanced

- Copy the mysqrt.py module to the HPC home
- Write a python script called root.py that calculates the sqrt of number using mysqrt module and prints it. The number should be passed to the script through bash using sys.argv variable
- Write a bash loop that calls root.py for the first 1000 even numbers, i.e. 2,4,6,…
- The results should be appended into a file in the following format:
    - Sqrt of 2 is 1.414xxx, time taken xx seconds
    - Sqrt of 4 is 2.0, time taken xx seconds
- Run the bash loop over the HPC cluster in the serial queue

HPSC 101 — Lecture 7

This lecture:

- Python debugging demo
- Compiled langauges
- Introduction to Fortran 90 syntax
- Declaring variables, loops, booleans

Compiled vs. interpreted language

Not so much a feature of language syntax as of how language is converted into machine instructions.

Many languages use elements of both.

Interpreter:

- Takes commands one at a time, converts into machine code, and executes.
- Allows interactive programming at a shell prompt, as in Python or Matlab.
- Can't take advantage of optimising over a entire program — does not know what instructions are coming next.
- Must translate each command while running the code, possibly many times over in a loop.

Compiled language

The program must be written in 1 or more files (source code).

These files are input data for the compiler, which is a computer program that analyzes the source code and converts it into object code.

Compiled language

The program must be written in 1 or more files (source code).

These files are input data for the compiler, which is a computer program that analyzes the source code and converts it into object code.

The object code is then passed to a linker or loader that turns one or more objects into an executable.

Compiled language

The program must be written in 1 or more files (source code).

These files are input data for the compiler, which is a computer program that analyzes the source code and converts it into object code.

The object code is then passed to a linker or loader that turns one or more objects into an executable.

Why two steps?

Object code contains symbols such as variables that may be defined in other objects. Linker resolves the symbols and converts them into addresses in memory.

Compiled language

The program must be written in 1 or more files (source code).

These files are input data for the compiler, which is a computer program that analyzes the source code and converts it into object code.

The object code is then passed to a linker or loader that turns one or more objects into an executable.

Why two steps?

Object code contains symbols such as variables that may be defined in other objects. Linker resolves the symbols and converts them into addresses in memory.

Often large programs consist of many separate files and/or library routines — don't want to re-compile them all when only one is changed. (Later we'll use Makefiles.)

Fortran history

Prior to Fortran, programs were often written in machine code or assembly language.

FORTRAN = FORmula TRANslator

Fortran history

Prior to Fortran, programs were often written in machine code or assembly language.

FORTRAN = FORmula TRANslator

Fortran I: 1954–57, followed by Fortran II, III, IV, Fortran 66.

Major changes in Fortran 77, which is still widely used.

Fortran history

Prior to Fortran, programs were often written in machine code or assembly language.

FORTRAN = FORmula TRANslator

Fortran I: 1954–57, followed by Fortran II, III, IV, Fortran 66.

Major changes in Fortran 77, which is still widely used.

> *"I don't know what the language of the year 2000 will look like, but I know it will be called Fortran."*
>
> *– Tony Hoare, 1982*

Fortran history

Major changes again from Fortran 77 to Fortran 90.

Fortran 95: minor changes.

Fortran 2003, 2008: not fully implemented by most compilers.

We will use Fortran 90/95.

gfortran — GNU open source compiler

Several commercial compilers also available.

Fortran syntax

Big differences between Fortran 77 and Fortran 90/95.

Fortran 77 still widely used:

- Legacy codes (written long ago, millions of lines...)
- Faster for some things.

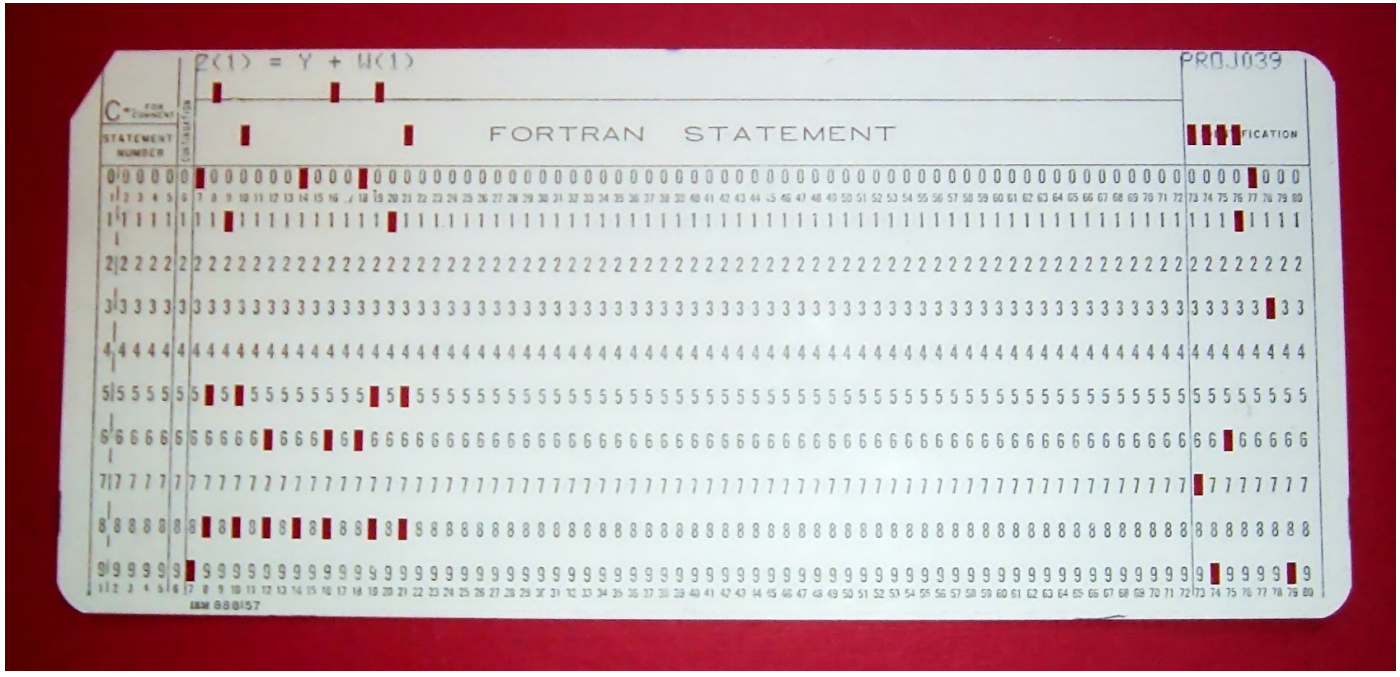
Note: In general adding more high-level programming features to a language makes it harder for compiler to optimize into fast-running code.

Fortran syntax

One big difference: Fortran 77 (and prior versions) required fixed format of lines:

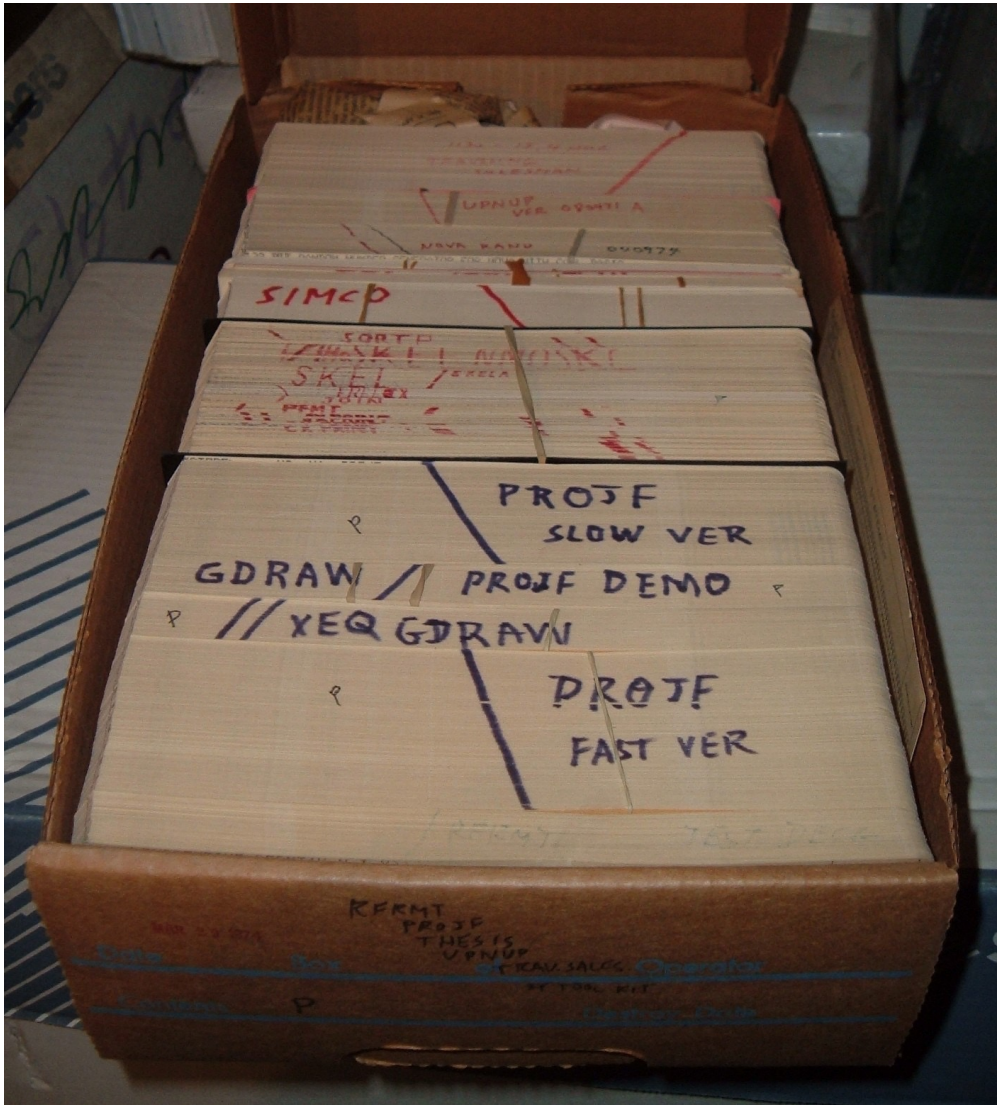
Executable statements must start in column 7 or greater,

Only the first 72 columns are used, the rest ignored!



http://en.wikipedia.org/wiki/File:FortranCardPROJ039.agr.jpg

Punch cards and decks



http://en.wikipedia.org/wiki/File:PunchCardDecks.agr.jpg

Paper tape

http://en.wikipedia.org/wiki/Punched_tape

Fortran syntax

Fortran 90: free format.

Indentation is optional (but highly recommended).

gfortran will compile Fortran 77 or 90/95.

Use file extension `.f` for fixed format (column 7 ...)

Use file extension `.f90` for free format.

Simple Fortran program

```
! example1.f90
program example1
    implicit none
    real (kind=8) :: x,y,z

    x = 3.d0
    y = 1.d-1
    z = x + y
    print *, "z = ", z
end program example1
```

Notes:

- Indentation optional (but make it readable!)
- First declaration of variables then executable statements
- implicit none means all variables must be declared

Simple Fortran program

```
! example1.f90
program example1
    implicit none
    real (kind=8) :: x,y,z

    x = 3.d0
    y = 1.d-1
    z = x + y
    print *, "z = ", z
end program example1
```

More notes:

- `(kind = 8)` means 8-bytes used for storage,
- `3.d0` means 3×10^0 in double precision (8 bytes)
- `2.d-1` means $2 \times 10^{-1} = 0.2$

Simple Fortran program

```
! example1.f90
  program example1
    implicit none
    real (kind=8) :: x,y,z

    x = 3.d0
    y = 2.d-1
    z = x + y
    print *, "z = ", z
end program example1
```

More notes:

- `print *, ...`: The `*` means no special format specified
  As a result all available digits of `z` will be printed.
- Later will see how to specify print format.

Compiling and running Fortran

Suppose example1.f90 contains this program.

Then:

```
$ gfortran example1.f90
```

compiles and links and creates an executable named `a.out`

To run the code after compiling it:

```
$ ./a.out
 z =     3.20000000000000
```

The command `./a.out` executes this file (in the current directory).

Compiling and running Fortran

Can give executable a different name with -o flag:

```
$ gfortran example1.f90 -o example1.exe
$ ./example1.exe
  z =    3.20000000000000
```

Can separate compile and link steps:

```
$ gfortran -c example1.f90 # creates example1.o

$ gfortran example1.o -o example1.exe
$ ./example1.exe
  z =    3.20000000000000
```

This creates and then uses the object code example1.o.

Compile-time errors

Introduce an error in the code: (`zz` instead of `z`)

```
program example1
    implicit none
    real (kind=8) :: x,y,z
    x = 3.d0
    y = 2.d-1
    zz = x + y
    print *, "z = ", z
end program example1
```

This gives an error when compiling:

```
 $ gfortran example1.f90
   example1.f90:11.6:
   zz = x + y
     1
Error: Symbol 'zz' at (1) has no IMPLICIT type
```

Without the “implicit none”

Introduce an error in the code: (`zz` instead of `z`)

```
program example1
    real (kind=8) :: x,y,z
    x = 3.d0
    y = 2.d-1
    zz = x + y
    print *, "z = ", z
end program example1
```

This compiles fine and gives the result:

```
$ gfortran example1.f90
$ ./a.out
   z =  -3.62666764177191E-038
```

Or some other random nonsense since `z` was never set.

Fortran types

Variables refer to particular storage location(s), must declare variable to be of a particular type and this won't change.

The statement

```
implicit none
```

means all variables must be explicitly declared.

Otherwise you can use a variable without prior declaration and the type will depend on what letter the name starts with.
Default:

- integer if starts with i, j, k, l, m, n
- real (kind=4) otherwise (single precision)

Many older Fortran codes use this convention!

Much safer to use `implicit none` for clarity,
and to help avoid typos.

Fortran arrays and loops

```
! loop1.f90
program loop1
   implicit none
   integer, parameter :: n = 10000
   real (kind=8), dimension(n) :: x, y
   integer :: i

   do i=1,n
      x(i) = 3.d0 * i
      enddo

   do i=1,n
      y(i) = 2.d0 * x(i)
      enddo

   print *, "Last y computed: ", y(n)
end program loop1
```

Fortran arrays and loops

```
program loop1
    implicit none
    integer, parameter :: n = 10000
    real (kind=8), dimension(n) :: x, y
    integer :: i
```

Comments:

- `integer, parameter` means this value will not be changed.
- `dimension(n) :: x, y` means these are arrays of length `n`.

Fortran arrays and loops

```
do i=1,n
   x(i) = 3.d0 * i
   enddo
```

Comments:

- `x(i)` means i'th element of array.
- Instead of `enddo`, can also use labels...

```
      do 100 i=1,n
         x(i) = 3.d0 * i
100      continue
```

The number 100 is arbitrary. Useful for long loops.
Often seen in older codes.

Fortran if-then-else

```
! ifelse1.f90

program ifelse1
    implicit none
    real(kind=8) :: x
    integer :: i

    i = 3

    if (i<=2) then
        print *, "i is less or equal to 2"
    else if (i/=5) then
        print *, "i is greater than 2, not equal to 5"
    else
        print *, "i is equal to 5"
    endif
end program ifelse1
```

Fortran if-then-else

Booleans: `.true.` `.false.`

Comparisons:

`<`	or `.lt.`	`<=`	or `.le.`
`>`	or `.gt.`	`>=`	or `.ge.`
`==`	or `.eq.`	`/=`	or `.ne.`

Examples:

```
if ((i >= 5) .and.  (i < 12)) then
```

```
if (((i .lt.  5) .or.  (i .ge.  12)) .and.  &
 (i .ne.  20)) then
```

Note: `&` is the Fortran continuation character.
Statement continues on next line.

Fortran if-then-else

```
! boolean1.f90
program boolean1
    implicit none
    integer :: i,k
    logical :: ever_zero

    ever_zero = .false.
    do i=1,10
        k = 3*i - 1
        ever_zero = (ever_zero .or. (k == 0))
        enddo

    if (ever_zero) then
        print *, "3*i - 1 takes the value 0 for some i"
    else
        print *, "3*i - 1 is never 0 for i tested"
    endif
end program boolean1
```

HPSC 101 — Lecture 8

This lecture:

- Fortran subroutines and functions
- Arrays
- Dynamic memory

Fortran functions and subroutines

For now, assume we have a single file `filename.f90` that contains the main program and also any functions or subroutines needed.

Later we will see how to split into separate files.

Will also discuss use of modules.

Fortran functions and subroutines

For now, assume we have a single file `filename.f90` that contains the main program and also any functions or subroutines needed.

Later we will see how to split into separate files.

Will also discuss use of modules.

Functions take some input arguments and return a single value.

Usage: `y = f(x)` or `z = g(x,y)`

Should be declared as external with the type of value returned:

```
real(kind=8), external ::  f
```

Fortran functions

```
1  ! $UWHPSC/codes/fortran/fcn1.f90
2
3  program fcn1
4      implicit none
5      real(kind=8) :: y,z
6      real(kind=8), external :: f
7
8      y = 2.
9      z = f(y)
10     print *, "z = ",z
11 end program fcn1
12
13 real(kind=8) function f(x)
14     implicit none
15     real(kind=8), intent(in) :: x
16     f = x**2
17 end function f
```

Prints out: `z =  4.000000000000000`

Fortran subroutines

Subroutines have arguments, each of which might be for input or output or both.

Usage: `call sub1(x,y,z,a,b)`

Can specify the intent of each argument, e.g.

```
real(kind=8), intent(in) :: x,y
real(kind=8), intent(out) :: z
real(kind=8), intent(inout) :: a,b
```

specifies that `x, y` are passed in and not modified,
`z` may not have a value coming in but will be set by `sub1`,
`a, b` are passed in and may be modified.

After this call, `z, a, b` may all have changed.

Fortran subroutines

```
1  ! $UWHPSC/codes/fortran/sub1.f90
2
3  program sub1
4      implicit none
5      real(kind=8) :: y,z
6
7      y = 2.
8      call fsub(y,z)
9      print *, "z = ",z
10 end program sub1
11
12 subroutine fsub(x,f)
13     implicit none
14     real(kind=8), intent(in) :: x
15     real(kind=8), intent(out) :: f
16     f = x**2
17 end subroutine fsub
```

Fortran subroutines

A version that takes an array as input and squares each value:

```
1  ! $UWHPSC/codes/fortran/sub2.f90
2
3  program sub2
4      implicit none
5      real(kind=8), dimension(3) :: y,z
6      integer n
7
8      y = (/2., 3., 4./)
9      n = size(y)
10     call fsub(y,n,z)
11     print *, "z = ",z
12 end program sub2
13
14 subroutine fsub(x,n,f)
15   ! compute f(x) = x**2 for all elements of the array x
16   ! of length n.
17   implicit none
18   integer, intent(in) :: n
19   real(kind=8), dimension(n), intent(in) :: x
20   real(kind=8), dimension(n), intent(out) :: f
21   f = x**2
22 end subroutine fsub
```

Array operations in Fortran

Fortran 90 supports some operations on arrays...

```
! $UWHPSC/codes/fortran/vectorops.f90
program vectorops
    implicit none
    real(kind=8), dimension(3) :: x, y

    x = (/10.,20.,30./)          ! initialize
    y = (/100.,400.,900./)

    print *, "x = "
    print *, x

    print *, "x**2 + y = "
    print *, x**2 + y            ! componentwise
```

Array operations in Fortran

```
! $UWHPSC/codes/fortran/vectorops.f90
! continued...

    print *, "x*y = "
    print *, x*y        ! = (x(1)y(1), x(2)y(2), ...)

    print *, "sqrt(y) = "
    print *, sqrt(y)                ! componentwise


    print *, "dot_product(x,y) = "
    print *, dot_product(x,y)       ! scalar product

end program vectorops
```

Array operations in Fortran — Matrices

```
! $UWHPSC/codes/fortran/arrayops.f90
program arrayops
    implicit none
    real(kind=8), dimension(3,2) :: a
    ...
    ! create a as 3x2 array:
    A = reshape((/1,2,3,4,5,6/), (/3,2/))
```

Note:

- Fortran is case insensitive: `A = a`
- Reshape fills array by columns, so

$$A = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}.$$

Array operations in Fortran — Matrices

```
! $UWHPSC/codes/fortran/arrayops.f90  (continued)
    real(kind=8), dimension(3,2) :: a
    real(kind=8), dimension(2,3) :: b
    real(kind=8), dimension(3,3) :: c
    integer :: i

    print *, "a = "
    do i=1,3
        print *, a(i,:)     ! i'th row
        enddo

    b = transpose(a)        ! 2x3 array

    c = matmul(a,b)         ! 3x3 matrix product
```

Array operations in Fortran — Matrices

```
! $UWHPSC/codes/fortran/arrayops.f90  (continued)
    real(kind=8), dimension(3,2) :: a
    real(kind=8), dimension(2) :: x
    real(kind=8), dimension(3) :: y

    x = (/5,6/)
    y = matmul(a,x)     ! matrix-vector product
    print *, "x = ",x
    print *, "y = ",y
```

Linear systems in Fortran

There is no equivalent of the Matlab backslash operator for solving a linear system $Ax = b$(`b = A\b`)

Must call a library subroutine to solve a system.

Later we will see how to use LAPACK routines

for this.

Note: Under the hood, Matlab calls LAPACK too!

So does NumPy.

Array storage

Rank 1 arrays have a single index, for example:

```
real(kind=8) :: x(3)
real(kind=8), dimension(3) :: x
```

are equivalent ways to define `x` with elements `x(1), x(2), x(3).`

You can also specify a different starting index:

```
real(kind=8) :: x(0:2), y(4:6), z(-2:0)
```

These are all arrays of length 3 and this would be a valid assignment:

```
y(5) = z(-2)
```

Multi-dimensional array storage

Memory can be thought of linear, indexed by a single address.

A one-dimensional array of length `N` will generally occupy `N` consecutive memory locations: `8N` bytes for floats.

A two-dimensional array (e.g. matrix) of size $m \times n$ will require mn memory locations.

Multi-dimensional array storage

Memory can be thought of linear, indexed by a single address.

A one-dimensional array of length `N` will generally occupy `N` consecutive memory locations: `8N` bytes for floats.

A two-dimensional array (e.g. matrix) of size $m \times n$ will require mn memory locations.

Might be stored by rows, e.g. first row, followed by second row, etc.

This what's done in Python or C, as suggested by notation:

```
A = [[10,20,30], [40,50,60]]
```

Multi-dimensional array storage

Memory can be thought of linear, indexed by a single address.

A one-dimensional array of length `N` will generally occupy `N` consecutive memory locations: `8N` bytes for floats.

A two-dimensional array (e.g. matrix) of size $m \times n$ will require mn memory locations.

Might be stored by rows, e.g. first row, followed by second row, etc.

This what's done in Python or C, as suggested by notation:

```
A = [[10,20,30], [40,50,60]]
```

Or, could be stored by columns, as done in Fortran!

Multi-dimensional array storage

$$Apy = \begin{bmatrix} 10 & 20 & 30 \\ 40 & 50 & 60 \end{bmatrix} \qquad Afort = \begin{bmatrix} 10 & 30 & 50 \\ 20 & 40 & 60 \end{bmatrix}$$

```
Apy = reshape(array([10,20,30,40,50,60]), (2,3))
Afort = reshape((/10,20,30,40,50,60/), (/2,3/))
```

Suppose the array storage starts at memory location 3401.

In Python or Fortran, the elements will be stored in the order:

```
loc 3401    Apy[0,0] = 10      Afort(1,1) = 10
loc 3402    Apy[0,1] = 20      Afort(2,1) = 20
loc 3403    Apy[0,2] = 30      Afort(1,2) = 30
loc 3404    Apy[1,0] = 40      Afort(2,2) = 40
loc 3405    Apy[1,1] = 50      Afort(1,3) = 50
loc 3406    Apy[1,2] = 60      Afort(2,3) = 60
```

Aside on np.reshape

The np.reshape method can go through data in either order:

```
>>> v = linspace(10,60,6)

>>> v
array([ 10.,  20.,  30.,  40.,  50.,  60.])

>>> reshape(v, (2,3))    # order='C' by default
array([[ 10.,  20.,  30.],
       [ 40.,  50.,  60.]])

>>> reshape(v, (2,3), order='F')
array([[ 10.,  30.,  50.],
       [ 20.,  40.,  60.]])
```

Aside on np.reshape

The np.reshape method can go through data in either order:

```
>>> A
array([[ 10.,  20.,  30.],
       [ 40.,  50.,  60.]])

>>> A.reshape(3,2)    # order='C' by default
array([[ 10.,  20.],
       [ 30.,  40.],
       [ 50.,  60.]])

>>> A.reshape((3,2),order='F')
array([[ 10.,  50.],
       [ 40.,  30.],
       [ 20.,  60.]])
```

Note: reshape can be called as function or method of A...

```
>>> reshape(A, (3,2), order='F')
```

Aside on np.flatten

The np.flatten method converts an N-dim array to a 1-dimensional one:

```
>>> A = np.array([[10.,20,30],[40,50,60]])
>>> A
array([[ 10.,  20.,  30.],
       [ 40.,  50.,  60.]])


>>> A.flatten()     # Default is 'C'
array([ 10.,  20.,  30.,  40.,  50.,  60.])


>>> A.flatten('F') # Fortran ordering
array([ 10.,  40.,  20.,  50.,  30.,  60.])
```

Memory management for arrays

Often a program needs to be written to handle arrays whose size is not known until the program is running.

Fortran 77 approaches:

- Allocate arrays large enough for any application,
- Use “work arrays” that are partitioned into pieces.

We will look at some examples from LAPACK since you will probably see this in other software!

Memory management for arrays

Often a program needs to be written to handle arrays whose size is not known until the program is running.

Fortran 77 approaches:

- Allocate arrays large enough for any application,
- Use “work arrays” that are partitioned into pieces.

We will look at some examples from LAPACK since you will probably see this in other software!

The good news:

Fortran 90 allows dynamic memory allocation.

Memory allocation

```
real(kind=8) dimension(:), allocatable :: x
real(kind=8) dimension(:,:), allocatable :: a

allocate(x(10))
allocate(a(30,10))

! use arrays...

! then clean up:
deallocate(x)
deallocate(a)
```

Memory allocation

If you might run out of memory, use optional argument stat to return the status...

```
real(kind=8), dimension(:,:), allocatable :: a

allocate(a(30000,10000), stat=alloc_error)

if (alloc_error /= 0) then
    print *, "Insufficient memory"
    stop
    endif
```

Passing arrays to subroutines — code with bug

```
1  ! $CLASSHG/codes/fortran/arraypassing1.f90
2
3  program arraypassing1
4
5      implicit none
6      real(kind=8) :: x,y
7      integer :: i,j
8
9      x = 1.
10     y = 2.
11     i = 3
12     j = 4
13     call setvals(x)
14     print *, "x = ",x
15     print *, "y = ",y
16     print *, "i = ",i
17     print *, "j = ",j
18
19 end program arraypassing1
20
21 subroutine setvals(a)
22     ! subroutine that sets values in an array a of length 3.
23     implicit none
24     real(kind=8), intent(inout) :: a(3)
25     integer i
26     do i = 1,3
27         a(i) = 5.
28         enddo
29 end subroutine setvals
```

Note: `x` is a scalar, setvals dummy argument `a` is an array.

Passing arrays to subroutines

The `call setvals(x)` statement passes the address where `x` is stored.

In the subroutine, the array `a` of length 3 is assumed to start at this address. So next `3*8 = 24` bytes are assumed to be elements of `a(1:3)`.

In fact these 24 bytes are occupied by

x. (8 bytes),
y. (8 bytes),
i. (4 bytes),
j. (4 bytes).

So setting `a(1:3)` changes all these variables!

Passing arrays to subroutines

This produces:

```
x =     5.00000000000000
y =     5.00000000000000
i =    1075052544
j =             0
```

Nasty!!

- The storage location of `x` and the next 2 storage locations were all set to the floating point value 5.0e0
- This messed up the values originally stored in `y, i, j`.
- Integers are stored differently than floats. Two integers take up 8 bytes, the same as one float, so the assignment `a(3) = 5.` overwrites both `i` and `j`.
- The first half of the float 5., when interpreted as an integer, is huge.

Passing arrays to subroutines — another bug

```
1  ! $CLASSHG/codes/fortran/arraypassing2.f90
2
3  program arraypassing2
4
5      implicit none
6      real(kind=8) :: x,y
7      integer :: i,j
8
9      x = 1.
10     y = 2.
11     i = 3
12     j = 4
13     call setvals(x)
14     print *, "x = ",x
15     print *, "y = ",y
16     print *, "i = ",i
17     print *, "j = ",j
18
19 end program arraypassing2
20
21 subroutine setvals(a)
22     ! subroutine that sets values in an array a of length 1000.
23     implicit none
24     real(kind=8), intent(inout) :: a(1000)
25     integer i
26     do i = 1,1000
27         a(i) = 5.
28         enddo
29 end subroutine setvals
```

Note: We now try to set 1000 elements in memory!

Passing arrays to subroutines

This compiles fine, but running it gives:

```
Segmentation fault
```

This means that the program tried to change a value of memory it was not allowed to.

Only a small amount of memory is devoted to the variables declared.

The memory we tried to access might be where the program itself is stored, or something related to another program that's running.

Segmentation faults

Debugging segmentation faults can be difficult.

Tip: Compile using `-fbounds-check` flag of gfortran.

This catches some cases when you try to access an array out of bounds.

But not the case just shown! The variable was passed to a subroutine that doesn't know how long the array should be.

For a case where this helps, see
`$UWHPSC/codes/fortran/segfault1.f90`

HPSC 101 — Lecture 9

This lecture:

- Multi-file Fortran codes
- Makefiles

Reading:

Software Carpentry lectures on Make

Dependency checking

Makefiles give a way to recompile only the parts of the code that have changed.

Also used for checking dependencies in other build systems,

e.g. creating figures, running latex, bibtex, etc. to construct a manuscript.

Dependency checking

Makefiles give a way to recompile only the parts of the code that have changed.

Also used for checking dependencies in other build systems,

e.g. creating figures, running latex, bibtex, etc. to construct a manuscript.

More modern build systems are available, e.g. SCons, which allows expressing dependencies and build commands in Python.

But make (or gmake) are still widely used.

Fortran code with 3 units

```
1  ! $UWHPSC/codes/fortran/multifile1/fullcode.f90
2
3  program demo
4      print *, "In main program"
5      call sub1()
6      call sub2()
7  end program demo
8
9  subroutine sub1()
10     print *, "In sub1"
11 end subroutine sub1
12
13 subroutine sub2()
14     print *, "In sub2"
15 end subroutine sub2
```

Split code into 3 separate files...

```fortran
! $UWHPSC/codes/fortran/multifile1/main.f90

program demo
    print *, "In main program"
    call sub1()
    call sub2()
end program demo
```

```fortran
! $UWHPSC/codes/fortran/multifile1/sub1.f90

subroutine sub1()
    print *, "In sub1"
end subroutine sub1
```

```fortran
! $UWHPSC/codes/fortran/multifile1/sub2.f90

subroutine sub2()
    print *, "In sub2"
end subroutine sub2
```

Splitting Fortran codes into files

Compile all three and link together into single executable:

```
$ gfortran main.f90 sub1.f90 sub2.f90 \
    -o main.exe
```

Run the executable:

```
$ ./main.exe
 In main program
 In sub1
 In sub2
```

Splitting Fortran codes into files

Can split into separate compile....

```
$ gfortran -c main.f90 sub1.f90 sub2.f90

$ ls *.o
main.o  sub1.o  sub2.o
```

... and link steps:

```
$ gfortran main.o sub1.o sub2.o -o main.exe

$ ./main.exe > output.txt
```

Note: Redirected output to a text file.

Splitting Fortran codes into files

Advantage: If we modify sub2.f90 to print "Now in sub2" we only need to recompile this piece:

```
$ gfortran -c sub2.f90

$ gfortran main.o sub1.o sub2.o -o main.exe

$ ./main.exe
 In main program
 In sub1
 Now in sub2
```

When working on a big code (e.g. 100,000 lines split between 200 subroutines) this can make a big difference!

Splitting Fortran codes into files

Advantage: If we modify sub2.f90 to print "Now in sub2" we only need to recompile this piece:

```
$ gfortran -c sub2.f90

$ gfortran main.o sub1.o sub2.o -o main.exe

$ ./main.exe
 In main program
 In sub1
 Now in sub2
```

When working on a big code (e.g. 100,000 lines split between 200 subroutines) this can make a big difference!

Use of Makefiles greatly simplifies this.

Makefiles

A common way of automating software builds and other complex tasks with dependencies.

A Makefile is itself a program in a special language.

```
1  # $UWHPSC/codes/fortran/multifile1/Makefile
2
3  output.txt: main.exe
4          ./main.exe > output.txt
5
6  main.exe: main.o sub1.o sub2.o
7          gfortran main.o sub1.o sub2.o -o main.exe
8
9  main.o: main.f90
10         gfortran -c main.f90
11 sub1.o: sub1.f90
12         gfortran -c sub1.f90
13 sub2.o: sub2.f90
14         gfortran -c sub2.f90
```

Makefiles

```
$ cd $UWHPSC/codes/fortran/multifile1
$ rm -f *.o *.exe    # remove old versions

$ make main.exe
gfortran -c main.f90
gfortran -c sub1.f90
gfortran -c sub2.f90
gfortran main.o sub1.o sub2.o -o main.exe
```

Uses commands for making `main.exe`.

note: First had to make all the `.o` files.
Then executed the rule to make `main.exe`

Structure of a Makefile

Typical element in the simple Makefile:

```
target:  dependencies
<TAB>  command(s) to make target
```

Important to use tab character, not spaces!!
Warning: Some editors replace tabs with spaces!

Typing “make target” means:

1. Make sure all the dependencies are up to date (those that are also targets)
2. If target is older than any dependency, recreate it using the specified commands.

Structure of a Makefile

Typical element in the simple Makefile:

```
target:  dependencies
<TAB>  command(s) to make target
```

Important to use tab character, not spaces!!
Warning: Some editors replace tabs with spaces!

Typing “make target” means:

1. Make sure all the dependencies are up to date (those that are also targets)
2. If target is older than any dependency, recreate it using the specified commands.

These rules are applied recursively!

Make examples

```
$ rm -f *.o *.exe

$ make sub1.o
gfortran -c sub1.f90

$ make main.o
gfortran -c main.f90

$ make main.exe
gfortran -c sub2.f90
gfortran main.o sub1.o sub2.o -o main.exe
```

Note: Last make required compiling `sub2.f90`
but not `sub1.f90` or `main.f90`.

Age of dependencies

The last modification time of the file is used.

```
$ ls -l sub1.*
-rw-r--r--  1 rjl  staff  111 Apr 18 16:05 sub1.f90
-rw-r--r--  1 rjl  staff  936 Apr 18 16:56 sub1.o

$ make sub1.o
make: `sub1.o' is up to date.

$ touch sub1.f90;   ls -l sub1.f90
-rw-r--r--  1 rjl  staff  111 Apr 18 17:10 sub1.f90

$ make main.exe
gfortran -c sub1.f90
gfortran main.o sub1.o sub2.o -o main.exe
```

Makefiles

First version of Makefile has 3 rules that are very similar

```
1  # $UWHPSC/codes/fortran/multifile1/Makefile
2
3  output.txt: main.exe
4          ./main.exe > output.txt
5
6  main.exe: main.o sub1.o sub2.o
7          gfortran main.o sub1.o sub2.o -o main.exe
8
9  main.o: main.f90
10         gfortran -c main.f90
11 sub1.o: sub1.f90
12         gfortran -c sub1.f90
13 sub2.o: sub2.f90
14         gfortran -c sub2.f90
```

Replace these with a pattern rule...

Implicit rules

General rule to make the `.o` file from `.f90` file:

```
1  # $UWHPSC/codes/fortran/multifile1/Makefile2
2
3  output.txt: main.exe
4          ./main.exe > output.txt
5
6  main.exe: main.o sub1.o sub2.o
7          gfortran main.o sub1.o sub2.o -o main.exe
8
9  %.o : %.f90
10         gfortran -c $<
```

Making `main.exe` requires `main.o sub1.o sub2.o` to be up to date.

Rather than a rule to make each one separately, the implicit rule (lines 9-10) is used for all three.

Specifying a different makefile

To use a makefile with a different name than `Makefile`:

```
$ make sub1.o -f Makefile2
gfortran -c sub1.f90
```

The rules in `Makefile2` will be used.

The directory `$UWHPSC/codes/fortran/multifile1` contains several sample makefiles.

See class notes: Makefiles for a summary.

Implicit rules

We have to repeat the list of `.o` files twice:

```
1  # $UWHPSC/codes/fortran/multifile1/Makefile2
2
3  output.txt: main.exe
4          ./main.exe > output.txt
5
6  main.exe: main.o sub1.o sub2.o
7          gfortran main.o sub1.o sub2.o -o main.exe
8
9  %.o : %.f90
10         gfortran -c $<
```

Simplify and reduce errors by defining a macro.

Makefile variables or macros

```
1  # $UWHPSC/codes/fortran/multifile1/Makefile3
2
3  OBJECTS = main.o sub1.o sub2.o
4
5  output.txt: main.exe
6          ./main.exe > output.txt
7
8  main.exe: $(OBJECTS)
9          gfortran $(OBJECTS) -o main.exe
10
11 %.o : %.f90
12         gfortran -c $<
```

By convention, all-caps names are used for Makefile macros.

Note that to use `OBJECTS` we must write `$(OBJECTS)`.

Makefile variables

```
1  # $UWHPSC/codes/fortran/multifile1/Makefile4
2
3  FC = gfortran
4  FFLAGS = -O3
5  LFLAGS =
6  OBJECTS = main.o sub1.o sub2.o
7
8  output.txt: main.exe
9          ./main.exe > output.txt
10
11 main.exe: $(OBJECTS)
12         $(FC) $(LFLAGS) $(OBJECTS) -o main.exe
13
14 %.o : %.f90
15         $(FC) $(FFLAGS) -c $<
```

Here we have added for the name of the Fortran command and for compile flags and linking flags.

Makefile variables

```
$ rm -f *.o *.exe
$ make -f Makefile4

gfortran  -O3 -c main.f90
gfortran  -O3 -c sub1.f90
gfortran  -O3 -c sub2.f90
gfortran  -O3 main.o sub1.o sub2.o -o main.exe
./main.exe > output.txt
```

Can specify variables on command line:

```
$ rm -f *.o *.exe
$ make main.exe FFLAGS=-g  -f Makefile4

gfortran  -g -c main.f90
gfortran  -g -c sub1.f90
gfortran  -g -c sub2.f90
gfortran  -g main.o sub1.o sub2.o -o main.exe
```

Phony targets — don't create files

```
1  # $UWHPSC/codes/fortran/multifile1/Makefile5
2
3  OBJECTS = main.o sub1.o sub2.o
4  .PHONY: clean
5
6  output.txt: main.exe
7          ./main.exe > output.txt
8
9  main.exe: $(OBJECTS)
10         gfortran $(OBJECTS) -o main.exe
11
12 %.o : %.f90
13         gfortran -c $<
14
15 clean:
16         rm -f $(OBJECTS) main.exe
```

Note: No dependencies, so always do commands

```
$ make clean -f Makefile5
rm -f main.o sub1.o sub2.o main.exe
```

Common Makefile error

Using spaces instead of tab...

If we did this in the `clean` commands, we'd get:

```
$ make clean -f Makefile5
```

```
Makefile5:14: *** missing separator.  Stop.
```

make help

```
1  # $UWHPSC/codes/fortran/multifile1/Makefile6
2
3  OBJECTS = main.o sub1.o sub2.o
4  .PHONY: clean help
5
6  output.txt: main.exe
7          ./main.exe > output.txt
8
9  main.exe: $(OBJECTS)
10         gfortran $(OBJECTS) -o main.exe
11
12 %.o : %.f90
13         gfortran -c $<
14
15 clean:
16         rm -f $(OBJECTS) main.exe
17
18 help:
19         @echo "Valid targets:"
20         @echo "  main.exe"
21         @echo "  main.o"
22         @echo "  sub1.o"
23         @echo "  sub2.o"
24         @echo "  clean:  removes .o and .exe files"
```

`echo` means print out the string.

`@echo` means print out the string but don't print the command.

Fancier things are possible...

```
1  # $UWHPSC/codes/fortran/multifile1/Makefile7
2
3  SOURCES = $(wildcard *.f90)
4  OBJECTS = $(subst .f90,.o,$(SOURCES))
5
6  .PHONY: test
7
8  test:
9          @echo "Sources are: " $(SOURCES)
10         @echo "Objects are: " $(OBJECTS)
```

This gives:

```
$ make test -f Makefile6
Sources are:  fullcode.f90 main.f90 sub1.f90 sub2.f
Objects are:  fullcode.o main.o sub1.o sub2.o
```

Note this found `fullcode.f90` too!

HPSC 101 — Lecture 10

Outline:

- Fortran modules
- Newton's method example

Fortran modules

General structure of a module:

```
module <MODULE-NAME>
    ! Declare variables
contains
    ! Define subroutines or functions
end module <MODULE-NAME>
```

A program/subroutine/function can use this module:

```
program <NAME>
    use <MODULE-NAME>
    ! Declare variables
    ! Executable statements
end program <NAME>
```

Fortran modules

Can also specify a list of what variables/subroutines/functions from module to be used.

Similar to `from numpy import linspace`
rather than `from numpy import *`

```
program <NAME>
    use <MODULE-NAME>, only: <LIST OF SYMBOLS>
    ! Declare variables
    ! Executable statements
end program <NAME>
```

Makes it easier to see which variables come from each module.

Fortran module example

```
1  ! $UWHPSC/codes/fortran/multifile2/sub1m.f90
2
3  module sub1m
4
5  contains
6
7  subroutine sub1()
8      print *, "In sub1"
9  end subroutine sub1
10
11 end module sub1m
```

```
1  ! $UWHPSC/codes/fortran/multifile2/main.f90
2
3  program demo
4      use sub1m, only: sub1
5      print *, "In main program"
6      call sub1()
7  end program demo
```

Fortran modules

Some uses:

- Can define global variables in modules to be used in several different routines.

  In Fortran 77 this had to be done with common blocks — much less elegant.

- Subroutine/function interface information is generated to aid in checking that proper arguments are passed.

  It's often best to put all subroutines and functions in modules for this reason.

- Can define new data types to be used in several routines. ("derived types" rather than "intrinsic types")

Compiling Fortran modules

If `sub1m.f90` is a module, then compiling it creates `sub1m.o` and also `sub1m.mod`:

```
$ gfortran -c sub1m.f90


$ ls
main.f90     sub1m.f90    sub1m.mod    sub1m.o
```

the module must be compiled before any subroutine or program that uses it!

```
$ rm -f sub1m.mod
$ gfortran main.f90 sub1m.f90
main.f90:5.13:

    use sub1m
            1
Fatal Error: Can't open module file 'sub1m.mod'
for reading at (1): No such file or directory
```

Another module example

```
1  ! $UWHPSC/codes/fortran/circles/circle_mod.f90
2
3  module circle_mod
4
5      implicit none
6      real(kind=8), parameter :: pi = 3.1415926535897793d0
7
8  contains
9
10     real(kind=8) function area(r)
11         real(kind=8), intent(in) :: r
12         area = pi * r**2
13     end function area
14
15     real(kind=8) function circumference(r)
16         real(kind=8), intent(in) :: r
17         circumference = 2.d0 * pi * r
18     end function circumference
19
20 end module circle_mod
```

Another module example

```fortran
! $UWHPSC/codes/fortran/circles/main.f90

program main

    use circle_mod, only: pi, area
    implicit none
    real(kind=8) :: a

    ! print parameter pi defined in module:
    print *, 'pi = ', pi

    ! test the area function from module:
    a = area(2.d0)
    print *, 'area for a circle of radius 2: ', a

end program main
```

Running this gives:

```
pi =     3.14159265358979
area for a circle of radius 2:     12.5663706143
```

Module variables

```
1  ! $UWHPSC/codes/fortran/circles/circle_mod.f90
2  ! Version where pi is a module variable.
3
4  module circle_mod
5
6      implicit none
7      real(kind=8) :: pi
8      save
9
10 contains
11
12     real(kind=8) function area(r)
13         real(kind=8), intent(in) :: r
14         area = pi * r**2
15     end function area
16
17     real(kind=8) function circumference(r)
18         real(kind=8), intent(in) :: r
19         circumference = 2.d0 * pi * r
20     end function circumference
21
22 end module circle_mod
```

Module variables

```
1  ! $UWHPSC/codes/fortran/circles/main.f90
2
3  program main
4
5      use circle_mod, only: pi, area
6      implicit none
7      real(kind=8) :: a
8
9      call initialize()   ! sets pi
10
11     ! print module variable pi:
12     print *, 'pi = ', pi
13
14     ! test the area function from module:
15     a = area(2.d0)
16     print *, 'area for a circle of radius 2: ', a
17
18 end program main
```

Module variables

The module variable `pi` should be initialized in a program unit that is called only once.

It can be initialized to full machine precision using

$$\pi = \arccos(-1)$$

```
1  ! $UWHPSC/codes/fortran/circles/initialize.f90
2
3  subroutine initialize()
4
5      ! Set the value of pi used elsewhere.
6      use circle_mod, only: pi
7      pi = acos(-1.d0)
8
9  end subroutine initialize
```

Makefile

```
1  # $UWHPSC/codes/fortran/circles2/Makefile
2
3  OBJECTS = circle_mod.o \
4            main.o \
5            initialize.o
6
7  MODULES = circle_mod.mod
8
9  .PHONY: clean
10
11 output.txt: main.exe
12         ./main.exe > output.txt
13
14 main.exe: $(MODULES) $(OBJECTS)
15         gfortran $(OBJECTS) -o main.exe
16
17 %.o: %.f90
18         gfortran -c $<
19
20 %.mod: %.f90
21         gfortran -c $<
22
23 clean:
24         rm -f $(OBJECTS) $(MODULES) main.exe
```

Fortran subroutines

A version that takes an array as input and squares each value:

```
1  ! $UWHPSC/codes/fortran/sub2.f90
2
3  program sub2
4      implicit none
5      real(kind=8), dimension(3) :: y,z
6      integer n
7
8      y = (/2., 3., 4./)
9      n = size(y)
10     call fsub(y,n,z)
11     print *, "z = ",z
12 end program sub2
13
14 subroutine fsub(x,n,f)
15   ! compute f(x) = x**2 for all elements of the array x
16   ! of length n.
17   implicit none
18   integer, intent(in) :: n
19   real(kind=8), dimension(n), intent(in) :: x
20   real(kind=8), dimension(n), intent(out) :: f
21   f = x**2
22 end subroutine fsub
```

Module version — creates an interface

Now do not need to pass the value `n` into the subroutine.

```
1  ! $UWHPSC/codes/fortran/sub3.f90
2
3  module sub3module
4
5  contains
6
7  subroutine fsub(x,f)
8    ! compute f(x) = x**2 for all elements of the array x.
9    implicit none
10   real(kind=8), dimension(:), intent(in) :: x
11   real(kind=8), dimension(size(x)), intent(out) :: f
12   f = x**2
13 end subroutine fsub
14
15 end module sub3module
16
17 !-----------------------------------------------
18
19 program sub3
20     use sub3module
21     implicit none
22     real(kind=8), dimension(3) :: y,z
23
24     y = (/2., 3., 4./)
25     call fsub(y,z)
26     print *, "z = ",z
27 end program sub3
```

Module for Newton's method

See the class notes: Fortran example for Newton's method