BRIEF INTRODUCTION TO TOOLS FOR SCIENTIFIC COMPUTING

OVERVIEW

- Scope of these two lectures
- Resources used in computing: CPU, GPU, memory, disk, network
- Operating systems: personal computers, super computers
- Computing languages, interpreted and compiled, programs split over several files
- Tools for programming: editor, command line, Integrated Development Environment (IDE)
- Debugging
- Revision control for code development: git

SCOPE

- To introduce some of the **tools** needed to **efficiently** take your computing from short programmes in individual files (“Hello World”) to **larger scale projects**
- Cover considerations necessary to obtain **fast execution**
- Considerations when you need more resources (**large scale computing**)
- Not a guide for a specific computing language – most large scale projects use several languages, both **compiled** and **interpreted,** to get each task done most efficiently

RESOURCES

- Multiple resources are used during execution of your programme
  - **CPU** (Central Processing Unit) – interpretation of all code, number crunching
  - Possibly a Graphics Processing Unit (**GPU**) – extremely fast for specific tasks, e.g. running Machine Learning algorithms TensorFlow
  - Random Access Memory (**RAM**) – the working memory has to hold both the programs and the data
  - **Disk** – long-term storage of data and the program
  - **Network** – on larger computers, the disk access might be over the network
- The access to each of these resources can be the bottleneck for a fast execution. Identify the most important.

OPERATING SYSTEM

- Modern examples: PCs: Windoze, iOS, Linux; Supercomputers: Linux
- At the most basic level, the operating system **allows the CPU to interact** with the other components of the computer: memory, disk, network card and gpu (and any other peripherals).
- Controls user access to files, processes, memory….
- Operating systems also come with a **user interface –** you can write your programme in windows and compile it to run on Linux. You can also choose to work with an appealing interface and execute your programs on an efficient platform.
- **Compiled programs** work for the platform they are compiled for, **interpreted languages** work where the right interpreter is installed

PROGRAMMING

- The CPU needs instructions in order to perform operations, calculations etc.
- Operations are performed on “registers” in the CPU, and content needs to be copied from memory before they can be operated on.
- The CPU understands only instructions from a low level “instruction set”:

asm	machine code	Description
add	0x03 *ModR/M*	Add one 32-bit register to another.
mov	0x8B *ModR/M*	Move one 32-bit register to another.
mov	0xB8 *DWORD*	Move a 32-bit constant into register eax.
ret	0xc3	Returns from current function.
xor	0x33 *ModR/M*	XOR one 32-bit register with another.
xor	0x34 *BYTE*	XOR register al with this 8-bit constant.

```
0:    b8 05 00 00 00          mov    eax,0x5
5:    b9 05 00 00 00          mov    ecx,0x5
a:    ba 05 00 00 00          mov    edx,0x5
```

COMPILED OR INTERPRETED

- Such low-level programming can create very efficient code, but is barely human readable, difficult to debug, and the (more advanced) instruction set CPU specific. Impractical.
- We often use high level languages, e.g. C++, Python, Perl, scripts etc. But the instructions from all of these languages are translated into machine code for the CPU to understand.
- This translation to machine code can either happen every time the programme is executed for run-time **interpreted** languages (like Python, Perl), or the programme is **compiled** (translated) once (C++, Fortran,…). The compilation offers much control and can lead to very efficient programs, but adds an extra layer of complication.

TYPE CHECKING LANGUAGES

- Some languages (C++, Fortran) check variable types to e.g. ensure functions are called with the right type of arguments:
- double power (double x, int n) $x\hat{}n$
  can by calculated simpler (=faster) with (n-1) multiplications than
  double power (double x, double y) $x\hat{}y$
- Additions and multiplications are fast, exponentials, logarithms, trigonometric functions evaluate 500 times slower
- Python etc. mostly does not check types – interprets how to execute commands for each type of argument

LARGER SOURCE CODES

- When projects get large, it starts being beneficial to split the code into several files:
  - Easier navigation, filenames can indicate the function
  - Code reuse
- C++: `#include` header file with definitions, link object files after compilation

  Python: `import <filename>` or

  `from <filename> import <function>`

  where `<filename>` excludes the `.py` ending

DEBUGGING

- Consider the hypothetical situation that a program provides a different result to the what you had intended. That it is “buggy” and needs “debugging”.
- **Do not underestimate the power of print()**: insert print statements to follow the evolution of variables, branches etc. A quick method for simple issues.
- A **Debugger** lets you step through each line of code and check the state of each variable, insert break points (“run until you reach this point”) and inspect more complicated data structures (e.g. arrays). gdb, pdb, ddd,…
- **Valgrind** is another useful tool for memory and speed inspections

THE ENVIRONMENT FOR DEVELOPING

- A good environment for developing code includes:

  Editor (or Integrated Development Environment) [Emacs]

  Interpreter or compiler [gcc, Python]

  Debugger [gdb, pdb, ddd]

  Revision control software [git, see next lecture]

THE COMMAND LINE

If you need to run your program on a bigger computer, you need to copy your source to that computer and compile it there.

You will often used `ssh` (secure shell) to connect and then be presented with a command line. Use `tmux` for persistence between logons. Learn to navigate the command line.

You can use the command line also on your own computer: Open a terminal (Linux, iOS) and with wsl in linux (https://fedoramagazine.org/wsl-fedora-33/).

EFFICIENCY CONSIDERATIONS

Additions, multiplications are cheap, trigonometric, logarithms, exponentials expensive. Use memory to save on evaluations:

```
if (abs(sin(x))<0.5) return -sin(x);

return sin(x);
```

Better:

```
temp=sin(x);

if (abs(temp)<0.5) temp=-temp;
return temp;
```

EFFICIENCY CONSIDERATIONS

Overhead in function calls. Example: Calculate n!=n*[(n-1)!].

Can use recursive definition:

```
int fac(int n) { // assume n>=0 for correct result
if (n<=1) return 1;
else return n*fac(n-1);}
```

Much faster:

```
int fac(int n) {
  int m=1;
  for (int i=1;i<=n;i++)
    m=m*i;
  return m;}
```

GETTING READY FOR CODING

- Install a good editor
- Install your language interpreter/compiler of choice
  C++: gcc.gnu.org, Python: www.python.org
- Make a simple program, ranging from “Hello World” to compare the timing for the calculation of n! using an recursive and an iterative method
- Install method for accessing the computer you need to run the project on