1 What is computational science?

- It's also known as numerical Analysis and Scientific Computing.
- Contains elements of mathematics and computer science.
- Main applications are in science, engineering and data science
- It has very old origins, but the rapid development of computers makes it an ever changing field. Hence why its called "computational science", since it predates computers
- **Definition:** The process of producing and analyzing approximate, numerical solutions to mathematical or scientific problems.

A famous example of computational science is PageRank, the system Google uses for page rankings. It can be represented as a linear system, to solve $Lw = w$ where the entry L_ij is related to the number of links from node j to node i

Computational Science as changed physics/engineering significantly.
Prior to the 1970s, the two pillars were *theory* and *experimentation*.
The main downfall of these methods were the expense and time. As such, numerical approximation (computational science) was

- more feasible
- easier to analyse
- easier to manipulate
- less expensive

An example of where this is used extensively now is in turbulent flow simulations, such as for when engineering planes, or high performance cars, where airflow is extremely important.

It is also used extensively in Machine Learning

More definition time:
Symbolic Computation: application of algebra and math identities to manipulate an expression involving symbols and variables. (basically just doing math by hand)
Example:

$$2\exp(-x^2 + a) = 1$$
$$ln(2) - x^2 + a = 0$$
$$x = \pm\sqrt{ln(2) + a}$$

Numerical Computation: Performing arithmetic on (approximate) numerical quantities. The result is a (set of) number(s) of finite precision.
Example:

$$2\exp(-x^2 + 2) = x$$

$x_0 = 1.\underline{5}$, $x_1 = 1.\underline{510}15398$, $x_2 = 1.\underline{510254}584...$ where the underlined section of each answer is the known precision, and everything after that is subject to change after the next iteration.

MAPLE and **MATHEMATICA** are symbolic computation programs / Computer Algebra Systems (CAS), but can do some numerical computation too.

Python and **MATLAB** are best suited for numerical computation, but can do some symbolic computation too.

C, C++ and **FORTRAN** are commonly used for numerical computation, and are generally faster than Python or Matlab.

Symbolic Computations are **exact**.

Numerical Computations are **approximate**.

1.1 non-linear equations

In computational science, non-linear equations/functions are solved using iterative methods

Algorithms solving $f(x^*) = 0$ are usually iterative

Starting from $x^{(0)}$, make sequence of iterates

Rule $x^{(k+1)} = f(x^{(k)})$ for $k \geq 0$ is recurrence relation.

Some popular methods for solving/approximating non-linear equations are:

- Bisection (linear convergence rate)
- Newton-Raphson (quadratic convergence* except when root is local min/max)
- Euler-Chebyschev (even faster convergence)
- Secant method (quadratic convergence* based on Newton-Raphson, but doesn't use tangent)

The equations needed for finding approximations of x:
Bisection:

$$f(x) = 0$$

Newton-Raphson:

$$x^{(k+1)} = x^{(k)} - \frac{f(x^{(k)})}{f'(x^{(k)})}$$

The secant method is a "second order recurrence" relation. which means it relates the next approximation to the two previous ones

Bisection only works for 1 unknown

Newton-Raphson: Function must be differentiable/continuous around x*.

For 2d information on Newton-Raphon, check Lecture 4 and Lecture 5 slides.

1.2 Systems of linear equations (matrices)

Uses Gaussian Elimination and LU decomposition

Never solve linear systems by computing A^{-1}

Solving for the inverse explicitly is slow and generally leads to large numerical error

Lecture 6 for LU decomposition information

"When does my computation work?": $A = LU$ decomposition works if and only if all leading principal submatrices of A (i.e. A(1: k, 1: k) for k $\leq$ n) are nonsingular (invertible) The $PA = LU$ decomposition works even if A is nonsingular (not invertible).

Step 1: solve Ly = Pb using forward substitution

Step 2: Solve Ux = y using backward substitution

Lecture 7 for norm information

1.3 NORMS AND RELATIVE ERROR

error formula:

$$\frac{\|x - x_*\|}{\|x\|}$$

Where x is the true solution/exact answer to the system, and x_* is the approximate solution

if $||x - x_*||$ is small, then x_* is a good approximation of x. To compute this, you must choose a norm, typically the 2 norm

Error Vector:

$$e := x - x_*$$

Error:

$$||e||$$

Residual Vector:

$$r := b - Ax_*$$

Residual:

$$||r||$$

If $x = x_*$, then $||e|| = ||r|| = 0$

Relative Maximal Error:

$$\frac{||e||}{||x||} \leq K(A)\frac{||r||}{||b||}$$

The RHS of the above equation is how to calculate the Relative Maximal Error. $K(A)$ is the condition number, and $\frac{||r||}{||b||}$ is the relative residual of x_*

so

$$\frac{||x - x_*||}{||x||} \leq K(A)\frac{||r||}{||b||}$$

and the LHS of the equation is the true relative error. Which is typically not computable since x is generally unknown

Calculate Condition number:

`numpy.linalg.cond(A)`

Calculate Norm

```
scipy.linalg.norm(x, 2) # 2-norm / Euclidean norm
scipy.linalg.norm(x, 1) # 1-norm / Manhattan norm
scipy.linalg.norm(x, scipy.inf) # max/infinity/chebyschev norm
scipy.linalg.norm(x, p) # p-norm
```

NOTE: Norm-wise errors can hide component-wise errors in vectors

As a rule of thumb, if $K(A) \approx 10^q$, you can compute q digits less for x_* than you know for b.

If $K(A) \approx 2.6$, then the relative error when solving $Ax = b$ is at most 2.6 times larger than the relative residual.

1.4 FLOP counting and SUMMATION FORMULAS

FLOP = Floating-point operations

The following all count as floating-point operations:

$$\begin{array}{cc} + & - \\ * & / \\ \% & \end{array}$$

Iterating from 0 to 10 does not count as floating-point operations though for some reason.

Counting flops steps:

- On each line, count the number of flops
- Count the number of times each line is executed (for loop)
- track the # of flops as summations
- use summation identities to derive a formula for the number of flops for a given n

Summation Identities:

$$\sum_{k=1}^{n} 1 = n$$

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$$

$$\sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}$$

$$\sum_{k=1}^{n} k^3 = \left[\frac{n(n+1)}{2}\right]^2$$

Summation limits can be transformed if necessary:

$$\sum_{k=\alpha}^{\beta} a_k = \sum_{l=1}^{\beta-\alpha+1}$$

e.g.

$$\sum_{k=3}^{27} 5 = 5 \times \sum_{l=1}^{27-3+1} 1 = 5 \times \sum_{l=1}^{25} 1 = 5 \times 25 = 125$$

Common Functions and their corresponding flops

function	flops
sum of n terms	$n-1$
product of n factors	$n-1$
dot product of n-vectors	$2n-1$
$n \times n$ matrix-vector product	$2n^2-n$
$n \times n$ matrix-matrix product	$2n^3-n^2$

1.5 Big $\mathcal{O}$

To get the Big $\mathcal{O}$, take the highest power of n in the flops, and remove the coefficient.

So, $3n^2+n-3$ would have a Big $\mathcal{O}$ of n^2, or $\mathcal{O}(n^2)$

1.6 2d, non-linear systems of equations (LECTURE 11)

We cannot use LU-decomposition for systems of non-linear equations. As such, we must use an iterative method like Newton-Raphson.

Newton-Raphson iteration can be generalized to n equations with n unknowns. We also generally need the same number of equations and unknowns to find (isolated) solutions.

$$f_1(x_1+\delta x_1, x_2+\delta x_2) \approx f_1(x_1,x_2) + \frac{\partial f_1}{\partial x_1}(x_1,x_2)\delta x_1 + \frac{\partial f_1}{\partial x_1}(x_1,x_2)\delta x_2 = 0$$

$$f_2(x_1+\delta x_1, x_2+\delta x_2) \approx f_2(x_1,x_2) + \frac{\partial f_2}{\partial x_1}(x_1,x_2)\delta x_1 + \frac{\partial f_2}{\partial x_1}(x_1,x_2)\delta x_2 = 0$$

Or can be represented as: (in matrix form)

$$\begin{pmatrix} \frac{\partial f_1}{\partial x_1}(x_1,x_2) & \frac{\partial f_1}{\partial x_2}(x_1,x_2) \\ \frac{\partial f_2}{\partial x_1}(x_1,x_2) & \frac{\partial f_2}{\partial x_2}(x_1,x_2) \end{pmatrix} \begin{pmatrix} \delta x_1 \\ \delta x_2 \end{pmatrix} = - \begin{pmatrix} f_1(x_1,x_2) \\ f_2(x_1,x_2) \end{pmatrix}$$

In the matrix form/representation, the first matrix (the big, 2x2 one) is called the Jacobian matrix, and can be represented as J.

The 2nd matrix/vector is called the "update vector" and can be denoted as δx
The RHS is denoted b

Or in more understandable terms, the Jacobian matrix for a 2 equation system would be:

$$\begin{pmatrix} (f_1:) \text{ derivative with respect to } x_1 & \text{derivative with respect to } x_2 \\ (f_2:) \text{ derivative with respect to } x_1 & \text{derivative with respect to } x_2 \end{pmatrix}$$

Terminology:

- $r^{(k)} := f(x^{(k)}) =$ residual
- $\delta x^{(k)} := -[f'(x^{(k)})]^{-1} r^{(k)} =$ update