Computational Methods and Modelling 3

Antonio Attili & Edward McCarthy, (Course Organiser)

antonio.attili@ed.ac.uk
ed.mccarthy@ed.ac.uk

School of Engineering
University of Edinburgh
United Kingdom

Lecture 1
Introduction to the course
Definition of Errors

Computational Methods and Modelling 3

- Semester 1, 2022-23
- MECE09033
- 10 credits
- 11 lectures
- 10 tutorials/workshops

Antonio Attili
Lecturer in Computational Reactive Flows
antonio.attili@ed.ac.uk

Edward McCarthy, (Course Organizer)
Senior Lecturer in Composites Design and Testing
ed.mccarthy@ed.ac.uk

Aims of the Course

- Consolidate the ability to write computer programmes
- Equip engineers to formulate a given design challenge as a numerical problem
- Allow engineers to solve complex problems using efficient numerical techniques
- Provide understanding of classic numerical techniques to select the appropriate ones for the situation

1. **Introduction to modelling and approximation**
   - Approximate calculations
   - Error analysis and control

2. **Numerical methods**
   - Finding roots of equations
   - Solving implicit equations
   - Simultaneous equations and matrix operations
   - Solution of ordinary differential equations
   - Interpolation
   - Integration

3. **Optimisation methods**
   - One-dimensional optimization (golden section search, direct and gradient methods)
   - Multi-dimensional and constrained optimization techniques

Exams and Assignments

Two assessment components:

1. **Open Book Exam (50%)**
   - Date to be announced

2. **Project (50%)**
   - Engineering problem to be modelled and solved with numerical methods implemented in Python
   - Deadline: Submit Python code and report by Tuesday 21 November, 4 PM

Course activities and resources

- **Lectures**
  - Tuesday 10:00AM - 11.00AM.
  - Slides, tutorials and resources available on Learn Ultra.

Course activities and resources

- **Lectures**
  - Tuesday 10:00AM - 11.00AM.
  - Slides, tutorials and resources available on Learn Ultra.

- **Tutorial/workshop/surgery**
  - Wednesday mornings from 11:00AM to 2:00PM; in-person in Alrick TLC
  - Three slots, one hour each; 1/3 of class in each slot
  - Seating arrangement will be organized to reflect project groups

Course activities and resources

- **Lectures**
  - Tuesday 10:00AM - 11.00AM.
  - Slides, tutorials and resources available on Learn Ultra.

- **Tutorial/workshop/surgery**
  - Wednesday mornings from 11:00AM to 2:00PM; in-person in Alrick TLC
  - Three slots, one hour each; 1/3 of class in each slot
  - Seating arrangement will be organized to reflect project groups
  - Tutorial questions are released on Learn at least 24 hours before the tutorial session
  - Opportunity to discuss tutorial(s)/theory/project with lecturers/demonstrators/peers
  - Explanation of tutorial solutions by lecturers/demonstrators towards the end of the session.

Course activities and resources

- **Lectures**
  - Tuesday 10:00AM - 11.00AM.
  - Slides, tutorials and resources available on Learn Ultra.

- **Tutorial/workshop/surgery**
  - Wednesday mornings from 11:00AM to 2:00PM; in-person in Alrick TLC
  - Three slots, one hour each; 1/3 of class in each slot
  - Seating arrangement will be organized to reflect project groups
  - Tutorial questions are released on Learn at least 24 hours before the tutorial session
  - Opportunity to discuss tutorial(s)/theory/project with lecturers/demonstrators/peers
  - Explanation of tutorial solutions by lecturers/demonstrators towards the end of the session.

- **Discussion board on Learn**
  - Opportunity to post questions and engage with lecturers/demonstrators
  - Opportunity to interact with other students

Course activities and resources

- **Lectures**
  - Tuesday 10:00AM - 11.00AM.
  - Slides, tutorials and resources available on Learn Ultra.

- **Tutorial/workshop/surgery**
  - Wednesday mornings from 11:00AM to 2:00PM; in-person in Alrick TLC
  - Three slots, one hour each; 1/3 of class in each slot
  - Seating arrangement will be organized to reflect project groups
  - Tutorial questions are released on Learn at least 24 hours before the tutorial session
  - Opportunity to discuss tutorial(s)/theory/project with lecturers/demonstrators/peers
  - Explanation of tutorial solutions by lecturers/demonstrators towards the end of the session.

- **Discussion board on Learn**
  - Opportunity to post questions and engage with lecturers/demonstrators
  - Opportunity to interact with other students

- **Surgery hour**
  - Thursday 11am - 12 noon, one-to-one meeting with lecturer
    In-person, Eddie's office: Room 1.077, Sanderson Building, Kings Buildings (first floor at rear of building, use rear corner door for quickest access).
  - Opportunity to discuss specific issues that could not be addressed during Wednesday morning sessions

Group project

Goals of the project

- Design and implement a simulation code to solve an engineering problem
- Use the developed code to perform simulations to address typical design tasks

Information

- Project brief will be released on Learn in Week 3
- Group activity, 5 or 6 students per group
- Group submission, one report and one code per group
- Groups for project consistent with the three slots for Wednesday sessions

Textbook and online resources

- The University has subscribed to a digital edition of the following text:
  Steven Chapra and Raymond Canale
  Numerical Methods for Engineers 8^{th} Edition

Getting Python and Spyder

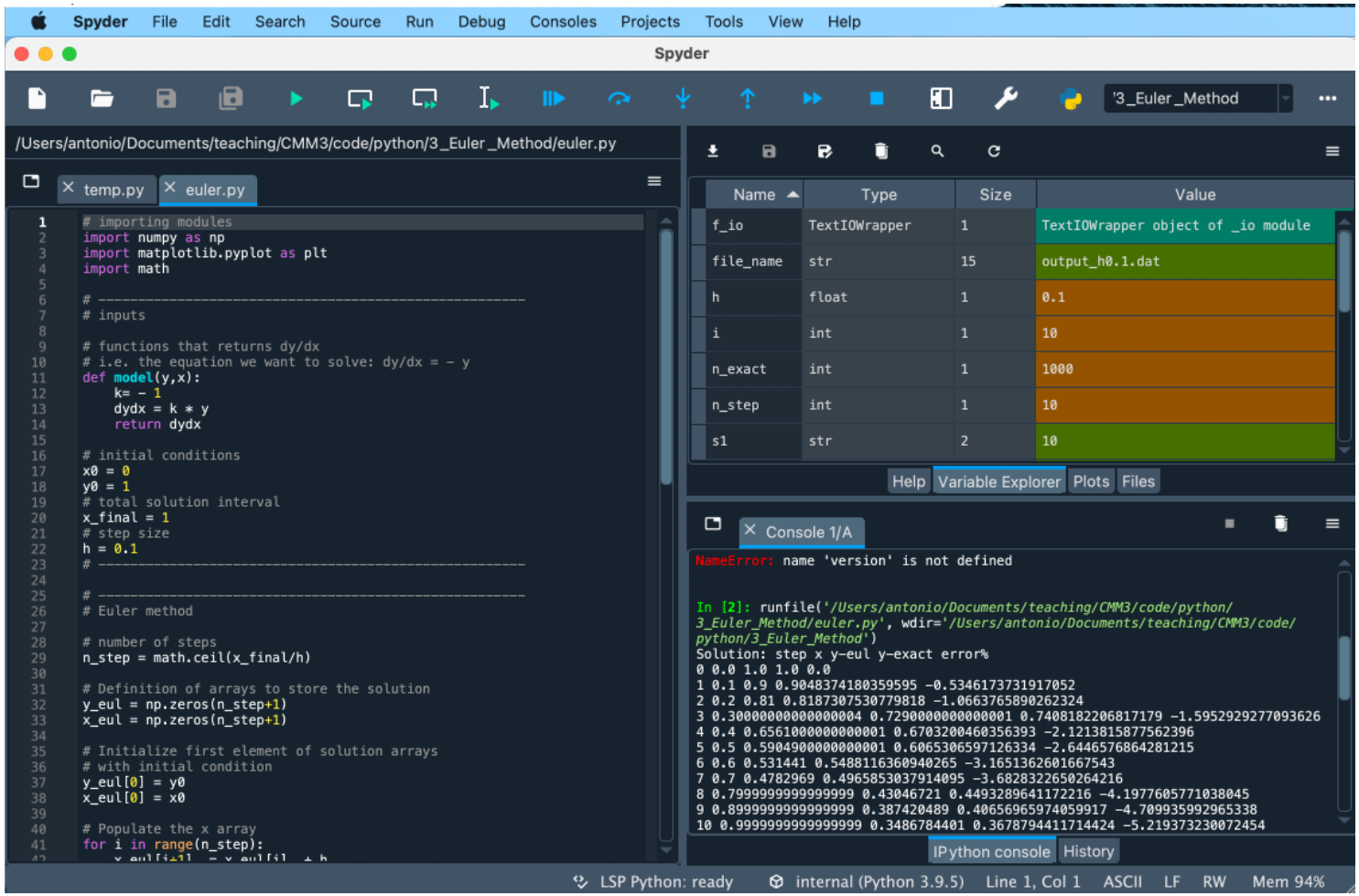
- We will use Python 3.9 and the Spyder platform. However, you are free to use any platform or editor with Python as long as it is Python 3.9 or newer.

Getting Python and Spyder

- We will use Python 3.9 and the Spyder platform. However, you are free to use any platform or editor with Python as long as it is Python 3.9 or newer.
- Go to webpage: https://www.spyder-ide.org/
- Download and install Spyder. Standalone installers on Windows and macOS.
  For Linux, cross-platform Anaconda distribution is recommended.

Getting Python and Spyder

- We will use Python 3.9 and the Spyder platform. However, you are free to use any platform or editor with Python as long as it is Python 3.9 or newer.
- Go to webpage: https://www.spyder-ide.org/
- Download and install Spyder. Standalone installers on Windows and macOS. For Linux, cross-platform Anaconda distribution is recommended.



Why computer programming?

Consider this large matrix:

$$\begin{pmatrix} -\frac{200x^2\left(\sqrt{\phi}-1\right)}{(\phi)^{\frac{3}{2}}}+\frac{200x^2}{\phi}+\frac{200\left(\sqrt{\phi}-1\right)}{\sqrt{\phi}}+\frac{2000\pi\omega y}{x^3\left(\frac{\phi}{x^2}\right)}+\frac{5000\pi^2y^2}{x^4\left(\frac{\phi}{x^2}\right)^2}-\frac{2000\pi\omega y^3}{x^5\left(\frac{\phi}{x^2}\right)^2} & -\frac{200xy\left(\sqrt{\phi}-1\right)}{(\phi)^{\frac{3}{2}}}+\frac{200xy}{\phi}-\frac{1000\pi\omega}{x^2\left(\frac{\phi}{x^2}\right)}-\frac{5000\pi^2y}{x^3\left(\frac{\phi}{x^2}\right)^2}+\frac{2000\pi\omega y^2}{x^4\left(\frac{\phi}{x^2}\right)^2} & \frac{1000\pi y}{x^2\left(\frac{\phi}{x^2}\right)} \\ -\frac{200xy\left(\sqrt{\phi}-1\right)}{(\phi)^{\frac{3}{2}}}+\frac{200xy}{\phi}-\frac{1000\pi\omega}{x^2\left(\frac{\phi}{x^2}\right)}-\frac{5000\pi^2y}{x^3\left(\frac{\phi}{x^2}\right)^2}+\frac{2000\pi\omega y^2}{x^4\left(\frac{\phi}{x^2}\right)^2} & -\frac{200y^2\left(\sqrt{\phi}-1\right)}{(\phi)^{\frac{3}{2}}}+\frac{200y^2}{\phi}+\frac{200\left(\sqrt{\phi}-1\right)}{\sqrt{\phi}}+\frac{5000\pi^2}{x^2\left(\frac{\phi}{x^2}\right)^2}-\frac{2000\pi\omega y}{x^3\left(\frac{\phi}{x^2}\right)^2} & -\frac{1000\pi}{x\left(\frac{\phi}{x^2}\right)} \\ \frac{1000\pi y}{x^2\left(\frac{\phi}{x^2}\right)} & -\frac{1000\pi}{x\left(\frac{\phi}{x^2}\right)} & 202 \end{pmatrix}$$

- Imagine trying to compute this matrix by hand for different values of x, y, ϕ
- Achievable in reasonable time? **No!**
- Good use of high level, creative engineers? **No!**

- Solution: implement a (flexible and general!) computer code

Motivation of numerical methods and simulations

- Numerical method: formulate a mathematical problem as a sequence of arithmetic operations
- One common feature of numerical methods: large numbers of tedious arithmetic calculations

Motivation of numerical methods and simulations

- Numerical method: formulate a mathematical problem as a sequence of arithmetic operations
- One common feature of numerical methods: large numbers of tedious arithmetic calculations
- Precomputer era: Analytical methods, graphical approaches, ...
- Issues: Limited class of problem, linearization, dimentionality reduction, geometry simplification

Motivation of numerical methods and simulations

- Numerical method: formulate a mathematical problem as a sequence of arithmetic operations
- One common feature of numerical methods: large numbers of tedious arithmetic calculations
- Precomputer era: Analytical methods, graphical approaches, ...
- Issues: Limited class of problem, linearization, dimentionality reduction, geometry simplification
- Numerical methods implemented on computers provided an efficient alternative

Motivation of numerical methods and simulations

- Numerical method: formulate a mathematical problem as a sequence of arithmetic operations
- One common feature of numerical methods: large numbers of tedious arithmetic calculations
- Precomputer era: Analytical methods, graphical approaches, ...
- Issues: Limited class of problem, linearization, dimentionality reduction, geometry simplification
- Numerical methods implemented on computers provided an efficient alternative
- **Shift effort from solution to formulation and interpretation**

Motivation of numerical methods and simulations

- Numerical method: formulate a mathematical problem as a sequence of arithmetic operations
- One common feature of numerical methods: large numbers of tedious arithmetic calculations
- Precomputer era: Analytical methods, graphical approaches, ...
- Issues: Limited class of problem, linearization, dimentionality reduction, geometry simplification
- Numerical methods implemented on computers provided an efficient alternative
- **Shift effort from solution to formulation and interpretation**

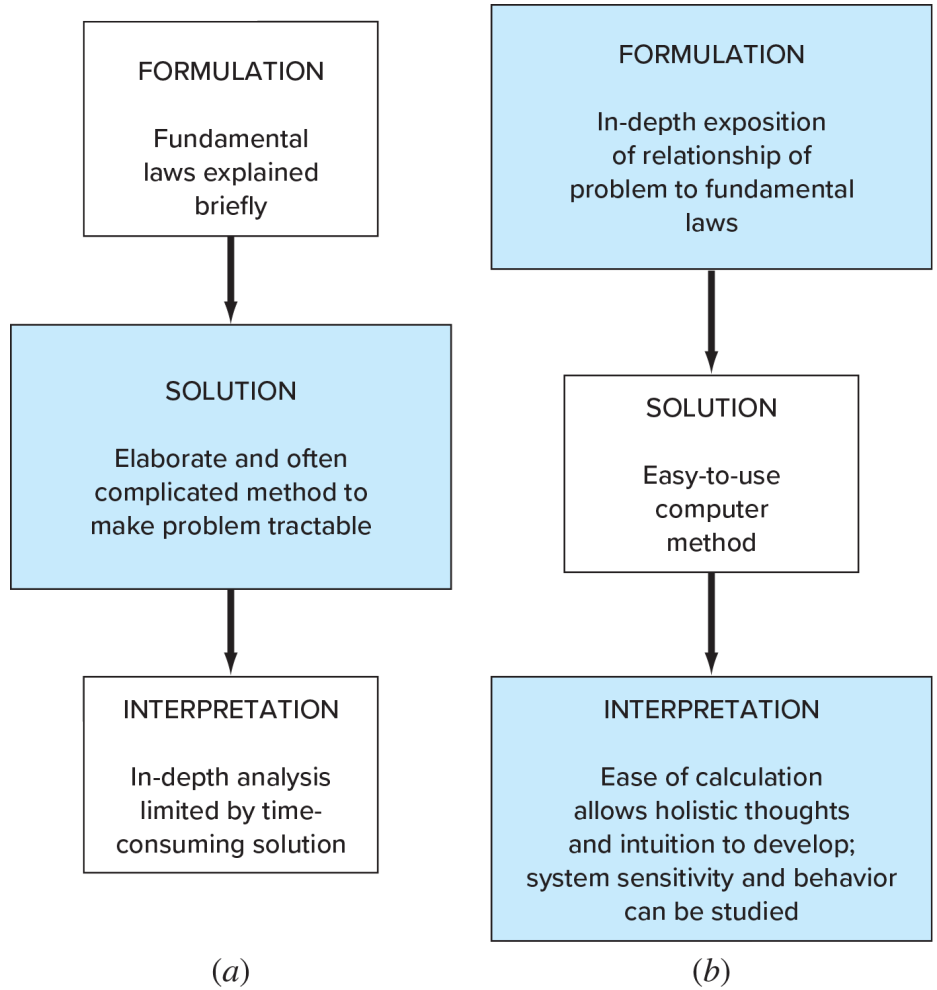


Figure: The three phases of engineering problem solving in (a) the precomputer and (b) the computer era.

Typical practical issues requiring numerical methods

1. **Nonlinear versus linear.** Linearization often needed for analytical solution, but not appropriate in many cases (e.g., fluid dynamics)

Typical practical issues requiring numerical methods

1. **Nonlinear versus linear.** Linearization often needed for analytical solution, but not appropriate in many cases (e.g., fluid dynamics)
2. **Large versus small systems.** Often not feasible to analyse large systems without a computer (rule of thumb: over three interacting components)

Typical practical issues requiring numerical methods

1. **Nonlinear versus linear.** Linearization often needed for analytical solution, but not appropriate in many cases (e.g., fluid dynamics)
2. **Large versus small systems.** Often not feasible to analyse large systems without a computer (rule of thumb: over three interacting components)
3. **Nonideal versus ideal.** Idealized laws are common in engineering, but often they are not accurate enough

Typical practical issues requiring numerical methods

1. **Nonlinear versus linear.** Linearization often needed for analytical solution, but not appropriate in many cases (e.g., fluid dynamics)
2. **Large versus small systems.** Often not feasible to analyse large systems without a computer (rule of thumb: over three interacting components)
3. **Nonideal versus ideal.** Idealized laws are common in engineering, but often they are not accurate enough
4. **Sensitivity analysis and uncertainty quantification.** How a system responds under different conditions, difficult to do without an approach implemented on a computer

Typical practical issues requiring numerical methods

1. **Nonlinear versus linear.** Linearization often needed for analytical solution, but not appropriate in many cases (e.g., fluid dynamics)
2. **Large versus small systems.** Often not feasible to analyse large systems without a computer (rule of thumb: over three interacting components)
3. **Nonideal versus ideal.** Idealized laws are common in engineering, but often they are not accurate enough
4. **Sensitivity analysis and uncertainty quantification.** How a system responds under different conditions, difficult to do without an approach implemented on a computer
5. **Design.** Often easy to determine performance as a function of its parameters. Inverse problem (determining parameters to get specified performance) usually more difficult

One of the first example: control rockets

- The NASA team of 1969 needed to calculate the optimum trajectory for the Apollo 11 spacecraft to escape Earth's gravity, travel toward the Moon, and enter the Moon's field.

One of the first example: control rockets

- The NASA team of 1969 needed to calculate the optimum trajectory for the Apollo 11 spacecraft to escape Earth's gravity, travel toward the Moon, and enter the Moon's field.

- Many equations required to model the complex system effectively.

One of the first example: control rockets

- The NASA team of 1969 needed to calculate the optimum trajectory for the Apollo 11 spacecraft to escape Earth's gravity, travel toward the Moon, and enter the Moon's field.

- Many equations required to model the complex system effectively.
  - Equations of motion (planetary and rocket motion)
  - Rotations of both planets both on their axes and w.r.t. each other
  - Effects of gravity in both planetary systems
  - Effect of launch angle, atmospheric drag
  - Many more factors had to be calculated

One of the first example: control rockets

- The NASA team of 1969 needed to calculate the optimum trajectory for the Apollo 11 spacecraft to escape Earth's gravity, travel toward the Moon, and enter the Moon's field.

- Many equations required to model the complex system effectively.
  - Equations of motion (planetary and rocket motion)
  - Rotations of both planets both on their axes and w.r.t. each other
  - Effects of gravity in both planetary systems
  - Effect of launch angle, atmospheric drag
  - Many more factors had to be calculated

- Methods up to 1960s were manual and laborious
- Teams of engineers running hand calculation algorithms repeatedly in relay teams
- Results relayed over the phone to astronauts guiding trajectory management

One of the first example: control rockets

- The NASA team of 1969 needed to calculate the optimum trajectory for the Apollo 11 spacecraft to escape Earth's gravity, travel toward the Moon, and enter the Moon's field.

- Many equations required to model the complex system effectively.
  - Equations of motion (planetary and rocket motion)
  - Rotations of both planets both on their axes and w.r.t. each other
  - Effects of gravity in both planetary systems
  - Effect of launch angle, atmospheric drag
  - Many more factors had to be calculated

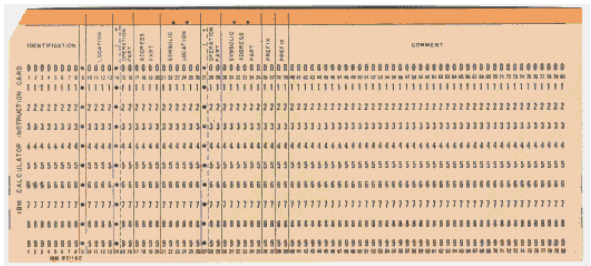


Figure: Punchcard of the 1960s

- Methods up to 1960s were manual and laborious
- Teams of engineers running hand calculation algorithms repeatedly in relay teams
- Results relayed over the phone to astronauts guiding trajectory management
- Not feasible approach for 100s of eventualities
- Had to be computerised to enable mission

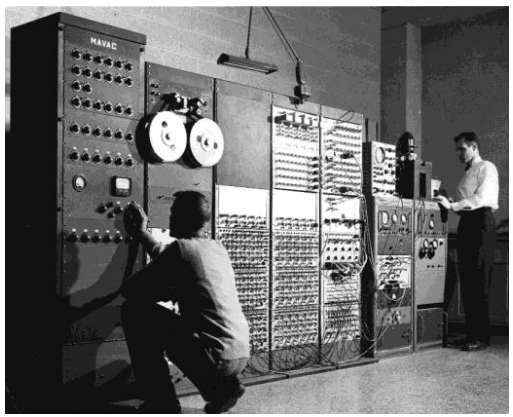


Figure: Large 1960 mainframe computers fed with cards and tape containing input parameters and source code

Recent example: full jet-engine simulation

- Computational Fluid Dynamics (CFD) simulation of compressor and combustion chamber
- Multi-physics simulation: fluid mechanics, heat transfer, combustion, radiation, ...
- Numerical methods to solve: partially differential equations, interpolation, numerical integration, optimization, ...
- Billion(s) degrees of freedom
- Massively parallel computing using tens/hundreds of thousands CPUs

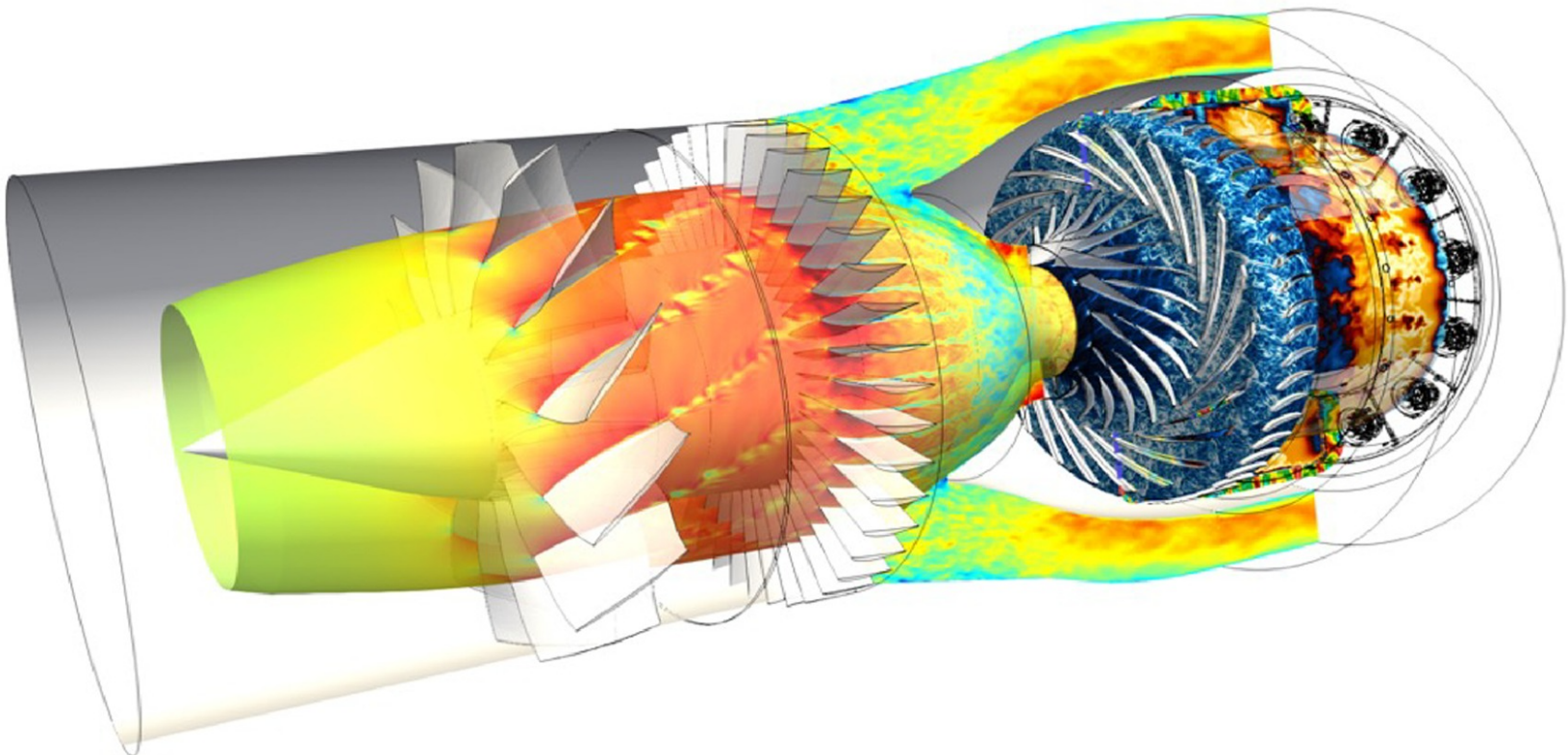


Figure: Simulation made at CERFACS. C. Pérez Arroyo et al., Journal of the Global Power and Propulsion Society, May, 2021, pp. 1-16. doi:10.33737/jgpps/133115.

First example, python code

- Task: calculate the sum of the squares of the first 20 natural numbers

$$sum = 1^2 + 2^2 + 3^2 + 4^2 + ... + 20^2$$

First example, python code

- Task: calculate the sum of the squares of the first 20 natural numbers

$$sum = 1^2 + 2^2 + 3^2 + 4^2 + ... + 20^2$$

- How long would it take you to do this by hand, even with a calculator?
- Can we automate this procedure (first n natural numbers)?
- Let's try writing such a code in Python

First example, python code

Sum of squares of first n integers

```
# ---------------------------
# Return the sum of
# square of first n
# natural numbers
def squaresum(n) :
    # Iterate i from 1
    # to n finding
    # the square of i and
    # add to sum.
    sm = 0
    for i in range(1, n+1) :
        sm = sm + (i * i)
    return sm
# ---------------------------

# ---------------------------
# Main Program
n = 20
print(squaresum(n))
# ---------------------------
```

- The function is defined by statement **def:** and needs n as input
- Comments start with # symbol. These are ignored by the code compiler/interpreter
- Block starting with **for** statement executes a summation in repeating loops until n+1 is reached
- A "Main program" calls the function that we defined above

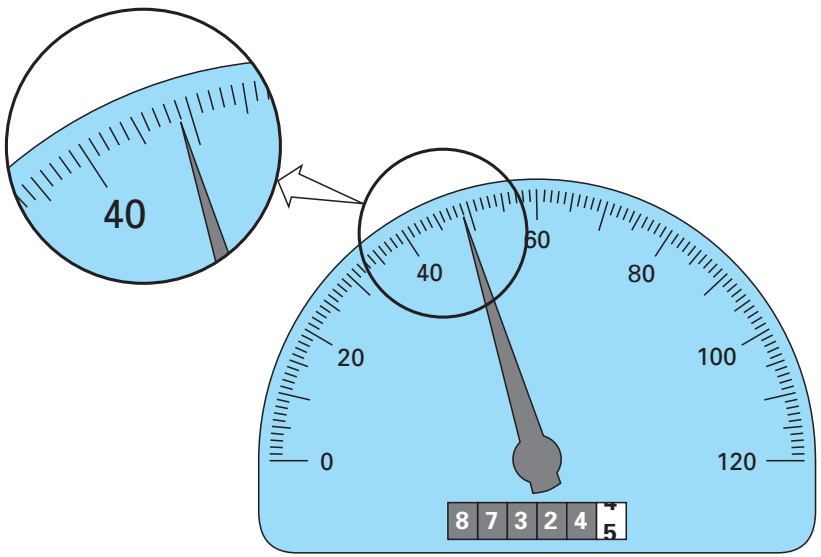
Definition and control of error

Error is a fundamental concept in numerical methods

- Every numerical method gives a solution with an error (with some lucky exceptions not very important in Engineering)

First important concept: Significant figures (significant digits)

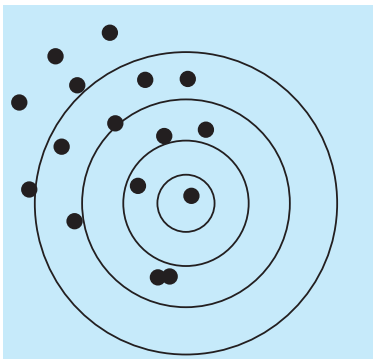
- When we employ a number in a computation, we must assess confidence



- Reading the speedometer, it is reasonable to say:
  - Speed is between 48 and 49
  - Speed is closer to 49
  - We can say with assurance that the car is travelling at approximately 49
- If we insist to estimate speed to one decimal place: one person might say 48.8, another might say 48.9
- Because of the limits of this instrument, only the first two digits can be used with confidence. Estimates of the third digit must be viewed as approximations.
- This number has two significant digits.

Definition and control of error

- We want to compute (good!) approximations of true value
- **Approximation:** a value or quantity that is nearly but not exactly correct.
- **Accuracy:** how closely a computed value agrees with the true value.
- **Precision:** how closely individual computed values agree with each other (i.e. spread of data).

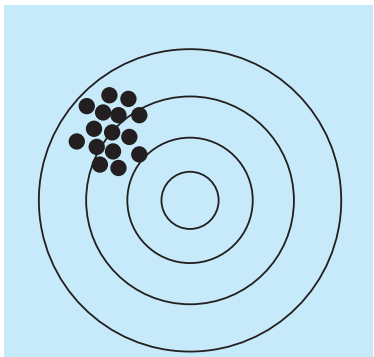


Precise ✗

Accurate ✗

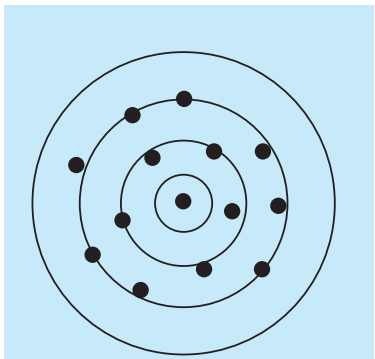
.

Big scatter and bias



Precise ✓

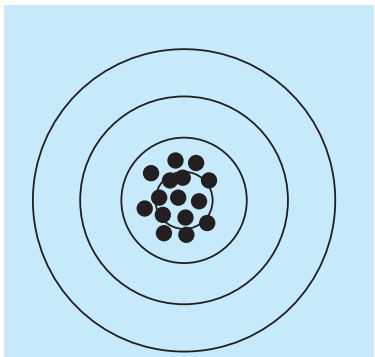
Accurate ✗



Precise ✗

Accurate **debatable**

Lack of accuracy due to lack of precision



Precise ✓

Accurate ✓

Error definition

- **True error:** The difference between the true value ϕ and the approximation A obtained with the numerical method:

$$E_t = \textit{true value} - \textit{approximation} = \phi - A$$

- **True fractional relative error:** account for the magnitudes of the quantities to normalize the error

$$E_f = \frac{E_t}{\phi} = \frac{\phi - A}{\phi}$$

- In percentage

$$\varepsilon_t = \frac{E_t}{\phi} 100\%$$

Error definition

- **Issue: we do not know the true value** ϕ (unless we know the analytical solution, which is almost never the case)
- We need to find a way to **estimate** the error
- **Approximate fractional relative error** for an iterative method:

$$e_a = \frac{\textit{current approximation} - \textit{previous approximation}}{\textit{current approximation}} = \frac{A_k - A_{k-1}}{A_k}$$

- e_a can be positive or negative. Often we are only interested in absolute value $|e_a|$
- This definition is often used as a way to stop the iterations, when a pre-selected acceptable error e_c is obtained

$$|e_a| < e_c$$

Example: compute $e^{0.5}$

- e^x can be expressed with a Maclaurin series expansion:

$$e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!}$$

- Truncating the series at n:

$$e^x = \sum_{i=0}^{n} \frac{x^i}{i!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots + \frac{x^n}{n!}$$

- Computing $e^{0.5}$ for different n:

n	num. result	true rel. error
0	1.0	-0.393469
1	1.5	-0.090204
2	1.625	-0.014387
3	1.6458333	-0.001751
4	1.6484375	-0.000172

Compute exponential

```
# ---------------------------
import math

x = 0.5
n = 5

# get exact value using
#intrinsic python function
ex_v = math.e**0.5

# initialise sum
e_to_x = 0

# do iteration n times (from i=0 to i=n-1)
for i in range(n):
    e_to_x = e_to_x + x**i/math.factorial(i)

# compute error
true_rel_error = (e_to_x-ex_v) / ex_v

print('numerical result',e_to_x)
print('exact value', ex_v)
print('true relative error',true_rel_error)
# ---------------------------
```

Exercise

- Can you modify the code in the previous slide to add error control?
- This means that the value of n should not be specified, but the iteration loop should be interrupted when a prescribed error E_c is achieved, e.g., $|E_f| < E_c = 10^{-7}$
- Instead of using the true relative error, can you use an error estimate?

Questions?