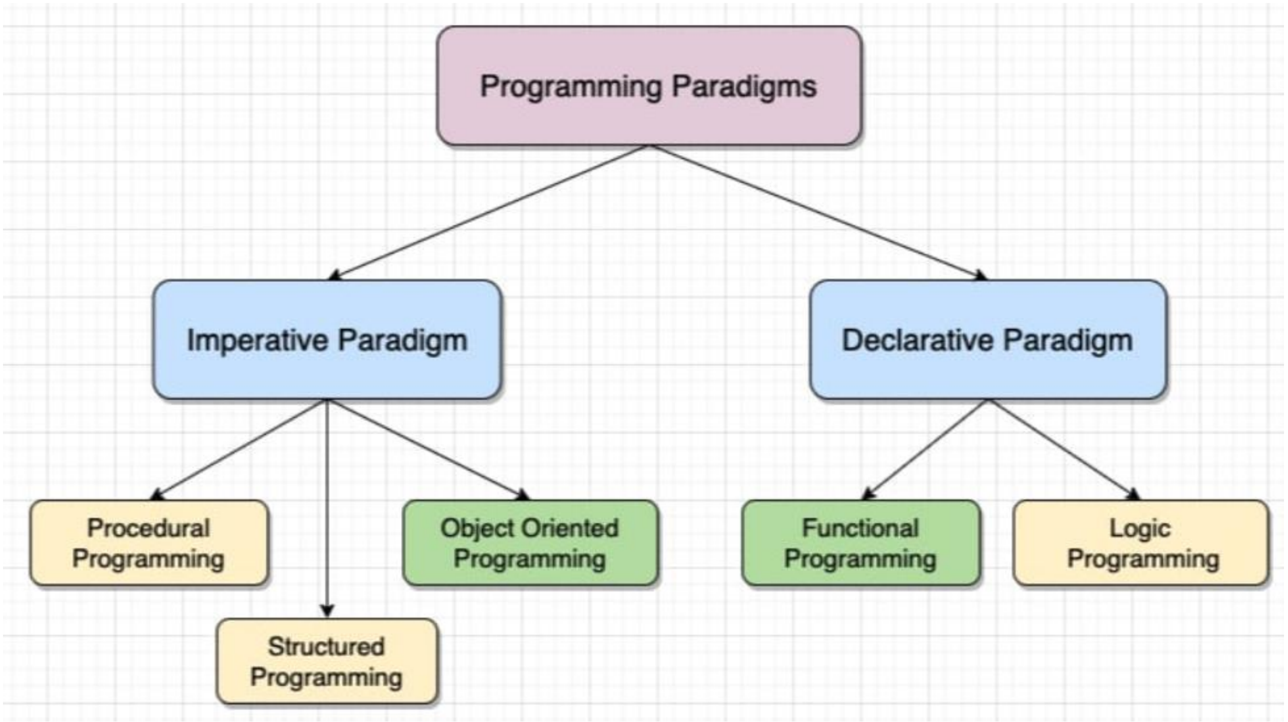
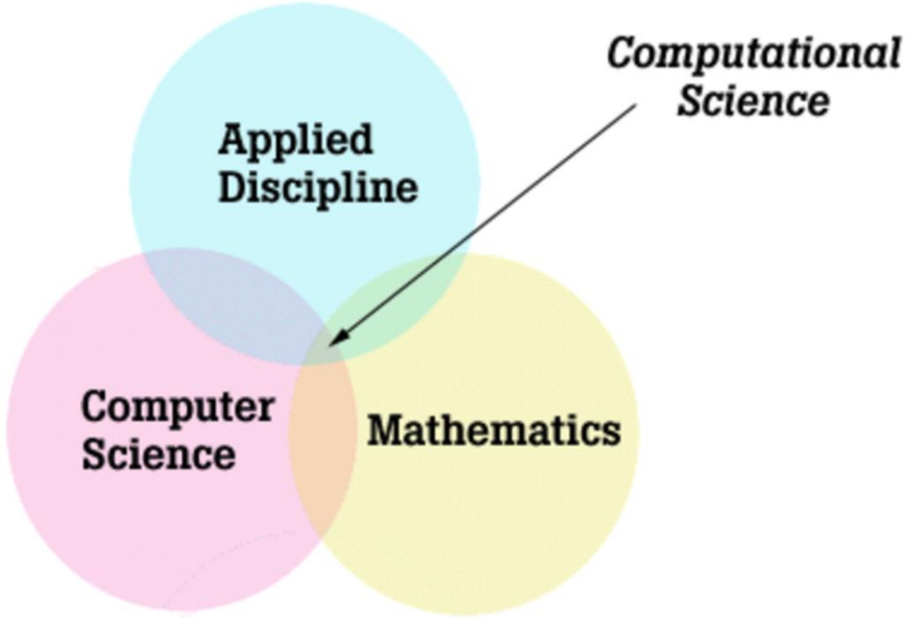
10. Programming paradigms & computational science





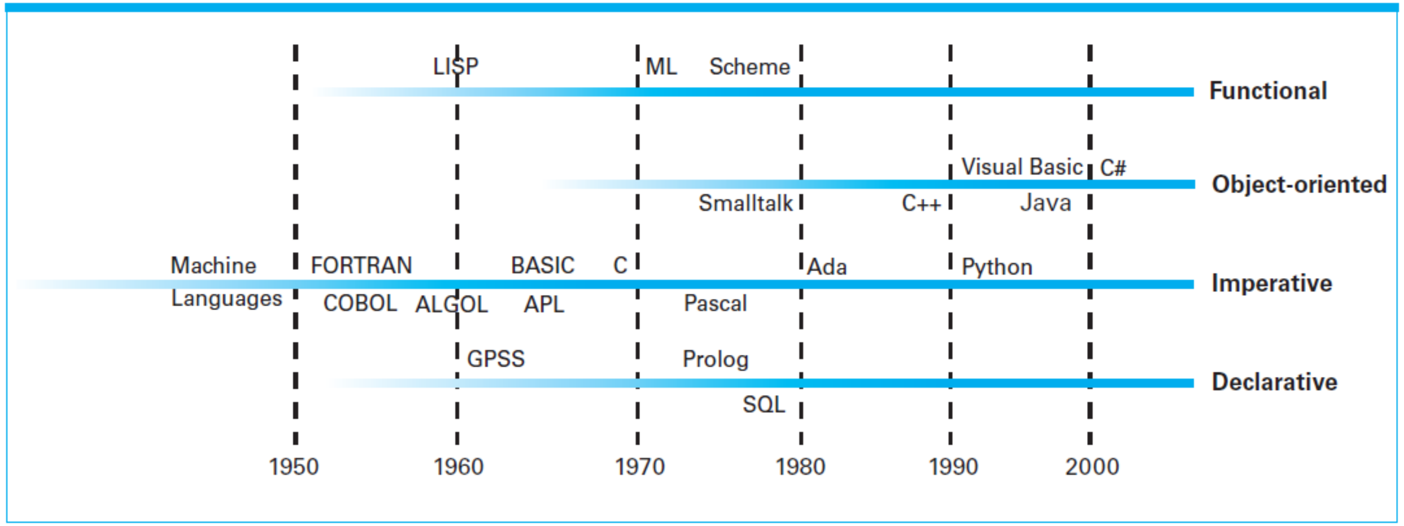
History of programming languages

- We've already discussed different programming languages, their evolution and translating from one language to another during the first half of this course
- Let's make a short revision:
  - Originally, computers were programmed using computer-specific (and essentially numeric) machine languages; these are commonly known as first-generation languages
  - Readability of code took a giant leap forward with the development of mnemonic assembly languages (which gave us the option to give variables and operations somewhat understandable names) – known as second-generation languages
  - The programs written in assembly language were still quite machine-dependent, because the primitives in them were equal to their machine language counterparts
  - Also, these primitives were very small as building blocks
  - Both these problems were solved by third generation languages, which were both machine independent as well as contained much larger primitives
  - Nowadays the generations span up to 5, but gen4 & gen5 languages are for specific purposes

Programming paradigms

- Generation-based approach is not the best for classification of languages
- Programming languages have developed along different paths as alternative approaches to the programming process – called *programming paradigms*
- These paradigms represent fundamentally different approaches on how to build solutions to presented problems
- Four main paradigms are
  - Imperative
  - Functional
  - Object-oriented
  - Declarative

Figure 6.2 The evolution of programming paradigms



Imperative paradigm

- Imperative paradigm (also known as the procedural paradigm; although some distinct procedural as a subset) is the traditional approach to programming
- Aim is to develop a sequence of imperative commands that (if followed) manipulates the data in such a way that the desired result is reached
  - Very engineer-ish “flowchart” method
- The structure of the program can be clarified by dividing it to subprograms called *procedures*
- Procedures can be executed sequentially (phases can be numbered) or concurrently
  - Concurrency requires additional control methods
- Examples: C, Ada, Pascal, FORTRAN, Basic, Cobol, …

Downsides to imperative paradigm

- Imperative (and procedural) algorithms have previously dominated programming, because they are very deterministic in nature and correspond to von Neumann-architecture
  - Troubleshooting is “easy”, because the problematic procedure can be identified
- On the other hand, if there are many procedures, meticulous attention needs to be put on the execution order; it is far too easy to design an algorithm that has large “visual complexity” and hence is prone for mistakes
  - For example, if there are several IF clauses: which “ELSE” is dedicated to which IF?
- Procedural algorithms are not intuitive in all cases
- Proof of correctness is a work-heavy task

Functional paradigm

- In functional paradigm, a program is seen as an entity that accepts inputs and produces outputs directly from the inputs by using smaller entities
  - These entities are called functions (hence the name)
- Program is constructed by connecting smaller functions in such a manner that the outputs of the previous function act as inputs to the next one
  - The ”main function” = the solution to the problem
- Information is presented as common data list structure
  - Data as well as the program are presented in same fashion
- Based on Church’s lambda calculus
  - Result of an algorithm is defined as a mathematical function (arguments = inputs)
- Examples: Lisp, Scheme, Erlang, Haskell, …

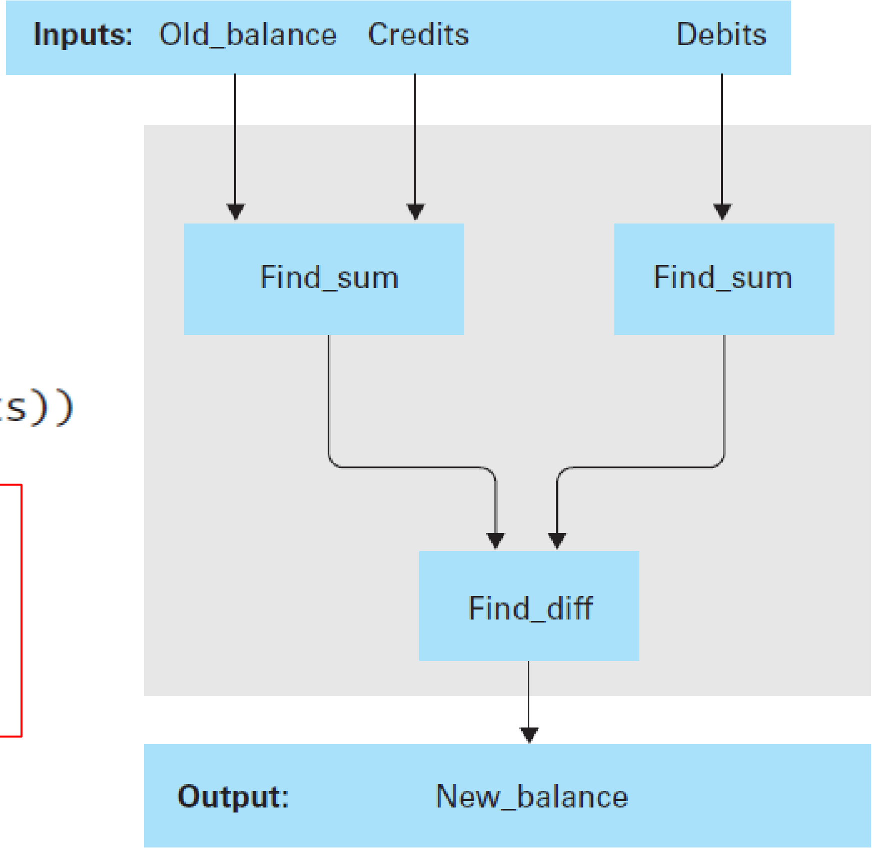
Example: imperative vs. functional paradigm

- Problem: find the balance of a checkbook
- Imperative:

```
Total_credits = sum of all Credits
Temp_balance = Old_balance + Total_credits
Total_debits = sum of all Debits
Balance = Temp_balance − Total_debits
```

- Functional:

```
(Find_diff (Find_sum Old_balance Credits) (Find_sum Debits))
```



At first glance, these may seem quite identical.
But note: imperative version has several statements, and the mid-results of all such statements are stored! Functional version only has one, which results in better efficiency.

Object-oriented paradigm

- In object-oriented paradigm, a software system is viewed as a collection of *objects* which can perform actions on themselves and/or request actions from other objects
  - For example, in a GUI, all icons are objects
- Methods linked to the object describe how the object responds to different events
  - Left mouse click, right mouse click, double click, …
- Advantage: if several programs use the same object, the functions needed are already provided with the object
- Description of the properties of an object is called a class
  - Creating multiple objects with same properties is easy (class can be applied to new objects)
- Foundation of most OOP languages is imperative, though - methods are small imperative program units
- Examples: C++, Java, Python

Declarative paradigm

- In declarative paradigm, programmer is asked to describe the problem to be solved
- Declarative algorithms are non-deterministic, so the execution order of phases of the algorithm is not typically known
- The system applies a general-purpose problem-solving algorithm
  - Such algorithms unfortunately don’t exist for very many problems
  - Commonly some deterministic search method is applied (depth-first search or similar)
- Declarative languages are therefore most suitable for some special applications
  - Hypothesis testing & predictions
  - Parallel computing
- Subfield: logic programming (find out whether claim x is true or not)
- Examples: Prolog, SQL

Concurrent vs. parallel computing

- The traditional programming method is to do everything in clearly ordered phases – so, in sequential fashion
- This limits us from making use of parallel computing
- Concurrent computing can be applied to sequential programs
    - Concurrent = multiple actions can be performed by rapidly switching the process in execution in order to give the user a feeling of parallelism
- True parallelism is based on parallel use of our system:
    - Multi-core processors / multiprocessor computers
    - Computer networks, clusters, decentralized computation
    - Supercomputers

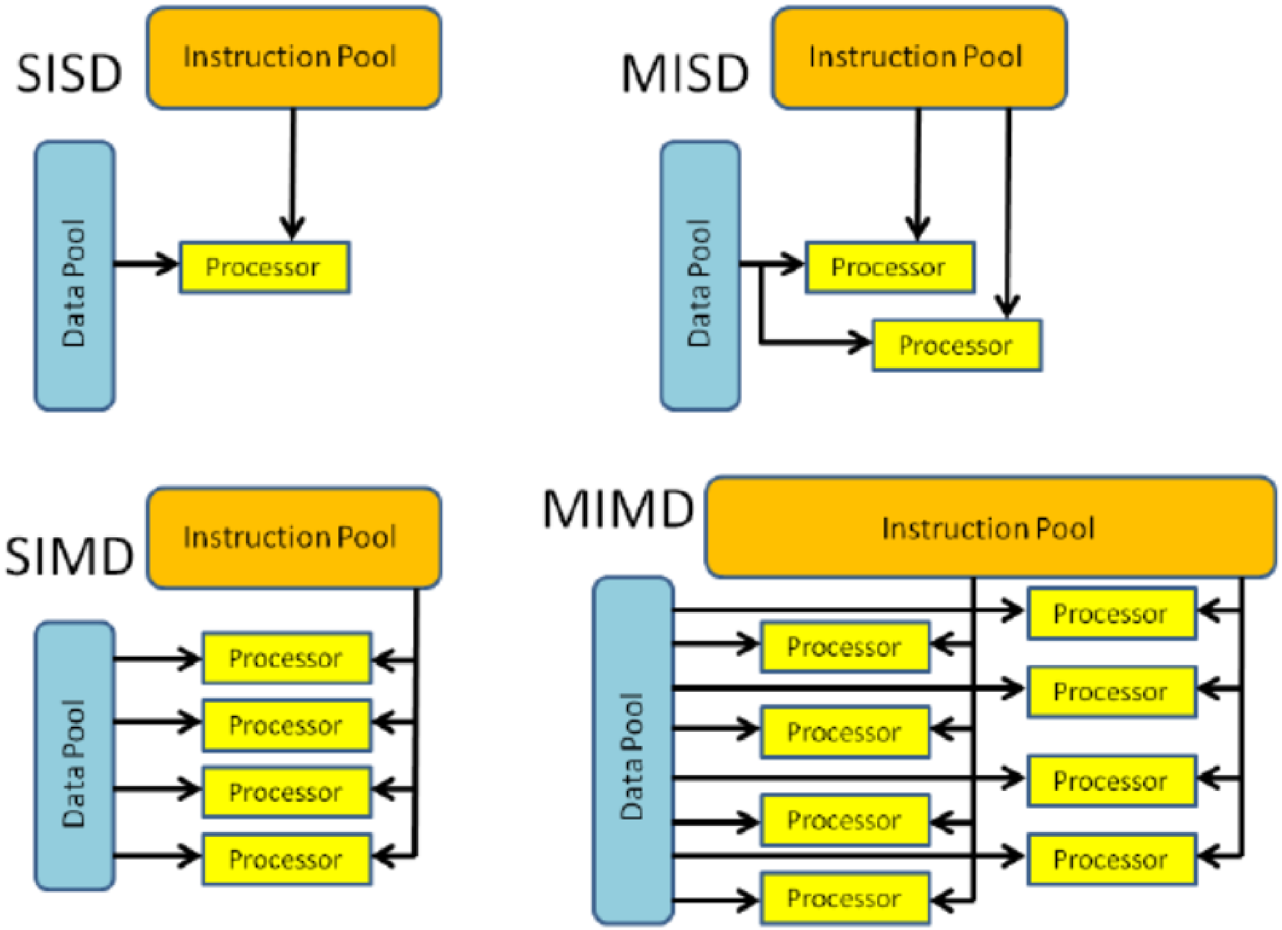
Parallel programming

- Not all problems are suitable for parallel computing
  - For example, problems where the previous phase produces the inputs of next phase
  - This is surprisingly common in practice
  - Parallel computing could be used in some phases, but not throughout the problem
- On the other hand, there are also problems for which parallelism feels natural
  - For example, face recognition: find a match from a database of 500 000 photos → if we have 10 computers available, the database can be divided so that each goes through 50 000 photos
- Designing a parallel algorithm may deviate a lot from a sequential algorithm
- There are programming languages which are especially suited for parallel programming (usually dependent on system or architecture)
- Parallelism in pseudolanguage: PARDO

```
FOR i := a,…,b PARDO
      <body>
END
```

Flynn’s taxonomy

- Computer architectures can be divided in four categories in relation to their ways to handle instruction & data streams:
  - SISD (single instruction, single data stream; all old single-core processors)
  - MISD (multiple instruction, single data; used only for fault detection)
  - SIMD (single instruction, multiple data; basically all modern GPUs)
  - MIMD (multiple instruction, multiple data; all modern multi-core processors)
- This division is known as *Flynn’s taxonomy*



Communication and speedup

- Especially in MIMD implementations, good communication between processors/cores is the key to performance improvements
  - Cores can share mid-results with each other etc.
- There are two ways to communicate and share information:
  - Shared memory (all cores can access the information)
  - Message passing (if memory is distributed; cores send “private messages” to each other)
- A benchmark quantity for measuring quality of parallelism is speedup
  - If the problem is divided to n cores/CPUs, how multiple will the performance rise?
  - Best-case: linear speedup = n (theoretical maximum)
- In special cases, a superlinear speedup is possible
  - For example, if breaking the problem to smaller parts results in all parts fitting in the cache (greatly reduced memory fetch times)

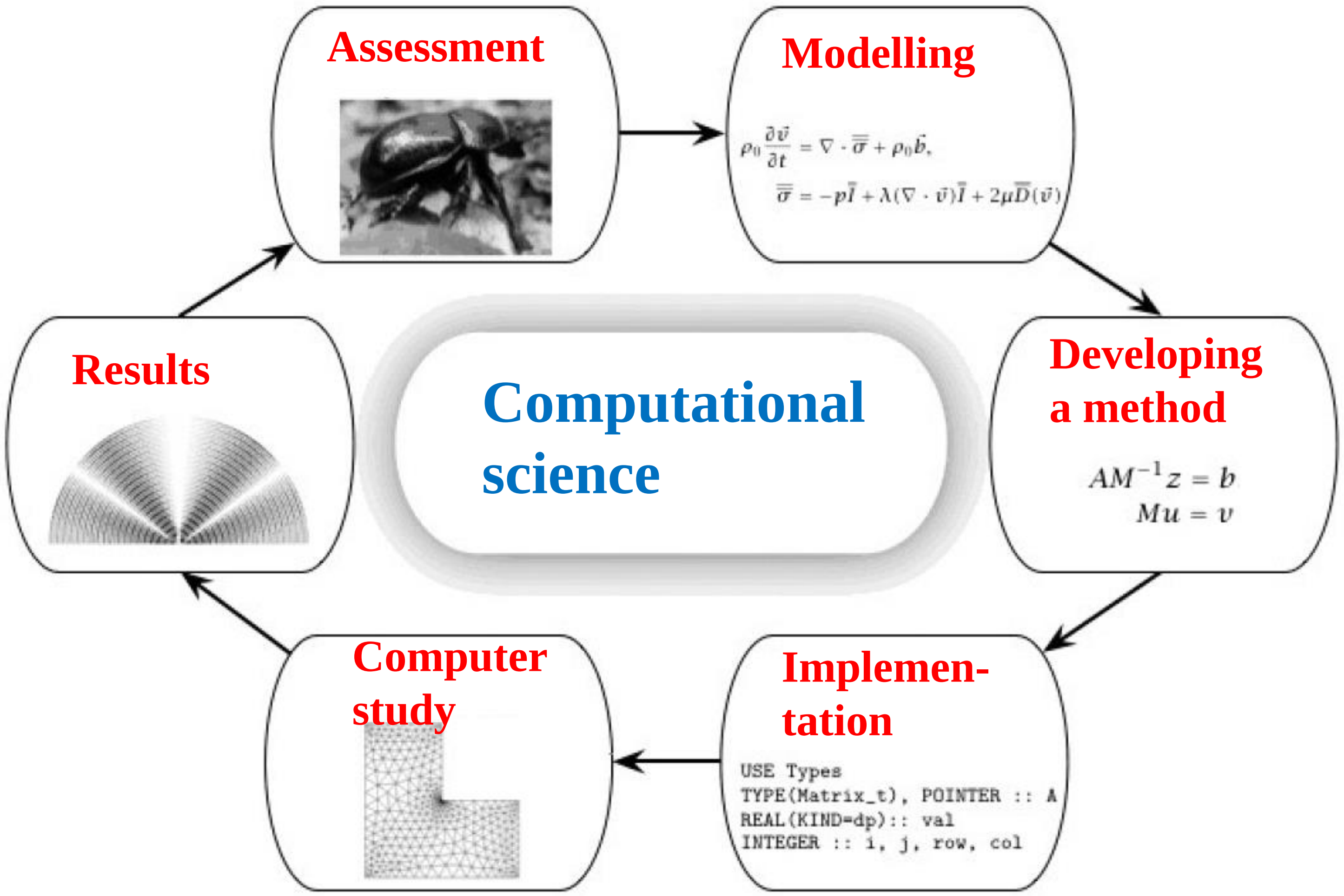
Computer science and mathematics

- Previously we've looked at things from the angle of a programmer
- We might remember from the first lecture that the first computer scientists were actually mathematicians; computer science and mathematics have a special bond
- There are problems in mathematics which are far too complex to solve in an analytical fashion
- The field that aims to solve these problems by taking advantage of modern computing capabilities is called *computational science*
- Computational science primarily deals with the most mathematical side:
  - Mathematical models
  - Algorithms
  - Optimization

Definition of computational science

- Computational science is a field mainly concentrated in three topics:
    1) Algorithms and programs, modelling & simulation knowhow
    2) Development of hardware, software, data communications technology and information processing components needed for solving challenging tasks
    3) Computing infrastructure that supports the problem-solving & development of related sciences
- Possibilities:
    - Create more realistic models for solving complex and wide study problems (climate change, garbage problem of seas, fusion energy, …)
    - Decrease the need for expensive experiments & prototypes
    - Study the correlations and causations between phenomena
    - Gain better understanding of things in the “big picture”
    - Improve risk control and tolerance to uncertainty

Process cycle of computational science



Modelling with data

- What part of the data is essential considering the question?
- How correct is the data we get from the sensors?
  - Calibration errors, measurement errors
- What internal relations does the data have?
  - Which data is “raw” and measured, which has been calculated from raw data?
- How can the data be saved during the measurement, are there delays?
  - For example, digital image correlation (DIC) systems require a quick hard drive
- Accessibility and transferability of data
  - Cloud storage or something else? Transferring hundreds of GBs of data is hard – especially if security is an issue

Types of mathematical models

Model type

Examined phenomenon →

Linear ↔ Nonlinear

Deterministic ↔ Stochastic

Static ↔ Dynamic

Homogenous ↔ Heterogenous

Black box ↔ White box

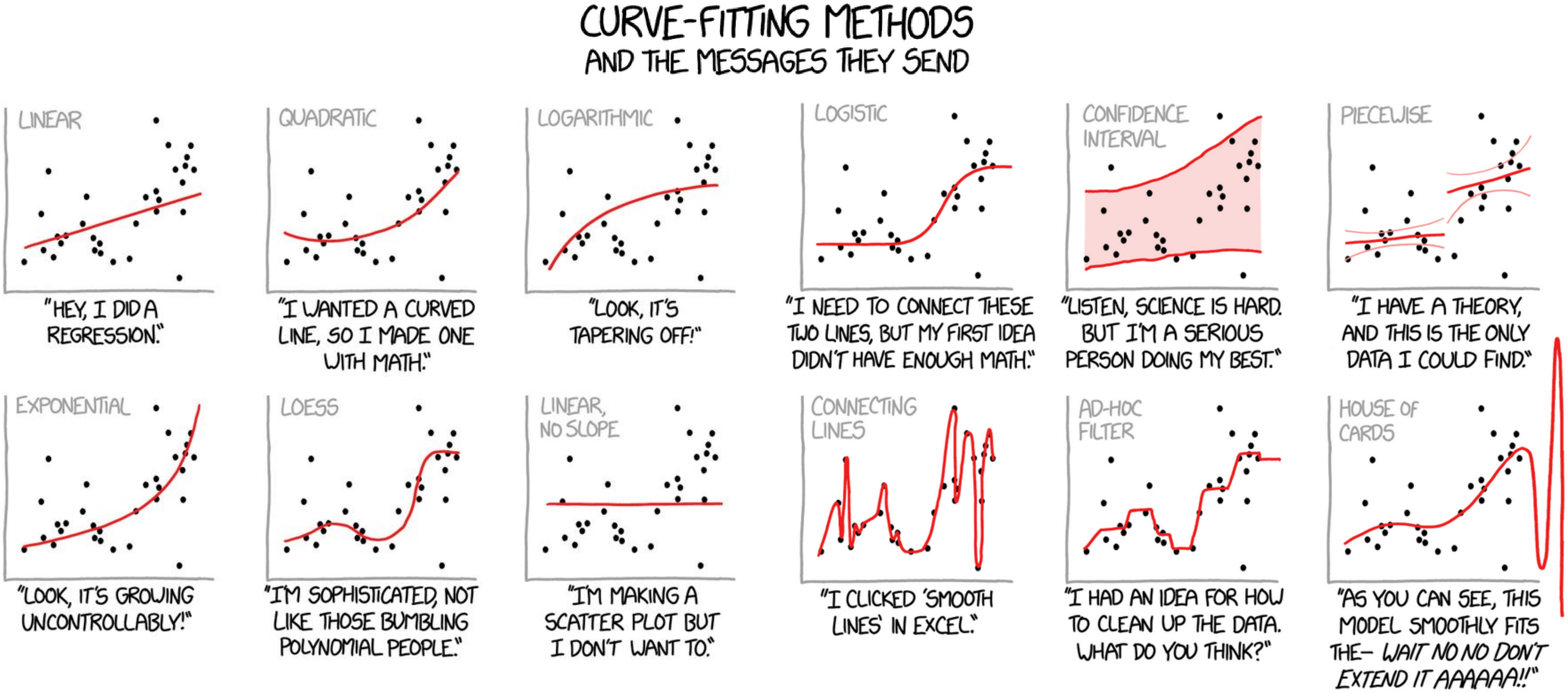
→ Verification

→ Optimization

→ Adjustment

→ Decision-making

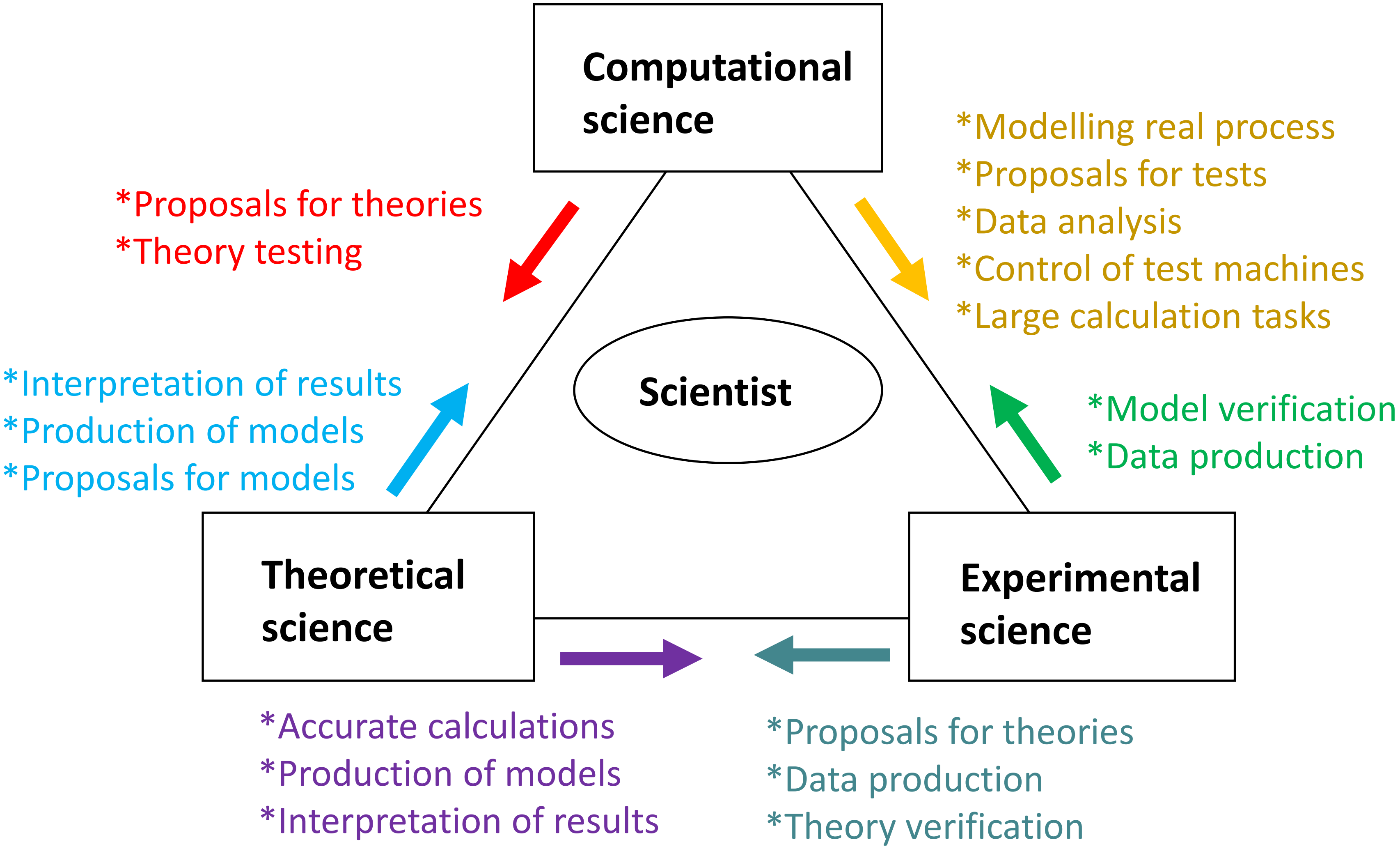
What model fits the data?



Optimization

- After a model has been created, it can be *optimized*
- Computational science has given rise to many advanced optimization techniques
- One good example are genetic algorithms, which mimic evolution:
  1) Initiation: create a random population of solutions
  2) Selection: select the best solutions using a fitness function and delete the worst ones. If a good enough solution has been found (or time has run out), stop and go to 5.
  3) Combination: perform crossovers and mutations to the remaining population on order to create a new population – the next evolution version
  4) Repetition: go back to stage 2 with the new population
  5) Post-processing & visualization of results, termination
- Especially good algorithms when problem is supposed to have multiple local minima where derivative-based methods may get stuck

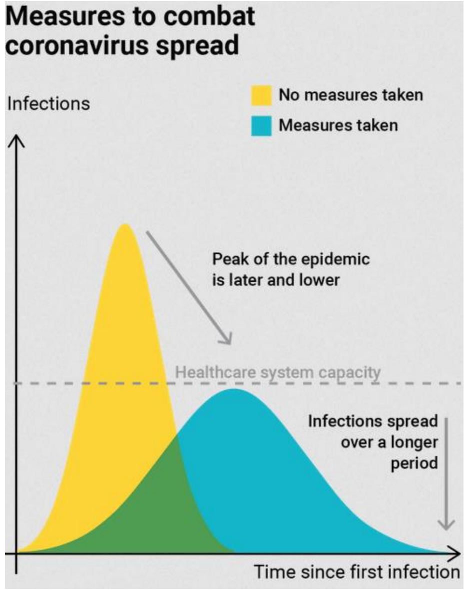
Role of computational science

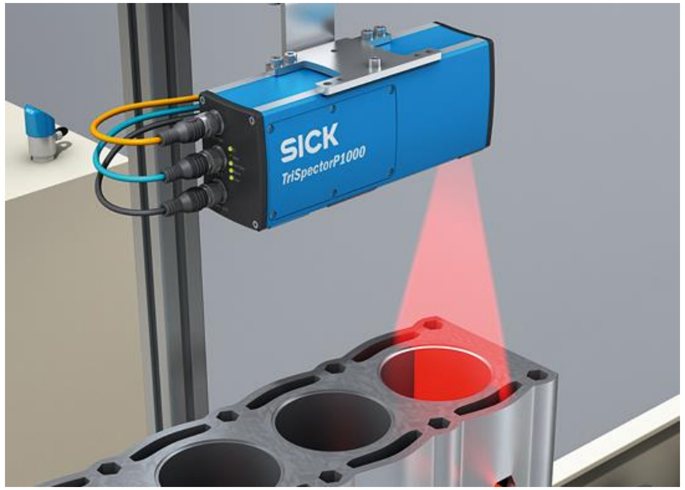


Application areas of computational science

- Modelling fusion reactor behavior
- Global epidemic models (ebola, covid-19)
- Streaming services (recommendation algorithms, user demand)
- Measurement & instrumentation technology
- Remote mapping of natural resources (laser scanning of forests etc.)
- Process diagnostics in factories
- Raw material analysis (optimal sawing of a log)
- Dynamic traffic steering
- Machine vision & pattern recognition applications

…and many, many more!





Thank you for listening!