COMP0043 Numerical Methods for Finance

1.1 Introduction

Guido Germano

Financial Computing and Analytics Group, Department of Computer Science, University College London
Systemic Risk Centre, London School of Economics and Political Science
UCL Centre for Blockchain Technologies

2 October 2024

Module team

Module leader: Prof. Guido Germano, Room 4.07, 66–72 Gower Street

Office hour: Wednesday 13–14pm in my office or on Zoom. Booking is recommended.

Teaching assistant: Junyi Yin, Room 5.12, 66–72 Gower Street

The teaching assistant helps with marking and is available for questions.

Administrator: Ricki Angel, Room 1.10, 66–72 Gower Street

Assessment: 40% test in Term 1, 60% exam in Term 3

Companion support course without exam: Introduction to Mathematics and Programming for Finance, taught by Dr. Carolyn Phelan. As it is not an examined module and thus does not give credits, it does not have a COMP0XYZ code and does not appear on Portico, but it has a Moodle page where you can self-enrol with the key IMPF2024.

History and focus

This module was set up in 2012 and taught for two years by Sebastian del Baño Rollin. He was a practitioner in banks in London, then a senior teaching fellow in our department and is now reader in mathematics at Queen Mary, University of London.

Sebastian and I shared an office in 2013–2014, I took over his PhD student Yiran Cui when he left, and we wrote together a few papers. We will discuss two of these papers on calibration in the second half of the module.

If you are interested, I kept Sebastian's teaching material at the bottom of the Moodle page. The content is different both in the choice of topics and of the programming language, for which Sebastian used Excel/VBA.

Until 2017–18 this module was called COMPG005 Numerical Analysis for Finance; "methods" rather than "analysis" reflects more accurately the syllabus (both mine and Sebastian's), whose focus is to turn mathematical equations into working code, rather than to prove theorems on analytical error bounds. Other names could be Scientific Computing for Finance, Computational Methods for Finance, or simply Computational Finance.

My first choice for the programming language was C++, but C++ is already covered by Riaz Ahmad in the term 2 optional module COMP0041 Applied Computational Finance, while most other modules were based on Matlab, so with the Head of Research Group Tomaso Aste we agreed on Matlab. In recent years we have been shifting to Python to meet students' interest, so you may submit your programming in Python if you prefer.

Requisite: COMP0048 Financial Engineering

This module is attended by a heterogeneous audience of students with different undergraduate backgrounds enrolled in several MSc programmes:

1. MSc Computational Finance, where it is a core module, and COMP0048 is core too.
2. MSc Financial Risk Management, where it is optional, and COMP0048 is core.
3. MSc Engineering with Finance, where it is optional, and MECH0092 is a requisite.

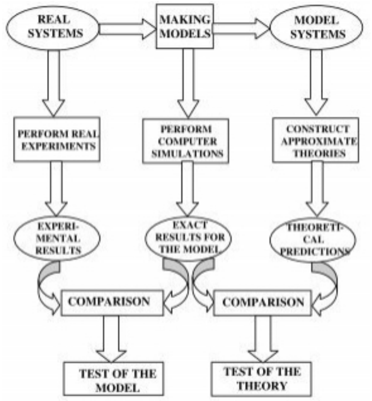
This module has been chosen also by students in other MSc programmes:

1. MSc Mathematical Modelling, where MATH0088 is a requisite optional.
2. MSc Scientific and Data Intensive Computing; requisites COMP0048 or MATH0088.
3. MSc Computational Statistics and Machine Learning, MSc Machine Learning.
4. MSc Statistics, where STAT0013 Stochastic Methods in Finance I is a requisite.
5. MSc Financial Mathematics, where the core modules are sufficient as requisites.

MATH0088 Quantitative and Computational Finance has been taught by Riaz Ahmad since 2005 in the MSc Mathematical Modelling seven years before the MSc Financial Mathematics and the MSc Financial Risk Management were set up in 2012. After being included in the diets of four MSc programmes (MM, FM, FRM, CF), it became overcrowded and so in 2015 the clone module COMP0048 taught by the same lecturer was made for CF, FRM, and later EwF.

Scientific computing or computational science

This area of research has emerged since the mid of the last century as a third pillar next to experiments and mathematical theory. At UCL it goes under the name of eResearch and is one of UCL's ten research domains.



M. P. Allen, D. J. Tildesley, *Computer Simulation of Liquids*, paperback edition, Oxford University Press, 1989, page 5, Fig. 1.2: Computer simulation as the connection between experiment and theory. Fig. 1.2 on page 5 of the 2nd edition of 2017 is similar.

Scientific computing and computational science

Computational Science is an interdisciplinary, cross-disciplinary, “transversal”, or “transdisciplinary” subject driven mainly by its applications in specific disciplines where mathematical problems cannot be solved analytically, but require the help of a computer: physics, chemistry, engineering, astronomy, meteorology, biology, finance, economics, etc.

Correspondingly, there are computational physics, computational chemistry, computational engineering, computational fluid dynamics, computational statistics, computational finance, computational archaeology, etc.

It is not surprising that UCL’s MSc Scientific Computing was started in 2016 as a cooperation between the Departments of Physics, Computer Science, and Mathematics; in 2020 it changed its name to MSc Scientific and Data Intensive Computing.

The UCL Dept. of Computer Science was born in 1980 from a split of the UCL Dept. of Statistics and Computer Science, which was an evolution of the UCL Dept. of Applied Statistics founded in 1911 by Karl Pearson as the first dept. of statistics in the world.

Catchier names for computational statistics are machine learning (introduced 1959; “machine learning is statistics for students in computer science”) and artificial intelligence (introduced 1955 by C. Shannon et al. in a research project proposal).

“Computational thinking will be a fundamental skill used by everyone in the world by the middle of the 21st century”: Jeannette M. Wing, “Computational thinking”, Communications of the ACM 49 (3), 33–35, 2006.

Murphy’s guide to modern science

If it’s incomprehensible, it’s mathematics.

If it doesn’t work, it’s physics.

If it explodes or stinks, it’s chemistry.

If it’s green and crawls, it’s biology.

If it’s dusty, it’s history.

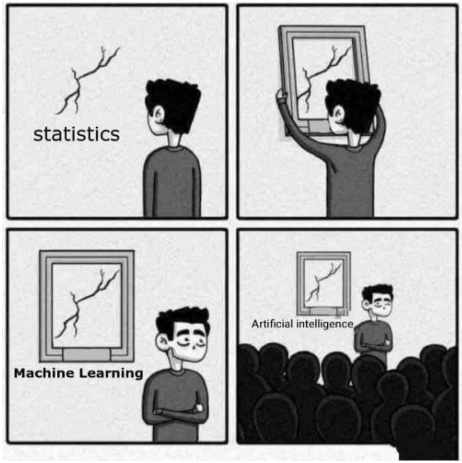
If it doesn’t make sense, it’s economics or psychology.

If it’s boring, it’s statistics.

People argue that “data science”, “machine learning” and “artificial intelligence” are successful rebrandings of (computational) statistics to overcome its image problem.

“When you’re fundraising, it’s AI. When you’re hiring, it’s ML. When you’re implementing, it’s logistic regression.” (source on next slide)

Is machine learning just glorified statistics?



Source: sandserifcomics, found in Joe Davison, “No, Machine Learning is not just glorified Statistics”, 27 June 2018.

Which operating system?

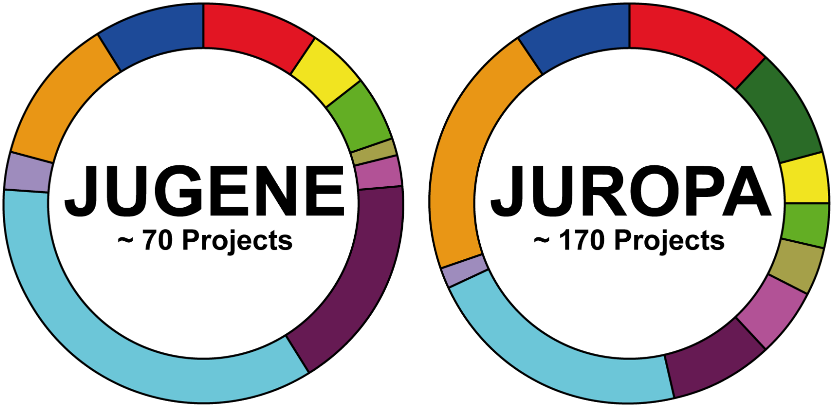
Serious large-scale scientific computing or computational science on massively parallel computers is done only with compiled languages in a Unix or Linux environment.

High-performance computers (at least all those I have seen) run only Unix or Linux as their operating system and support only Fortran and C/C++ compilers; sometimes also Ada, developed for the US Department of Defense. For a list of machines, see www.top500.org.

Interpreted languages like Matlab, R or Python are useful for quick small-scale development with the benefit of built-in graphical output. Julia tries to go beyond this.

Of course it may be subject to change. Please tell me if you find a machine in top500.org which runs Windows or anything else than Unix/Linux (very unlikely), or supports other programming languages than C/C++, Fortran and Ada (possible).

Use of high-performance computing at FZ Jülich in 2011



🟥 Astrophysics	🟩 Earth & Environment	🟦 Elementary Particle Physics
🟩 Biophysics	🟫 Plasma Physics	🟪 Computer Science
🟨 Chemistry	🟪 Soft Matter	🟧 Condensed Matter
	🟪 Fluid Dynamics	🟦 Materials Science

A brief history of operating systems

The development of Unix was started in 1969 at Bell Laboratories by Ken Thompson and Dennis Ritchie. First they used assembly language; in 1973, they rewrote Version 4 in the C programming language, which Dennis Ritchie had released the year before. Unix time is the number of seconds elapsed since 1/1/1970 UTC; try the command `date +%s`.

Unix ran on powerful workstations and required too much resources for the first personal computers and Macintoshes. Thus Microsoft bought the much simpler QDOS which then became MS-DOS (1981), while Apple developped its own System 1 (1984), later Mac OS.

When small computers increased their processing power, Steve Jobs built the NeXTSTEP operating system (1989) upon BSD Unix, which became the basis of Mac OS X (2001), now macOS, and Linus Torvalds developped the Unix-like operating system Linux (1991) for Intel architectures. Also iOS (2007) is based on BSD Unix and Android (2007) on Linux.

Although Microsoft owned since 1980 a Unix distribution called Xenix, it based Windows NT (1993) on a different development which shares similarities with VMS, the operating system of the VAX computers of Digital Equipment Corporation: in 1988, Microsoft hired DEC's chief developper David Cutler, who hated Unix as “a junk operating system designed by a committee of PhDs”. At Bell, Thompson and Ritchie developped Unix on a DEC PDP-11.

Thompson and Ritchie received the Turing Award (1983) and the Computer Pioneer Award (1994); Thompson became Member of the National Academy of Sciences (1985), Ritchie got the National Medal of Technology (1998), etc. Cutler's list of awards is much shorter.

Dennis Ritchie and Ken Thompson at a DEC PDP-11 in 1970

Dennis Ritchie (standing) and Ken Thompson at a DEC PDP-11 16-bit minicomputer in 1970, on which they developped Unix at Bell Laboratories. Source: Wikipedia.

Which programming language?

C++ is used in COMP0041 Applied Computational Finance. The MSc Scientific and Data Intensive Computing offers COMP0210 Research Computing with C++ by Jamie Quinn.

Some **Excel** is still used by Riaz Ahmad in COMP0048 Financial Engineering.

Python has been included in Introduction to Mathematics and Programming for Finance, COMP0041 and most other modules. The MSc Scientific and Data Intensive Computing offers COMP0233 Research Software Engineering with Python by David Perez-Suarez.

Matlab is still used alongside with Python also in COMP0040 Data-driven Modelling of Financial Markets, COMP0047 Data Science, COMP0050 Machine Learning with Applications in Finance, etc.

Different employers in finance require different languages: C++, C#, Matlab, Python, Julia, R, Rust. There is no single solution, so you must be flexible, and things change over time.

Fortran is still used a lot in large sectors of scientific computing, but little outside this niche, and historically it has not played a role in finance other than in function libraries.

Java and, before that, **Pascal** have been taught as the first programming language to undergraduates in computer science, but did not catch on in scientific computing.

The future may be **Julia**, yet nobody knows whether it will become widely adopted. Read J. Bezanson, S. Karpinski, V. B. Shah, A. Edelman, “Why we created Julia”, 14 Feb 2012.

Which programming language in computational finance?

There are thousands of programming languages and dozens of major ones. Choosing one depends a lot on the environment one interacts with, and is subject to fashions.

The TIOBE programming computing index measures the popularity of a Turing-complete, i.e. universal, programming language from the number of search engine queries that contain the name of the language, but it does not disaggregate for specific sectors like scientific computing or computational finance.

“Honesty compels us to point out that in the 20-year history of *Numerical Recipes*, we have never been correct in our predictions about the future of programming languages for scientific programming, not once! At various times we convinced ourselves that the wave of the scientific future would be ...Fortran ...Pascal ...C ...Fortran 90 (or 95 or 2000) ...Mathematica ...Matlab ...C++ or Java .... Indeed, several of these enjoy continuing success and have significant followings (not including Pascal!). None, however, currently command a majority, or even a large plurality, of scientific users.”

W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery, *Numerical Recipes in C++: The Art of Scientific Computing*, 3rd edition, Cambridge University Press, 2007, page 1.

Further reading

Toby Driscoll, Matlab vs. Julia vs. Python, https://tobydriscoll.net/blog, 28 June 2019.

Jon Danielsson, Yuyang Lin, Choosing a numerical programming language for economic research: Julia, MATLAB, Python or R, VoxEU, 13 August 2022, https://cepr.org/voxeu.

Mohammed Ayar, Python is destroying the planet, Level Up Coding, 4 October 2022, https://levelup.gitconnected.com.

- Wikipedia: Lists of programming languages
- Wikipedia: History of programming languages
- Wikipedia: Timeline of programming languages

Programming paradigms

These languages share many features: they are high-level Turing-complete **imperative** multiparadigm languages with similar syntaxes, which support, among others, software design techniques for structured, procedural, modular and object-oriented programming.

For **declarative** rather than imperative programming, have a look at Haskell (1990), taught in COMP0020 Functional Programming by Christopher Clack, who also teaches COMP0075 Financial Market Modelling and Analysis. Haskell is an example of a pure **functional** language, which is a particular form of declarative programming. Another form is **logic** programming, implemented e.g. in the programming language Prolog (1972).

(Chris originally set up the MSc Financial Computing in 2006, which was substituted by the MSc Computational Finance in 2015. The MSc Financial Mathematics and the MSc Financial Risk Management were set up in 2012 by Prof. William Shaw, who in 2013 left for Winton Capital.)

Programming language generations: 1st, 2nd and 3rd

A **low-level programming language** has no or little abstraction from the processor's instruction set, and thus is close to the hardware. It deals with registers, memory addresses and call stacks, is optimised for a given system architecture and thus is not well portable.

The *first generation* is machine code, which runs directly on the processor without a compiler or interpreter. The *second generation* is assembly language, which provides one abstraction level over machine code and is transformed into the latter by an assembler.

A **high-level programming language** has a strong abstraction from the hardware details of a particular computer. It deals with variables, arrays, objects, control structures, functions, and focuses on usability by a human programmer rather than on runtime optimisation, at the cost of an abstraction penalty. The first working example was Fortran (1956), developped by IBM and still in use today, e.g. by the BLAS library included in TensorFlow. Unix was the first operating system written in a high-level programming language, C (1972), which made it portable across a variety of computers.

Most currently used programming languages are *third generation*; besides Fortran (now 2018) and C, e.g. Lisp (1960), Objective-C (1984), C++ (1985), Python (1991), Java (1995), JavaScript (1995), C# (2000), Clojure (2007), Go (2009), Kotlin (2011), Swift (2014), Rust (2015), etc. Also Algol (1958) and Pascal (1970), which are not used any more, belong to this category, as well as COBOL (1959) that is still used for business applications.

Programming language generations: 4th and 5th

Since Forth (1968), *fourth generation* programming languages are defined as having an even higher level of abstraction and ease of use, possibly at the cost of less versatility; e.g. SQL (Structured Query Language, 1974), a **domain-specific** programming language for the management of relational databases which is not Turing-complete.

Also Prolog (1972), Matlab (1984), Haskell (1990), R (1993) and Julia (2012) are usually listed as fourth-generation. They are all Turing-complete, but only Julia is **general-purpose**, whereas the others are considered domain-specific or specialised languages: Prolog for symbolic and logical programming, artificial intelligence, expert systems and natural language processing; Haskell for functional programming, compiler development and theorem proving; Matlab for scientific computing and R for statistics.

Fifth generation programming languages have been discussed since the 1980s; they are supposed to make the computer solve a problem with a minimum of programming at the highest level of abstraction, but the concept is not well defined. They are thought to be declarative rather than imperative and used for artificial intelligence and expert systems.

Large language models based on generative artificial intelligence like GitHub Copilot (2021), ChatGPT (2022) and Google Bard (2023) that understand natural language have fifth generation features, but are software engineering tools that help developping code rather than programming languages.

Algorithms and data types

An **algorithm** is a sequence of steps that leads from input data to an expected output. It can be described in plain language, pseudo-code or a programming language; in the latter case it can be run on a computer. An algorithm should be:

1. correct;
2. easy to understand and possibly elegant;
3. efficient with respect to its use of computing resources, i.e. CPU time and memory.

A **Turing machine** (1936) is an abstract computer that can implement any computer algorithm. A programming language is Turing-complete if it can simulate a Turing machine. Turing-complete languages have the features and constructs necessary to perform general-purpose computations, although they are not all equally good at it.

A **data type** is a set of values, and possibly of operations that can be performed on them.

A **primitive** data type is provided by the programming language: int, char, float, double, etc.

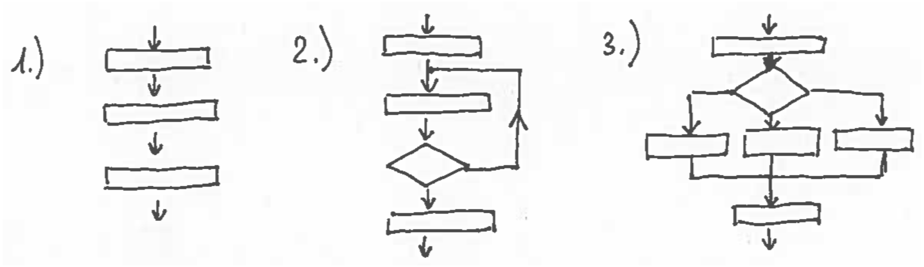
A **structured** data type (structure) is created by the user aggregating primitive data types.

A **class** combines a structured data type with functions that manipulate its attributes. An **object** is an instance of a class. The bundling of data with the methods to modify them is called **encapsulation** and often has the purpose of **information hiding** to protect particular regions of the memory from direct access and thus from invalid manipulation.

Flow-control structures and structured programming

Early programming languages allowed undiscriminated ‘go to’ instructions, i.e. jumps from a line of the code to another arbitrary line before or after. In 1968 a famous paper by Edsger Dijkstra, “Go to statement considered harmful”, objected against this practice and popularised the proof of Corrado Böhm and Giuseppe Jacopini (1966) that three **flow-control structures** are sufficient to express any algorithm in an imperative programming language:

1. sequential execution
2. iterated execution (jump backward): for, do, while, do while, break, continue, ...
3. conditional execution (jump forward): if – else if – else, switch – case – default, ...



Structured, procedural, modular and OO programming

Structured programming uses only these three flow-control structures, *with sequential or nested but not intertwined go tos*. This helps writing code that is easy to read and maintain. In particular, it avoids the undiscriminated go to jumps of early programming languages that lead to so-called “spaghetti code” with a complicated flow-control structure that is difficult to understand. The first Fortran had only if and go to; Algol 60 introduced while and if-else.

Procedural programming splits the code into separate procedures, also called functions, (sub)routines or methods, which call each other through clearly defined interfaces.

Modular programming is an extension of the “divide et impera” approach of procedural programming in order to manage large software projects. It groups procedures into namespaced files or modules so that a variable or procedure does not conflict with an identifier that shares the same name, but is in another file or module.

Object-oriented programming is a further approach to manage large software projects that is based on classes and has become dominant since the 1980s.

Structured, procedural and modular programming organise code at small, medium and large scale; object-oriented programming organises data.

Array programming

Array programming avoids iterated execution over array elements by exploiting operations on whole arrays. The result is more compact, readable and efficient code. It was pioneered by APL (1966) and then taken over by Matlab, Fortran 90, Julia and other programming languages. It allows e.g. to add and multiply matrices with the commands `A+B` and `A*B`, or perform elementwise operations, e.g. `A.*B` and `A./B`, whereas the older approach of scalar languages like Pascal, C and Fortran 77 was to nest operations on single array elements inside for loops. For instance, in C the multiplication of two $n \times n$ matrices **A**, **B** is done with

```
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++) {
        C[i][j] = 0;
        for (k = 0; j < n; j++)
            C[i][j] += A[i][k]*B[k][j];
        }
```

whereas in Matlab and Julia it is just

```
C = A*B
```

Array programming

Python and C++ do not natively support array programming, and manipulation of arrays has to be achieved with libraries. For instance, matrix multiplication in Python is done with

```
import numpy as np
C = np.dot(A,B)
```

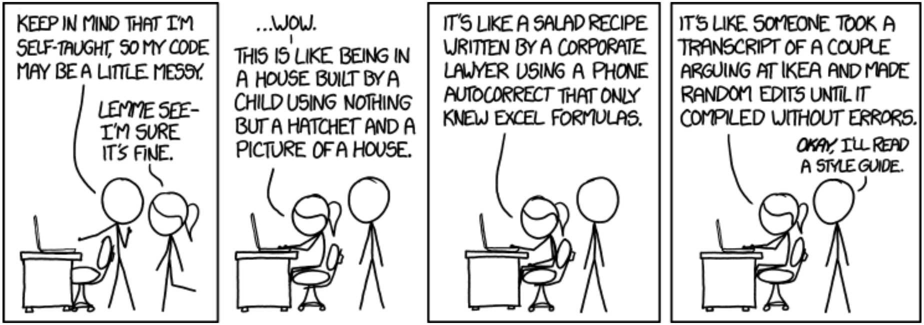
Also in Matlab matrix multiplication can be done with a function call,

```
C = mtimes(A,B)
```

but `C = A*B` with a matrix class and an overloaded multiplication operator is more elegant, closer to mathematical notation, and native in Matlab and Julia.

In C++, the Standard Template Library released in 1998 includes a vector class. A matrix class was planned too, but not completed. The situation has not changed since then, and people work around this with a vector of vectors or with one of many ad-hoc libraries.

Programming style


KEEP IN MIND THAT I'M SELF-TAUGHT, SO MY CODE MAY BE A LITTLE MESSY.
LEMME SEE— I'M SURE IT'S FINE.
...WOW.
THIS IS LIKE BEING IN A HOUSE BUILT BY A CHILD USING NOTHING BUT A HATCHET AND A PICTURE OF A HOUSE.
IT'S LIKE A SALAD RECIPE WRITTEN BY A CORPORATE LAWYER USING A PHONE AUTOCORRECT THAT ONLY KNEW EXCEL FORMULAS.
IT'S LIKE SOMEONE TOOK A TRANSCRIPT OF A COUPLE ARGUING AT IKEA AND MADE RANDOM EDITS UNTIL IT COMPILED WITHOUT ERRORS.
OKAY, I'LL READ A STYLE GUIDE.

http://xkcd.com/1513

Further reading: W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery, *Numerical Recipes in C++: The Art of Scientific Computing*, 3rd edition, Cambridge University Press, 2007, Section 1.2.3 “How Tricky is Too Tricky?”

Von Neumann architecture

Electronic digital stored-program computers have essentially been built according to the same design proposed by John von Neumann in 1945:

1. A central processing unit (CPU) with some internal memory, consisting of:
    a. a control unit with instruction registers, a program counter and a system clock;
    b. an arithmetic-logic unit (ALU) with processor registers that performs arithmetic and logical operations on integers, and a floating-point unit (FPU) that performs operations on floats;
    c. an address-generating unit (AGU), a memory management unit (MMU) that stores and retrieves data and instructions, and a small fast CPU cache memory organised in levels.
2. A large external memory (i.e. external to the CPU):
    a. primary or main memory (onboard): random-access memory (RAM);
    b. secondary or mass memory (online): hard-disk drive (HDD), solid-state drive (SSD/flash);
    c. tertiary or bulk memory: tape robot (nearline), removable drives (offline), cloud storage.
3. Input/output (I/O) devices:
    a. Input: keyboard, mouse, trackpad, pencil, joystick, etc.
    b. Output: monitor, beamer, printer, etc.
    c. Input and output: network cards (Ethernet, WLAN, Bluetooth, etc.)
4. A system bus that connects the above parts.

Von Neumann and memory bottlenecks

CPU, memory and I/O devices ara connected through a single system bus; the same hardware is used to encode, store and retrieve both program instructions and data.

This design, also called Princeton architecture, is simpler than the Harvard architecture with two separate memories and buses for instructions and data, but suffers of the von Neumann bottleneck: an instructions fetch and a data operation cannot be done at the same time because they share a common bus. In modern computers this bottleneck is mitigated with separate level 1 caches for instructions and data as well as other devices.

A modern CPU is so fast that usually it is limited by memory access anyway and spends much of its time idle, waiting for memory read/write to complete.

FLOPS

The numerical performance of a computer is measured in floating-point operations per second, FLOP/s or FLOPS. Parallel computers have many CPUs, a modern CPU contains several cores, and each core does several FLOPs per cycle, so that

$$\text{FLOPS} = \text{CPUs} \times \frac{\text{cores}}{\text{CPU}} \times \frac{\text{cycles}}{\text{seconds}} \times \frac{\text{FLOPs}}{\text{cycle}}.$$

With fused multiply-accumulate (MAC or FMA3) and multiply-add (MAD or FMA4), a core does a multiplication and an addition and thus two FLOPs in one clock cycle: FMA3 uses three registers and does `c += a*b`, FMA4 uses four registers and does `d = a*b + c`. They were introduced in the IBM POWER1 processor (1990) and are examples of single-instruction multiple data (SIMD) parallelism according to the Flynn taxonomy of 1966.

FMA3 has been available in the Intel x86 series since the Haswell (2013) processor. Thanks to this and other SIMD operations, each of the 8 cores of an Intel Rocket Lake (2021) processor can do up to 32 FLOPs per cycle; the processor operates at 5 GHz and thus has a theoretical peak performance of about 1 TFLOPS,

$$1280\,\text{GFLOPS} = 8\,\text{cores} \times 5\,\text{GHz} \times 32\,\text{FLOPs/cycle}.$$

In July 2023 Intel released the Ponte Vecchio processor with 52 TFLOPS.
In 1990, a processor did 1 MFLOPS.

Memory hierarchy

There is a **memory hierarchy** with a trade-off between response speed and capacity, ranging from internal memory (processor registers and cache) with the lowest response time and capacity to tertiary memory with the largest response time and capacity.

Memory type	Size	Access speed
Processor registers	few KB	1 clock cycle
L0 micro operations cache	6KB	few clock cycles
L1 instruction cache	128 KB	700 GB/s
L1 data cache	128 KB	700 GB/s
L2 shared cache	1 MB	200 GB/s
L3 shared cache	6 MB	100 GB/s
L4 shared cache	128 MB	40 GB/s
Main memory (DDR4 SDRAM)	16 GB	10 GB/s
Mass memory or storage (flash)	1 TB	2 GB/s
Bulk memory or storage	PB–EB	200 MB/s

Approximate figures for a dual-core Intel Core i7 Haswell Mobile 3 GHz processor in an Apple MacBook Pro 13” of 2014. DDR4 SDRAM = double data rate 4 synchronous dynamic random-access memory.

Reference locality and cache hits

The transfer efficiency between the memory hierarchy levels is enhanced by **reference locality** causing **cache hits**. Spatial locality or **data locality** consists in the sequential access of adjacent (or at least nearby) memory addresses; temporal locality consists in the repeated access of the same memory addresses.

For instance, because memory is addressed linearly, a matrix can be stored by columns (column-major languages, e.g. Fortran, Matlab, R, Julia) or rows (row-major languages, e.g. C, C++ and Python). Traversing a matrix in the same way as it is stored is more efficient because it exploits spatial locality of reference and thus CPU caching; moreover, contiguous access enables low-level SIMD instructions that operate on vectors of data. This is done by array instructions like `C = A*B` where `A,B,C` are matrices.

A **cache-oblivious** algorithm takes advantage of the CPU cache without knowing its size. The concept was introduced in 1996. There are optimal cache-oblivious algorithms for matrix multiplication, matrix transposition, LU decomposition, the fast Fourier transform, sorting, etc. Many of these algorithms were known before 1996.

Computational complexity

The computational complexity of an algorithm is measured by the amount of resources it requires:

1. CPU time, which is proportional to the number of required elementary operations;
2. memory;
3. for distributed computing, also the communication between the processors.

Library functions and array instructions implement computationally efficient algorithms.

E.g., standard multiplication of two $N \times N$ matrices requires about $2N^3$ operations. In 1969, Strassen introduced a matrix multiplication algorithm which requires $7N^{\log_2 7} - 6N^2$ operations, where $\log_2 7 \approx 2.807 < 3 = \log_2 8$; see Numerical Recipes, Section 2.11. The crossover with standard matrix multiplication is at $N \approx 1000$. However, this comes at the cost of a weaker numerical stability. The Strassen algorithm is cache-oblivious.

Over the years, Strassen-like algorithms have been discovered with an asymptotic complexity of $O(N^\omega)$ operations where $\omega \approx 2.373$, but they are practically irrelevant because the crossover is at too large N for contemporary computers.

Solution of linear systems of equations

Given a linear system of N equations with N unknowns $\mathbf{x}$,

$$\mathbf{Ax} = \mathbf{b}, \tag{1}$$

where $\mathbf{A}$ is an $N \times N$ matrix of coefficients and $\mathbf{b}$ is an N-dimensional column vector of known terms, the solution requires $(N+1)! + N$ operations with Cramer's method (1750) and $2N^3/3 + 2N^2 - 7N/6$ operations with Gaussian elimination, actually due to Newton (1670); by Stirling's approximation, $N! \approx \sqrt{2\pi N}(N/e)^N$. Moreover, elimination is stable, Cramer's method is not. The CPU time in the table assumes 1 TFLOPS.

	Cramer		Elimination	
N	FLOPs	CPU time	FLOPs	CPU time
2	8	8 ps	11	11 ps
3	27	27 ps	33	33 ps
4	124	124 ps	70	70 ps
10	40E6	40 µs	855	0.85 ns
15	2.1E13	21 s	2683	2.7 ns
20	5.1E19	1.6 y	6110	6.1 ns
30	8.2E33	2.6E14 y	19765	20 ns

Elimination does $N = 10\,000$ in 0.67 s. For even faster methods, see Numerical Recipes, Chapter 2. In Matlab, the solution is obtained with the mldivide operator, `x = A\b`. It uses different solvers depending on $\mathbf{A}$; see flowcharts with `doc \` or `doc mldivide`.

Computer generations, Moore's law and quantum computers

Generation	Years	Hardware
1	1940–1956	Vacuum tube
2	1957–1963	Transistor
3	1964–1970	Integrated circuit
4	1971–	Microprocessor

A 5th generation of computers based on artificial intelligence has been discussed since the 1980s, but like 5th generation programming languages the concept is not well defined.

Gordon Moore, co-founder and CEO of Intel, observed in 1965 that the number of transistors in an integrated circuit doubles every year; this became known as **Moore's law**. He revised it in 1975: the number of transistors in a microprocessor doubles every two years. In 2005, he stated the obvious, "it can't continue forever", also because "the size of transistors is approaching the size of atoms which is a fundamental barrier".

We have discussed only **classical computers**. Since the 1990s, **quantum computers** have been pursued, but they are still quite experimental. In our department, Toby Cubitt is Professor of Quantum Information and teaches the undergraduate module COMP0157 Quantum Computation. Moreover, there is the UG module COMP0196 Quantum Information and Communication and the PG module PHAS0070 Quantum Computation and Communication.