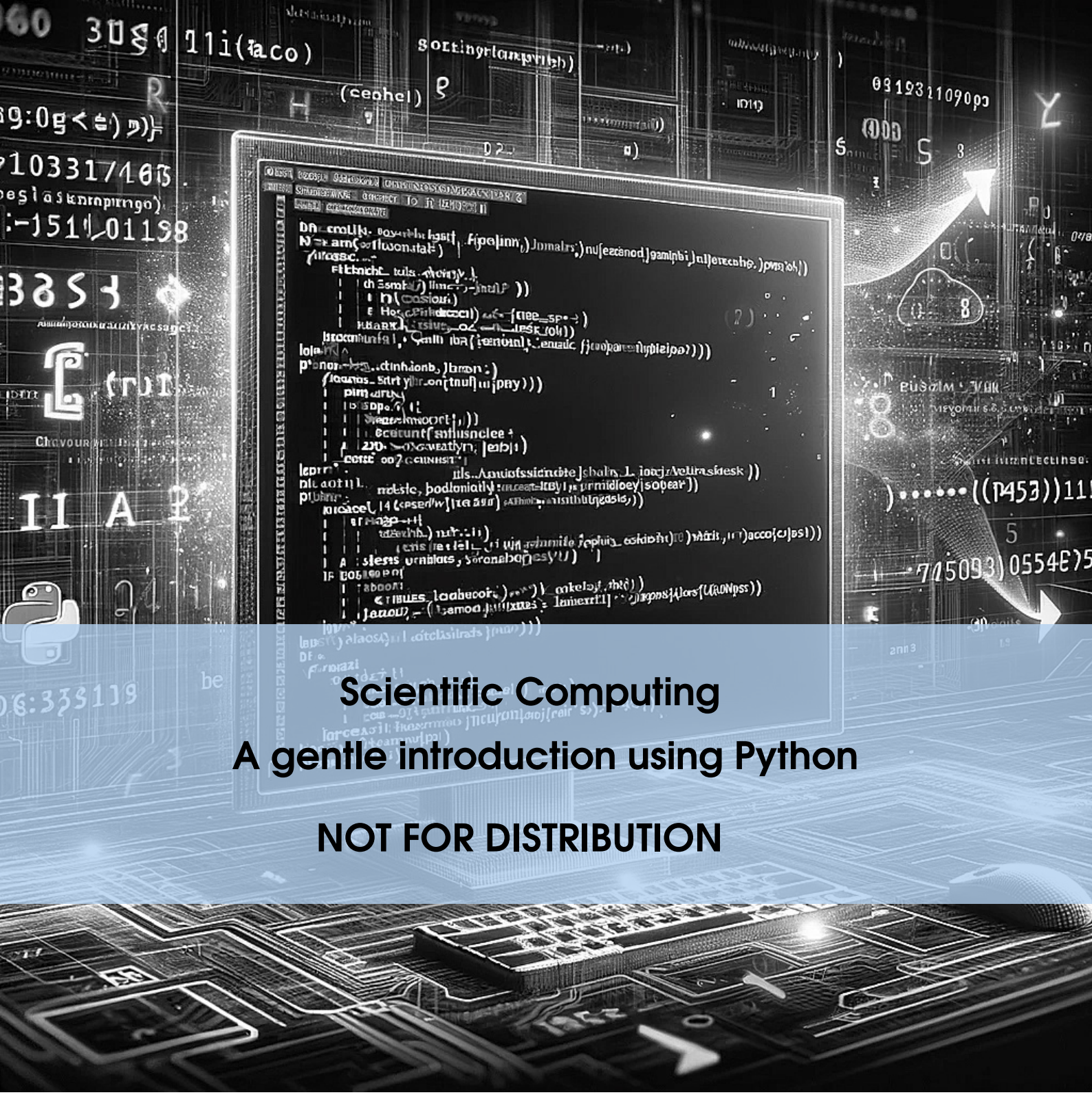


Scientific Computing

A gentle introduction using Python

NOT FOR DISTRIBUTION

Tentative schedule. Updated: January 23, 2024

Lecture room(s): PAB141 / PAB159.

A = assignment

Day	Time	Hours	Total	Notes
Fri 12.1. 2024	11:30 - 12:20	1	1/36	Scheduling & practicals
Fri 19.1. 2024	12:30 - 13:20	1	2/36	Ch 1: Scientific Computing: The Modern Scientific Paradigm
Mon 22.1. 2024	13:30 - 16:30	3	5/36	Ch 2-8: Linux, Python command line, software, A1 posted
Fri 26.1. 2024	no class	no class		
Mon 29.1. 2024	no class	no class		
Fri 2.2. 2024	no class	no class		
Mon 5.2. 2024	13:30 - 16:30	3	8/36	Random numbers. A1 due, A2 posted
Fri 9.2. 2024	12:30 - 13:20	1	9/36	
Mon 12.2. 2024	13:30 - 16:30	3	12/36	
Fri 16.2. 2024	12:30 - 13:20	1	13/36	
READING WK				
Mon 19.2. 2024	Family Day	no class		
Tue 20.2. 2024	11:30 - 14:30	3	16/36	
Wed 21.2. 2024	11:30 - 14:30	3	19/36	
Mon 26.2. 2024	no class	no class		
Fri 1.3. 2024	no class	no class		
Mon 4.3. 2024	13:30 - 16:30	3	22/36	
Fri 8.3. 2024	12:30 - 13:20	1	23/36	
Mon 11.3. 2024	13:30 - 16:30	3	26/36	
Fri 15.3. 2024	12:30 - 13:20	1	27/36	
Mon 18.3. 2024	13:30 - 16:30	3	30/36	
Fri 22.3. 2024	12:30 - 13:20	1	31/36	
Mon 25.3. 2024	13:30 - 16:30	3	34/36	
Fri 29.3. 2024	Good Friday	no class		
Mon 1.4 2024	13:30 - 15:30	2	36/36	
Fri 5.4. 2024	in case needed			
Mon 8.4. 2024	Classes end	no class		
Fri 19.4. 2024	12:30-14:30	Presentations. Projects due.		

Table 1: Tentative schedule for winter 2024. Changes are possible.

Contents

Chapter 1

Scientific Computing: The Modern Scientific Paradigm

Never in the history of mankind has it been possible to produce so many wrong answers so quickly!

Carl-Erik Fröberg, Swedish mathematician, physicist and chemist. Professor of Numerical Analysis, Lund University, Sweden.

IN THE LANDSCAPE OF CONTEMPORARY SCIENTIFIC RESEARCH, the role of scientific computing has become as pivotal as traditional theoretical and experimental methods. Scientific computing, an interdisciplinary field at the confluence of mathematics, computer science, and various scientific disciplines, leverages computational models and simulations to solve complex scientific problems. It has had a transformative impact on research and discovery.

Scientific computing is characterized by the development and application of computational algorithms and simulations to analyze and solve scientific problems. It involves the use of advanced computing capabilities to understand and model systems where theoretical analysis or experimental procedures are challenging or impossible. This field is not just about performing calculations but also about understanding how these calculations can provide insights into scientific phenomena.

While Scientific Computing and High-Performance Computing (HPC) belong to the same class of approaches and the difference between them is somewhat semantic, they do have distinct identities. While both are integral to solving complex scientific problems using computational methods, they focus on different aspects of computation. Scientific Computing is primarily concerned with the development and implementation of computational algorithms to model and solve scientific problems. High-Performance Computing, on the other hand, focuses on optimizing and executing computations at a *large scale*, often involving *supercomputers* and *massive data sets*.

The evolution of Scientific Computing and HPC has been influenced by technological advancements. The rise of machine learning and artificial intelligence, for instance, is reshaping aspects of scientific computing, particularly in data analysis. In HPC, the advent of cloud computing and the increasing focus on energy efficiency are influencing the development of new systems and architectures. Challenges include accuracy and stability of algorithms, and translating complex

physical phenomena into computable models. Particular to HPC, the limitations of Moore's Law, data management, and the complexity of parallel programming are significant hurdles.

The future of scientific computing is intrinsically linked with advancements in machine learning and artificial intelligence. These technologies have the potential to revolutionize how models are developed and simulations are performed, leading to more accurate predictions and deeper insights.

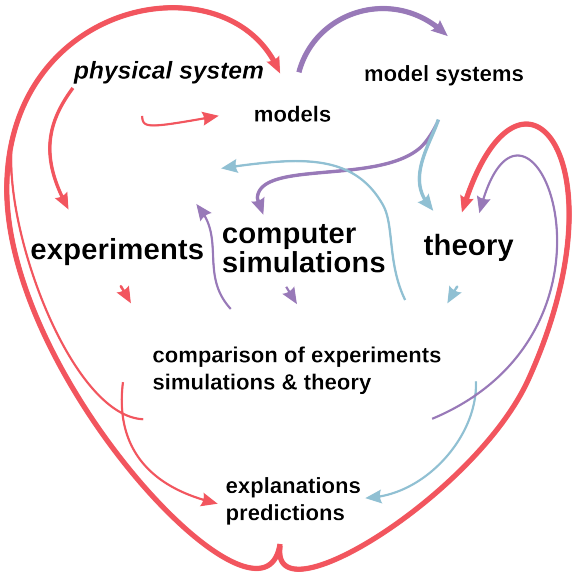


Figure 1.1: The three scientific paradigms, experiments, theory and computer simulations, and their inter-relations. Original figure: Dr. Markus Miettinen. Note that historically, even the position of theory and the uses of mathematics has been contested. In 1830, Auguste Comte, the philosopher behind positivism, stated that it is impossible and destructive to try to study chemistry using mathematics wrote (translated) in his *Cours de philosophie positive*: "Every attempt to employ mathematical methods in the study of chemical questions must be considered profoundly irrational and contrary to the spirit of chemistry.... if mathematical analysis should ever hold a prominent place in chemistry - an aberration which is happily almost impossible - it would occasion a rapid and widespread degeneration of that science." Mathematics is, of course, the universal language of science.

1.1 Methodological Framework

The methodological backbone of scientific computing includes numerical analysis, algorithm development, and high-performance computing. Numerical methods are employed to approximate solutions to mathematical problems that cannot be solved analytically. These methods include, but are not limited to, finite difference methods, finite element analysis, optimization, integration, differentiation and spectral methods. Computational science is inherently interdisciplinary. It not only draws on mathematics and computer science for methods and techniques but also deeply involves domain-specific knowledge from various scientific disciplines. This integration is crucial for accurately modeling and solving real-world problems.

Algorithm development in scientific computing is focused on creating efficient and accurate ways to solve scientific problems. This involves designing algorithms and methods that are not only mathematically sound but also optimized for performance, especially considering the limitations of computational resources, and minimize errors. Particularly important aspects involve differential equations, optimization, statistical analysis, spectral methods, linear algebra, processing and interpreting large data sets, and visualizing the results, topics of later Chapters.

High-performance computing (HPC), another cornerstone, involves supercomputers and parallel processing techniques, including the use of graphics processing units (GPUs). HPC allows for the handling of large datasets and the execution of (very) complex models and simulations, which are integral to fields like climate modeling, materials modeling, astrophysics, and genomics. Scientific computing bridges the gap between theory and experimentation. It allows scientists to test hypotheses *in silico* and gain insights into systems where experimental methods are either too costly, dangerous, or impractical. For instance, in climate science, predictive models are essential for understanding future climate scenarios, which cannot be experimentally replicated. In particle physics, simulations provide a way to explore phenomena that are beyond the reach of current experimental capabilities.

Scientific computing finds applications across a vast array of disciplines. In physics, it enables the simulation of systems ranging from subatomic particles to large-scale structures in the universe. In chemistry and materials science, it aids in understanding molecular interactions and predicting material properties. In biology and medicine, it is used for modeling biological systems, understanding disease mechanisms, and even in drug discovery. Environmental sciences use computational models for weather prediction, climate modeling, and assessing ecological impacts. For the relation between computing, with the traditional paradigms of science, that is, theory and experiments, see Figure 1.1

1.2 Brief History of Scientific Computing & High-Performance Computing

The roots of scientific computing can be traced back to the development of numerical methods by mathematicians such as Euler and Gauss. These early methods laid the groundwork for numerical analysis, a cornerstone of scientific computing. However, a significant leap occurred with Charles Babbage's conception of the Analytical Engine in the 1830s, a mechanical forerunner to the modern computer, which was designed to perform complex calculations.

Modern scientific computing and computational modeling are relatively young fields – this is easy to understand since the first digital (in contrast to mechanical) computers were developed in the late 1930's. The first programmable computer, the Z1, was developed by Konrad Zuse in Germany in the late 1930's. The early developments in programmable digital computers were largely driven by military applications in 1940's and 1950's.

The earliest computers were not reliable or stable, or were built more as calculators for a single purpose. ENIAC (Electronic Numerical Integrator and Computer) that was built in 1946 is often said to be the first real computer. Transistors did not exist at that time and ENIAC (in Philadelphia) had 17,480 vacuum tubes. Its original purpose was to determine trajectories for ballistic missiles, but due to the war ending, it was used to study nuclear reactions. Nuclear warhead and storage related simulations are still one of the main uses of the largest supercomputers. One additional notable aspect of ENIAC is that, unlike modern computers, it was not able to store programs.

ENIAC was followed by MANIAC I (Mathematical Analyzer Numerical Integrator And Computer Model I at Los Alamos) and it was used in the landmark paper 'Equations of State Calculations by Fast Computing Machines' by Metropolis et al. [1] that introduced the Metropolis Monte Carlo method, one of the most used and important algorithms even today. MANIAC was also used in the development of the hydrogen bomb. Generalization of the Metropolis Monte Carlo method is known as the Markov Chain Monte Carlo and it is important in fields such as optimization and machine learning. On pure chemistry side, the earliest computational papers appeared in the 1960's and the term computational chemistry was (probably; hard to trace) first used in 1970's.

In 1964 the CDC 6600 was introduced. It was designed by Seymour Cray at Control Data Corporation (CDC), and was the first machine to be widely considered a supercomputer, outperforming all existing computers at the time. After the CDC, Cray Research became the dominant power in supercomputing. The Cray-1 was introduced in 1976 and it became the fastest computer in the world upon its release, capable of 160 MFLOPS (million floating-point operations per second). Another milestones was Cray-2, the world's fastest supercomputer from 1985 to 1990, achieving speeds of 1.9 GFLOPS (billion, or 10^9, floating-point operations per second).

In the 1993, two remarkable supercomputers deserve special attention, the Thinking Machines CM-5 was introduced in 1993 and it is known for its use in the film "Jurassic Park." The CM-5 was among the world's fastest computers in the early 1990s. Three years later in 1996, the ASCI Red, as a part of the U.S. Department of Energy's Accelerated Strategic Computing Initiative, was the first computer to break the teraflop barrier (1 trillion, or 10^{12}, calculations per second).

The jump from teraflops to petaflops occurred in 2008 when IBM introduced the Roadrunner installed at Los Alamos National Laboratory. It was the first to break the petaflop barrier (1 quadrillion, or 10^{15}, calculations per second). The development since then has been very rapid. In 2016, the Sunway TaihuLight, developed by China's National Research Center of Parallel Computer Engineering & Technology (NRCPC), reached 93 petaflops, and in 2018 IBM Summit, located at Oak Ridge National Laboratory in the United States, achieved a peak performance of 200 petaflops. Currently, the race towards exascale computing (1 exaflop or 1 quintillion or 10^{18} calculations per second) is on, with several projects underway in the United States, China, Japan, and Europe. The U.S. Department of Energy's Aurora and Frontier systems, as well as Japan's Fugaku, are key contenders.

Figure 1.2: The TOP500 list ranks the world's 500 most powerful computers twice a year, June and November. The first list was published in June 1993.

Since its introduction in June 1993, the TOP500 list ranks the 500 most powerful computers in the world. The list is updated twice a year, June and November. In November 2020, China had 212 systems on the list, the USA was the second with 113, and Japan the third with 34 systems. Canada had 12. Continent-wise top countries: Asia: China (overall #1 in terms of the number of systems), North America: USA (overall #2), Europe: Germany (#4), South America: Brazil (#12), Africa: Morocco (#27), Australia: Australia (#23). However, that a given country has a large number of systems on the list does not say anything about the *access* to those resources. For an individual researcher what matters is the ease of access and that is handled differently in each of the countries. In addition, countries and groups of countries have allocation modes that give preference to certain projects, that is, in the same style as grants. For example, in Canada the Digital Research Alliance of Canada Canada has the normal allocation that is very easily accessible to researchers, and a separate National Resource Allocation Competition which requires a formal application process for priority allocations to some systems and resources. In Europe, there is PRACE (Partnership for Advanced Computing in Europe). It has a pan-European competition for resource allocation.

In addition to the supercomputing centers, *distributed computing* in its many forms, such as cloud computing and citizen science, has lead to very interesting projects including so-called *citizen science* in which individual people donate the idle time on their computer for research purposes. This includes open projects such as Folding@home, Foldit, SETI@home, DreamLab, and BOINC (the Berkeley Open Infrastructure for Network Computing).

One aspect of the discussion is **Moore's law**, the observation that the number of transistors in integrated circuits doubles every 24 months. Despite the name, this is rather an observation than a law, Figure 1.4. Moore's law is due to Gordon Moore, co-founder of Intel, who made the prediction

Figure 1.3: IBM Blue Gene P. The Blue Gene family of computers were among the most powerful systems until to about 2015. Since then, IBM appears to have stopped the development. The Blue Gene/P that is in the figure was a 2nd generation Blue Gene computer. In terms of numbers, the processors on the Blue Gene/P were of IBM's PowerPC type. A variant of the PowerPC chip was used in Macs before Apple switched to Intel processors in 2005. The Blue Gene/P used 850 GHz PowerPC450 processors. Figure: Wikipedia, CC BY-SA 2.0; supercomputer installation at the Argonne Leadership Angela Yang Computing Facility located in the Argonne National Laboratory, Lemont, Illinois, USA

in 1965. Moore originally stated that the number of transistors on a microchip would double every year while the costs were halved. In 1975, Moore revised this prediction to a doubling approximately every two years. This observation became widely accepted in the semiconductor industry and has been used as a guiding principle for the development of microchips and technology in general. Importantly, Moore's Law is not a law of physics, but rather just an empirical observation and projection of a historical trend. It has, however, been remarkably accurate over several decades, leading to exponential increases in computing power, reduction in cost, and miniaturization of electronic devices. However, it is important to note that it is not expected to hold indefinitely, as physical limitations of semiconductor technology, such as heat dissipation and quantum effects in extremely small structures, pose challenges to the continuation of this trend.

1.3 The Role of Computational Physics and Chemistry

Historically, scientific computing and computational modeling have deep roots in physics and chemistry, but independent if one talks about physics, chemistry, or some other field, the development can be divided roughly in the theoretical developments in the field, developments of numerical methods and algorithms, and hardware. The development of numerical methods cannot be over-emphasized as it is critical to be able to model and simulate larger systems and for longer times. This will become clear as we discuss various methods and how they scale in terms of the number of degrees of freedom.

Currently, computational approaches such as hybrid methods that combine quantum mechanical and classical molecular dynamics and coarse-grained methods have gained a lot of popularity. Methods that combine different time and/or length scales can in general be called multiscale methods and this will be discussed in more detail later. In addition, methods that go under the term machine learning are perhaps the most rapidly growing family of methods and they are applied in almost

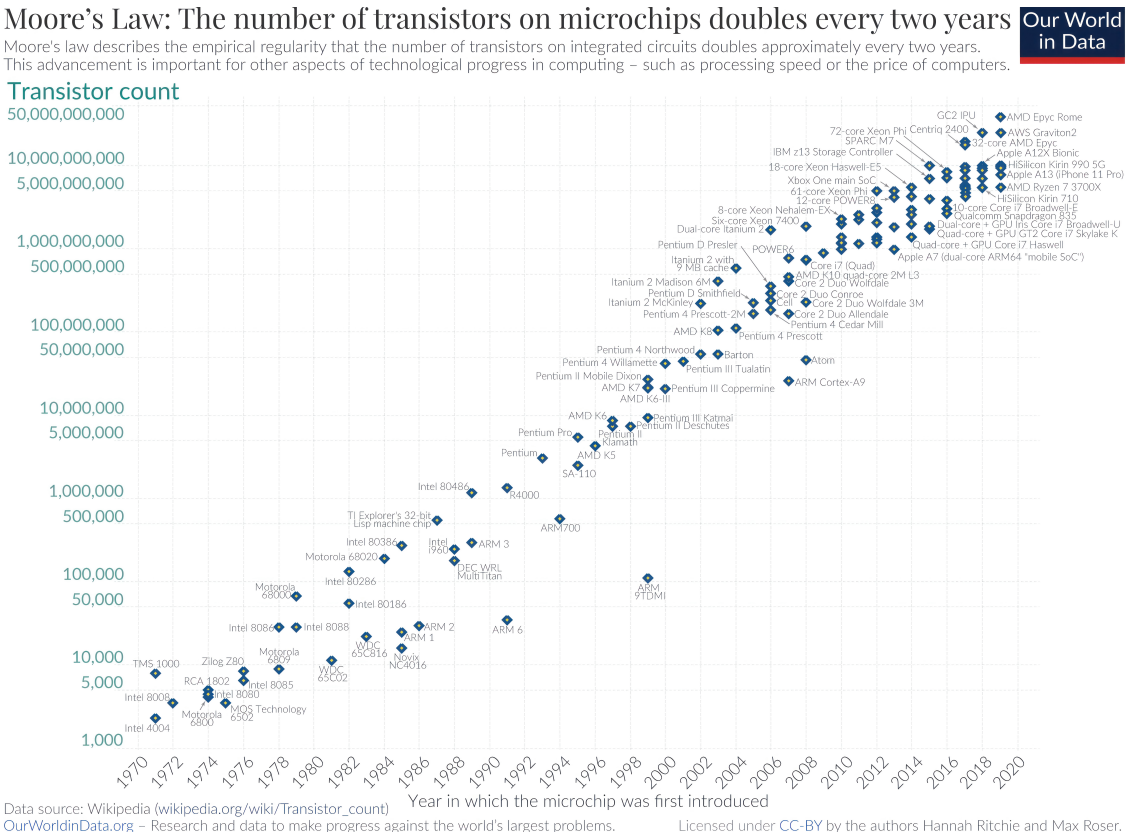


Figure 1.4: Moore's law. From Wikipedia, Creative Commons license.

all imaginable fields from medicine to digital humanities and computational chemistry. Machine learning is not a simulation method as such, but rather a class of methods that allows classification (clustering), analyses and model building using large data sets. Machine learning methods rely heavily on statistics, linear algebra and stochastic methods such as Markov processes. We will discuss such methods later.

1.3.1 Computational microscope

The term *computational microscope* was probably first introduced by Walker and Mezey in 1995 in their article "A new computational microscope for molecules: High resolution MEDLA images of taxol and HIV-1 protease, using additive electron density fragmentation principles and fuzzy set methods" [2]. This work combined imagining and density functional theory (DFT) calculations. The term was then re-introduced in the context of MD simulations by Lee et al. in their 2009 article "Discovery Through the Computational Microscope" [4]. The analogy is a very good one: molecular simulations can mimic behavior of systems in the atomic scale and they allow one to see, analyze and track the atoms. Lee et al. [4] conclude their article by writing

> *"Overall, simulations can cast light on the basic architectural principles of molecules that explain fundamental cellular mechanics and might eventually direct the design of proteins with desired mechanical properties. Most importantly, beyond the understanding gained regarding the molecular mechanisms of force-bearing proteins, these examples serve to demonstrate that incessant advancements of MD methodology bring about discoveries stemming from simulation rather than from experiment. Molecular modeling, while useful as a means to complement many experimental methodologies,*

> *is rapidly becoming a tool for making accurate predictions and, thereby, discoveries that stand on their own. In other words, it is becoming a computational microscope. "*

The above quote summarizes the situation very well. Independent if one is a theorist, experimentalist or a computational scientist, it is imperative to have a reasonable understanding what the other methods of research can and cannot do.

It is important to keep in mind that while we are dealing with *empirical sciences* and any prediction has to be proven by experiments, theory and computation provide predictions and explanations for observations in a manner that is often impossible for experimentation alone. In addition, experiments have their own problems and to get a signal out of any system, one must poke it, that is, one has to perturb the system to get a response. Simulations don't have such a limitation but can even be used to study what the effects of probes or perturbations are on a given system. All in all, experiments, theory and simulations all have their advantages and problems and one should not consider any of them to be more superior than the others, see Figure 1.1. With the rise of big data, one could even extend the methods of research to *data-intensive analysis*.

1.3.2 Nobel Prizes to Computational Chemistry: 1998 and 2013

To date, the Nobel Prize is Chemistry has been awarded twice to computational chemists - thus far no Nobel Prizes have been granted directly to computational physics. The first computational chemistry Nobel was 1998, when Walter Kohn (see also his biographical notes), "for his development of the density-functional theory", and John A. Pople (see also his biographical notes), "for his development of computational methods in quantum chemistry", were the recipients. Kohn was born in Austria, escaped the nazis to Canada and received his BSc in Applied Mathematics at the University of Toronto. He moved to study physics at Harvard for his PhD. His PhD supervisor was Julian Schwinger, 1965 Nobel Laureate in Physics. He was at the University of California in Santa Barbara at the time of his Prize. Kohn passed away in 2016 at the age of 93. Sir John Pople received his PhD in Mathematics from Cambridge. He was at Northwestern University at the time of the Nobel Prize. He passed away in 2004 at the age of 78. Pople is one of the authors of the first version of the software suite Gaussian. Pople's PhD supervisor was Sir John Lennard-Jones, the Lennard-Jones potential used in almost all classical molecular dynamics simulations is named after him.

The second Nobel Prize to computational chemistry came in 2013 when it was awarded to Martin Karplus, Michael Levitt and Arieh Warshel "for the development of multiscale models for complex chemical systems." The press release of the Royal Swedish Academy of Science stated the importance of computational chemistry very clearly:

> *"Today the computer is just as important a tool for chemists as the test tube. Simulations are so realistic that they predict the outcome of traditional experiments."*
>
> -Pressmeddelande (Press Release), Kungliga Vetenskapsakademien (The Royal Academy of Science)

Like Kohn, Martin Karplus was born in Vienna, Austria and moved to the USA to escape the nazis. He did his PhD in chemistry at Caltech with the two-time Nobel Laureate Linus Pauling. He is the originator of the CHARMM molecular simulation package. Karplus was at Université de Strasbourg and Harvard at the time of the Prize. Arieh Warshel was postdoc (1970) with Karplus' at Harvard in Chemical Physics. He went to the Weizmann Institute in Isreal for a faculty job but moved later to the University of Southern California after Weizmann didn't grant him tenure.

Michael Levitt was born in Pretoria, South Africa and he did his PhD in Biophysics at Cambridge. At the time of the Prize he was at Stanford where he is a professor of Structural Biology. As a side note on tenure and Nobel Prizes, Nobel winners that were not given tenure include at least Tom Sargent (Economics 2011; University of Pennsylvania didn't give tenure), and Lars Onsager (Chemistry 1968; dismissed/let go by both John's Hopkins and Brown universities).

1.3.3 Other notable prizes in computational chemistry & physics

Since we got into the Nobel Prizes above, let's list a few other significant recognitions. Probably the best know is the Berni J. Alder CECAM Prize, the acronym CECAM comes from Centre Européen de Calcul Atomique et Moléculaire and it is currently headquartered at the École polytechnique fédérale de Lausanne (EPFL) in Switzerland. CECAM is a pan-European organization and on the practical side it is well-known for its workshops, summer schools and conferences. The Prize, named after Berni Alder one of the pioneers of molecular simulations, is granted every three years. The prize was established in 1999 and has thus far been granted to

2022: Kurt Kremer, Max Planck Institute for Polymer Research, Mainz, Germany, for his 'exceptional contributions to the microscopic simulation of polymers, setting the standards for using numerical simulation to probe conceptual issues in polymer physics and establishing it as an indispensable tool in materials science.'

2019: Sauro Succi, Italian Institute of Technology, Center for Life Nanosciences at La Sapienza in Rome, for his pioneering work in lattice-Boltzmann (LB) simulations.

2016: David M Ceperley, Department of Physics, University of Illinois at Urbana-Champaign and Eberhard Gross, Max Planck Institute of Microstructure Physics, Halle for "fundamental ground-breaking contributions to the modern field of electronic structure calculations".

2013: Herman J.C. Berendsen and Jean-Pierre Hansen, Groningen and Cambridge for "outstanding contributions to the developments in molecular dynamics and related simulation methods"

2010: Roberto Car (Princeton) and Michele Parrinello (ETH Zürich) for "their invention and development of an ingenious method that, by unifying approaches based on quantum mechanics and classical dynamics, allows computer experiments to uncover otherwise inaccessible aspects of physical and biological sciences"

2007: Daan Frenkel, Cambridge.

2004: Mike Klein, Temple University. The committee wrote: "Mike Klein's leadership has been crucial in the development of a variety of computational tools such as constant-temperature Molecular Dynamics, Quantum simulations (specifically path-integral simulations), extended-Lagrangian methods and multiple-timestep Molecular Dynamics"

2001: Kurt Binder, University of Mainz "for pioneering the development of the Monte Carlo method as a quantitative tool in Statistical Physics and for catalyzing its application in many areas of physical research"

1999: Giovanni Ciccotti, University of Rome La Sapienza for " pioneering contributions to molecular dynamics"

1.4 Some Pioneers and Innovators

The list below is by no means even close to complete. The hope is that it servers as a starting point to having a better view of the developments and timelines regarding computers, computing and algorithms.

1.4.1 Charles Babbage: 'The father of the computer'

Charles Babbage (December 26, 1791 – October 18, 1871), was an English mathematician, philosopher, inventor, and mechanical engineer who played a pivotal role in laying the conceptual and engineering foundations for the modern computer. His importance to computing is highlighted by several key contributions: Invention of the *Difference Engine* – in the 1820s, Babbage conceived the idea of a mechanical device that could perform mathematical calculations. This device, known as the Difference Engine, was designed to compute and print mathematical tables. While he never completed a full-scale version of this machine, his design was groundbreaking and demonstrated the feasibility of using machinery for calculations. The next one was the design of the *Analytical Engine*: Babbage's most significant contribution to computing in the 1830s. This machine was the first to conceptually include all the basic elements of a modern computer, including a control unit, memory, and the ability to be programmed using punched cards. The Analytical Engine was a leap forward from the Difference Engine, capable of more complex and general-purpose computations. He also contributed to the introduction of programming concepts: The Analytical Engine's design introduced the concept of *programmability*. Ada Lovelace, a mathematician and collaborator of Babbage, is credited with writing the first algorithm intended to be processed by a machine, making her arguably the first computer programmer. Her notes on the Analytical Engine included what is essentially the first description of software.

1.4.2 Ada Lovelace: The first programmer

Babbage referred to Ada Lovelace (née Byron; December 10, 1815 – November 27, 1852) as the "Enchantress of Numbers." She wrote, what many consider, to be the first computer program —- an algorithm designed to be processed by a machine (Babbage's Analytical Engine). Lovelace foresaw the potential of computers beyond mere number crunching. She envisioned that they could be used to manipulate symbols according to rules and that they could be applied to various fields like music and art, effectively predicting the modern multi-purpose computer.

1.4.3 John von Neumann: A Foundational Pillar

John von Neumann's (December 28, 1903 – February 8, 1957) contributions to scientific computing are foundational. His work on the architecture of electronic computers, particularly the concept of stored-program computers, laid the groundwork for modern computing. Von Neumann was also instrumental in developing numerical methods for hydrodynamic calculations, and his work in the early days of the *Manhattan Project* set the stage for the use of computers in large-scale scientific simulations.

1.4.4 Konrad Zuse: The world's first fully functional, programmable computer

Zuse (June 22, 1910 – December 18, 1995)) created the Z3 in 1941, which is widely considered the world's first fully functional, programmable computer. The Z3 was a significant advancement because it was the first machine that could automatically execute complex calculations using a series of instructions (programs). The Z3 used telephone switching equipment and was based on electromechanical technology. He also created the first high-level programming language - Plankalkül. He developed the Plankalkül (Plan Calculus) around 1945, although it was not widely known until its publication in 1972. Plankalkül is considered the first high-level programming language, a precursor to the modern programming languages used today. Zuse's earlier work includes the Z1, completed in 1938, which was the first freely programmable computer using binary floating-point arithmetic and Boolean logic. Although it was mechanical and not fully reliable, the

Z1 laid the groundwork for his later, more successful designs.

1.4.5 Grace Hopper: Developer of the first compiler

Grace Hopper (née Murray; December 9, 1906 – January 1, 1992), a pioneering computer scientist and U.S. Navy rear admiral, made instrumental contributions to the development of modern computer science. She Hopper was a key person in the development of the first compiler, a program that translates instructions written in a programming language into machine code. This work was a foundational step in moving away from programming in raw machine code and laid the groundwork for modern programming languages. She also played a key role in the creation of the Common Business-Oriented Language (COBOL), one of the first high-level programming languages. COBOL was designed to be readable by business people and remains in use today, particularly in financial and administrative systems. At the more conceptial level, she contributed to the development of machine-independent programming languages, which allow software to run on different types of computers. This concept is a cornerstone of modern software development.

1.4.6 Alan Turing: The Genesis of Computational Theory

Alan Turing (June 23, 1912 – June 7, 1954), best known for his role in breaking the Enigma code during World War II, also laid the theoretical foundations for computer science. His concept of the Turing machine is a cornerstone of the theory of computation. Turing's work in the field of artificial intelligence and his development of the Turing Test have had profound implications in computational science and beyond.

1.4.7 Gene Amdahl: Amdahl's Law and Mainframe Computing

Gene Amdahl (November 16, 1922 – November 10, 2015)) is renowned for formulating Amdahl's Law, a fundamental principle in parallel computing. Amdahl's Law provides an insight into the potential speedup of a program as a function of the number of processors and the proportion of the program that can be parallelized. Amdahl also made significant contributions to the architecture of IBM mainframes and later founded Amdahl Corporation, a key player in the mainframe market.

1.4.8 Seymour Cray: The Father of Supercomputing

Seymour Cray (September 28, 1925 – October 5, 1996) is often hailed as the father of supercomputing. In the 1960s and 1970s, Cray's designs for the CDC 6600 and CDC 7600 set new standards for computer performance. Later, as the founder of Cray Research, he continued to revolutionize HPC with systems like the Cray-1, Cray-2, and Cray X-MP, which were among the fastest computers in the world at their time.

1.4.9 Gene Golub: Numerical Linear Algebra and Matrix Computations

Gene Golub (February 29, 1932 – November 16, 2007), was a giant in the field of numerical linear algebra and matrix computations, areas critical to scientific computing. His work on algorithms for matrix decompositions and iterative methods has been widely adopted in various scientific and engineering applications. Golub's contributions have had a lasting impact on the efficiency and accuracy of numerical methods in computational science. His also the author (with Charles F. van Loan) of the classic book *Matrix Computations* [5]. Another classic for anyone needing numerical methods is *Numerical Recipes* by Press et al. [3] (available in C/C++, Fortran and Fortran 90).

1.4.10 Frances E. Allen: Pioneer in Compiler Design and Optimization

Frances E. Allen (August 4, 1932 – August 4, 2020), the first female IBM Fellow and Turing Award winner, made seminal contributions to the theory and practice of compiler optimization, which are critical for efficient HPC. Her work on program optimization and parallelization has had a profound impact on compiler design.

1.4.11 Gordon Bell: Architect of Parallel Computing

Gordon Bell (b. August 19, 1934) is known for his contributions to the design and development of parallel computing systems. His work at Digital Equipment Corporation (DEC) on the VAX series of computers was instrumental in advancing computer architecture. The Gordon Bell Prize, awarded for outstanding achievements in HPC, is named in his honor.

1.4.12 Margaret Hamilton: Software Engineering in Space Exploration

Margaret Hamilton's (née Heafield; b. August 17, 1936) contributions to software engineering, particularly in the context of the Apollo space missions, have been groundbreaking. Her work in developing the onboard flight software for the Apollo missions demonstrated the critical role of software reliability and robustness in scientific computing, particularly in high-stakes environments.

1.4.13 Ken Kennedy: Compiler Optimization and Parallel Computing

Ken Kennedy's (August 12, 1945 – February 7, 2007) work in the field of compiler optimization and parallel computing has been foundational. He was instrumental in developing techniques for optimizing compilers, which are crucial for high-performance computing. Kennedy's research on parallel programming and his advocacy for high-level programming languages in HPC have had a significant impact on the field.

1.4.14 Jack Dongarra: Numerical Algorithms and Software for HPC

Jack Dongarra (b. July 18, 1950) is a key figure in the development of software and algorithms for high-performance computing. He has contributed to projects like LINPACK and LAPACK, essential libraries for numerical computing. Dongarra's work on the TOP500 project, which ranks the world's fastest supercomputers, has been influential in tracking and analyzing trends in HPC.

1.4.15 John L. Hennessy and David A. Patterson: Pioneers of Computer Architecture

John L. Hennessy (b. September 22, 1952)) and David A. Patterson (b. November 16, 1947) are influential figures in the field of computer architecture. Their work on Reduced Instruction Set Computing (RISC) laid the foundation for efficient, high-performance processor design. Hennessy and Patterson co-authored "Computer Architecture: A Quantitative Approach," a seminal text in the field, and their contributions have had a lasting impact on how modern processors are designed and built.

1.4.16 Leslie Greengard and Vladimir Rokhlin: Fast Multipole Methods

Leslie Greengard (b. 1957) and Vladimir Rokhlin's (b. August 4, 1952) development of the Fast Multipole Method (FMM) revolutionized the field of computational physics. FMM provided

an efficient way to compute long-range forces in N-body simulations, drastically reducing the computational complexity of these calculations. Their work has had profound implications in fields ranging from astrophysics to molecular dynamics.

1.4.17 Linus Torvalds

Linus Torvalds (b. December 28, 1969) created Linux, the OS that dominates the HPC computing in the world. He released Linux as free and open-source software, available for anyone to use, modify, and distribute. In 2005, he created Git, a distributed version control system that is widely used for source code management in software development.

1.5 Programming Languages

High-Performance Computing (HPC) is a domain that demands extreme computational power and efficiency to solve complex scientific, engineering, and data analysis problems. The choice and evolution of programming languages in HPC have been pivotal in harnessing and optimizing the capabilities of supercomputers and parallel computing architectures.

A key challenge in HPC programming is balancing performance with productivity. While languages like C and Fortran offer high performance, they often require more development effort and expertise. Higher-level languages like Python have gained popularity due to their ease of use and readability, despite potential performance trade-offs. Efforts to bridge this gap include integrating high-level languages with performance-oriented libraries or using JIT (Just-In-Time) compilation techniques. Languages like Julia, which aim to combine the performance of traditional HPC languages with the ease of use of high-level languages, are also gaining attention. Furthermore, the integration of AI and machine learning in HPC is likely to influence the development of new programming paradigms and languages.

Below, we discuss some aspects of programming languages most from the point of view of HPC. Python will be discussed more extensively in Chapter 8.

1.5.1 Historical Evolution of HPC Programming Languages

The early days of HPC were dominated by languages designed specifically for scientific computing. The introduction of FORTRAN (FORmula TRANslation) in the 1950s at IBM by the team lead by John Backus marked a significant milestone. As the first high-level programming language, FORTRAN was designed for numerical computation and scientific computing, offering a balance between abstraction and hardware-level control. Fortran is still being used in HPC, the latest version is Fortran 2018.

In the 1970s and 1980s, languages like C (developed at the Bell labs by Dennis Ritchie in the early 1970's) and later C++ (developed by Bjarne Stroustrup and released in 1985) began to gain popularity in HPC due to their performance and lower-level control over hardware. C++, with its object-oriented features, provided a more modern approach to large-scale software engineering in HPC applications.

1.5.2 Parallelism

The shift from single-core to multi-core processors and the advent of distributed computing systems necessitated languages that could efficiently handle parallelism. This led to the development

of parallel programming models and extensions, such as MPI (Message Passing Interface) for distributed memory systems and OpenMP (Open Multi-Processing) for shared memory systems. These models are often used in conjunction with traditional languages like C and Fortran.

1.5.3 The Rise of Domain-Specific Languages (DSLs)

To address the specific needs of certain HPC applications, Domain-Specific Languages began to emerge. These languages, tailored for particular problem domains, allow for more efficient expression of algorithms and can lead to significant performance optimizations. Examples include SQL for database queries and Halide for image processing.

1.5.4 The Challenge of Heterogeneous Computing

The rise of heterogeneous computing architectures, combining traditional CPUs with GPUs (Graphics Processing Units) and other accelerators, introduced new challenges and opportunities in HPC programming. NVIDIA's CUDA (Compute Unified Device Architecture) became a game-changer for GPU programming, enabling dramatic performance improvements for parallel computations. Similarly, OpenCL (Open Computing Language) provided a framework for writing programs that execute across heterogeneous platforms. We will discuss GPU computing in more detail below.

1.6 GPU Computing: Revolutionizing High-Performance Computing

The evolution of Graphics Processing Units (GPUs) from specialized graphics rendering devices to powerful general-purpose processors has been one of the most significant developments in HPC. GPU computing, also known as GPGPU (General-Purpose computing on Graphics Processing Units), has transformed computational science, enabling unprecedented performance in a range of applications.

GPUs were originally designed to accelerate the rendering of 3D graphics. The pivotal moment in the evolution of GPUs came with the realization that their highly parallel structure could be harnessed for general-purpose computing. This was particularly attractive for tasks involving large-scale, repetitive numerical calculations, which are common in scientific computing.

The introduction of NVIDIA's CUDA (Compute Unified Device Architecture) in 2007 was a watershed moment in GPU computing. CUDA provided a development environment that allowed programmers to use C/C++ for coding on GPUs. This made parallel computing more accessible and opened up a wide range of applications beyond graphics processing – GPUs have turned out to be very versatile and are used in a wide range of applications, from molecular dynamics and fluid dynamics to machine learning and artificial intelligence. Other common applications include GPUs have accelerated climate and weather modeling, the analysis of large genomic datasets, and financial data. Furthermore, GPUs offer superior performance per watt compared to CPUs, making them fairly energy-efficient for high-performance computing, and their inherently parallel nature makes them highly scalable for large-scale computations (*scalability*). Some of the challenges include programming complexity, that is, efficiently programming GPUs requires specialized knowledge, which can be a barrier for some researchers, and memory limitations in the case for very larga data sets especially in machine learning applications.

Over the years, GPU architecture has evolved to enhance its suitability for general-purpose computing. Key developments include:

- **Increased Core Count:** Modern GPUs contain thousands of cores, allowing for massive

parallelism. For example, the consumer level NVIDIA RTX 4080 (introduced in late 2022) has 9,728 CUDA cores and the GTX 1080 from 2016 has 2,560 cores.

- **Unified Memory Access:** Improvements in memory architecture have facilitated faster data transfer between the CPU and GPU, reducing bottlenecks.
- **Enhanced Double-Precision Performance:** Earlier GPUs were optimized for single-precision calculations, but advancements have significantly improved their double-precision capabilities, crucial for scientific computing.

1.7 Practical: How HPC is done

Most HPC is done using computer clusters (such as those of the Digital Research Alliance of Canada). A cluster is – as the name suggests – a cluster of individual computers that are *networked* to each other. These individual computers are called *nodes*. Inter-node networking is one of the most critical parts of any cluster since it largely determines *latency* – the nodes need to share information between each other and the speed at which information is transferred can easily be the rate-limiting step for a given computing task. This is the reason why there are many different networking solutions such as Infiniband (>200Gb/s). The properties of each node are determined by the individual computer that makes up the node. If all computers, that is nodes, in a cluster are identical, then the cluster is called *homogeneous* (in contrast to *heterogeneous*). An individual user doesn't access the nodes directly but through a textitlogin node, also called *head node*.

Figure 1.5 shows a typical computing cluster with 16 individual nodes, see also Figure 1.6. There is also another computing paradigm called SMP, Symmetric Multi-Processing. In this case there are multiple CPUs but they all share the same memory (in a cluster all the nodes have individual independent memory). SMP clusters are typically used for special-purpose simulations and require very large amount memory.

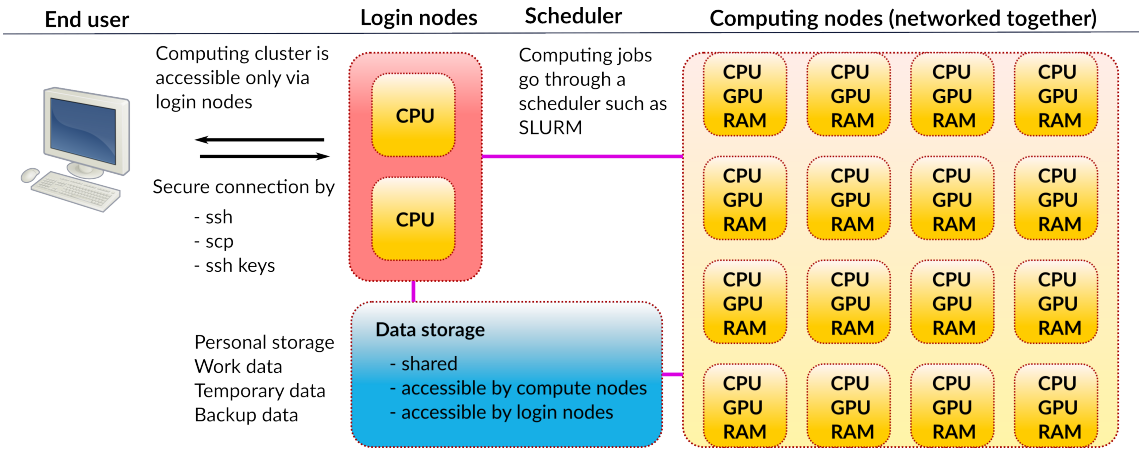


Figure 1.5: Basic organization of a typical HPC computing cluster. The end user accesses the cluster through login nodes only, the individual computing clusters are not directly accessible. For connecting to login nodes, the user uses `ssh` and `scp` for transferring data from and to personal computer. To enhance security (and also convenience of connecting), ssh keys are typically used - one should use ssh keys for any remote connections if possible. The computing nodes have their own CPU, memory and storage, possibly also GPU(s). Computing within a node is always faster since otherwise data has to be transferred between the nodes and that causes latency. Computing jobs must be started using a scheduler, `SLURM` is a commonly used one. Data storage is shared. HPC systems typically have different levels of storage for different types (work vs long term storage) data.

Important: If a task fits inside one node (by its memory and processing requirements), its

performance is always much better when it is kept inside one node rather than distributing it on an equal number of cores (or equal resources) across two or mode nodes.

When starting computing jobs, an individual user cannot simply fire them away and sit back. Rather, all the computing jobs are sent to the cluster using a *scheduler*. There are several types of schedulers but nowadays the SLURM scheduler is the most common one. The name SLURM comes from Simple Linux Utility for Resource Management.

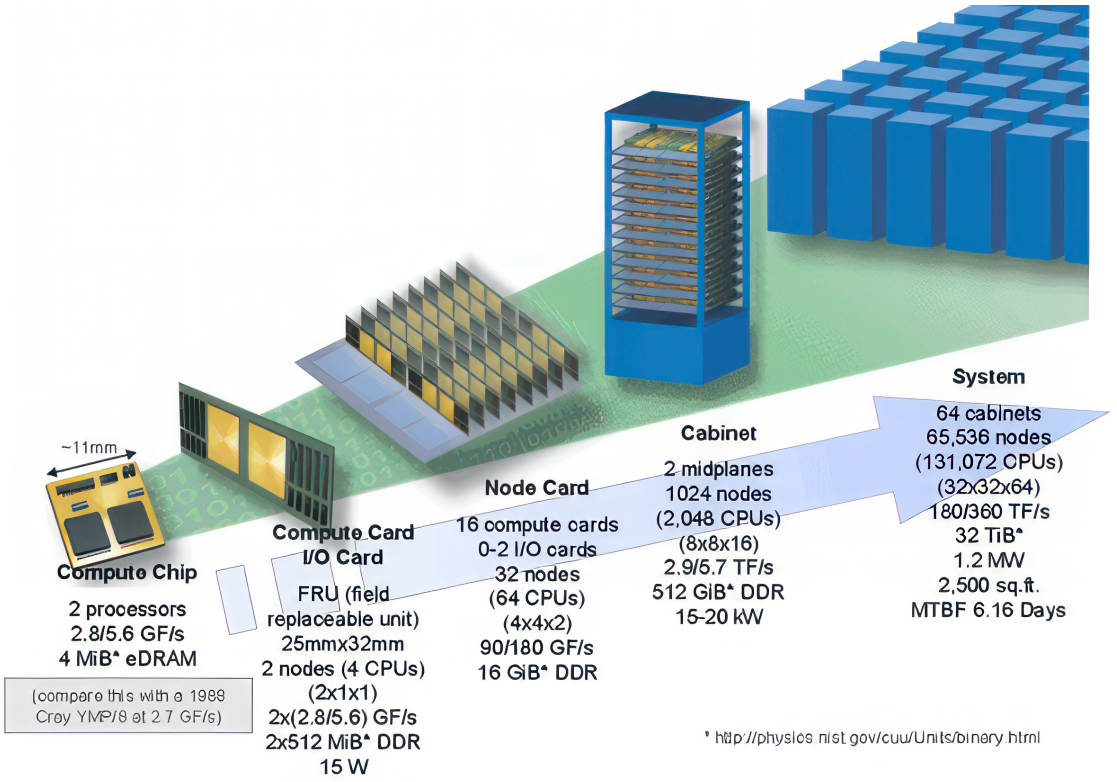


Figure 1.6: The figure shows the processing units used in IBM's second generation Blue Gene P supercomputer. The basic compute chip has two processors. This makes a node. At the next level, the compute card bundles two compute chips. At the higher level, the node card has 8 nodes or 16 compute cards. The node cards are put in a cabinet that has 1,024 nodes and the full system has 64 such cabinets. Figure: Wikipedia, public domain.

1.8 Practical: Terminology – CPU, Core, Thread, GPU, kernel, hyper-thread

CPU - Central Processing Unit. Also called the processor or the processing unit. They represent numbers in binary form, that is, zeros and ones. Physically, CPUs are on an integrated circuit (IC), or what is often called the chip (metal oxide semiconductor). The chip is not synonymous to CPU since it typically has also other components on it, in particular *cache memory*. Some computers employ a multi-core processor, which is a single chip or "socket" containing two or more CPUs called "cores"." We will use this concept directly when compiling and running simulations.

Core. Core refers to the CPU, a physical component or the processing unit. It is a more general term than CPU and it is also used in the case digital signal processors (DSP) and others. In the case of a single core processor, the commands, or instructions to be more precise, are executed *sequentially* one after another. A new task cannot start before the previous one has finished. In the case of a multi-core processor, a single integrated circuit (or chip) contains several physical CPUs

(cores) that can execute several processes simultaneously in parallel. This sometimes called the MIMD, or Multiple Instruction, Multiple Data paradigm. The term multi-CPU is not the same as multi-core. In the case of a multi-CPU, the processing units are not on the same integrated circuit as in the case with multicore. More or less all current processors are multicore - if you don't know how many cores you have, we can easily check that and make use of the cores to speed up simulations.

Thread. It is a unit of execution. In contrast to a core which is a physical component, a thread is a virtual component.

Multithreading. Ability to run many threads simultaneously.

GPU - Graphics Processing Unit. As the name suggests, GPUs are processing units specially designed for handling graphics. They are different from CPUs in that they (modern day ones) are massively parallel, one GPU unit can contains thousands of processing units (the consumer level NVIDIA RTX2080 has 2,944 cores and AMD Pro Vega II has 4,096 cores) while CPU has typically a small number cores between 2 and 32. For this reason, GPUs have been optimized for parallel processing as compared to CPUs that have their origin in serial processing. In addition, they were designed to handle floating point numbers very efficiently. Programming-vice, there are significant differences between programs that are designed for GPUs vs CPUs since the parallelism is the paradigm of GPU computing; *vectorization* and *multi-threading* are the two main differences. GPUs are now part of any typical HPC cluster for tasks such as molecular simulations and machine learning, GPUs are programmed with C++-like languages, the general purpose OpenCL and NVIDIA's CUDA. The latter is the most common one. The latest AMD GPUs use a language called HIP which is a direct rival to CUDA.

GPUs and software: Software such as Gromacs, NAMD, Amber, BigDFT, CP2K, HOOMD-Blue, LAMMPS, and so on all have GPU acceleration, as do machine learning software such as Tensorflow and PyTorch, and image processing software such as Blender and Adobe's software, and the list keeps growing very rapidly.

GPUs on a personal computer: It is naturally possible to use the GPU a personal computer or a laptop to run even rather demanding simulations. The basic requirement is a discrete GPU. If one wants to use CUDA acceleration, an NVIDIA card is required. OpenCL runs on both NVIDIA and AMD cards. As for efficiency and support, CUDA is currently by far the most used one and this includes software such as Gromacs, NAMD, Amber, Tensorflow, PyTorch etc.

The basic idea of vectorization: Older supercomputers (such as CDC and Cray-YMP) up to the early 1990's were predominantly *vector computers*. GPUs also operate similarly to vector computers. So the question is: What is a vector computer? Let's start from *serial computers*, that is essentially what our usual PCs are. Without going into the details of computer architecture, serial means that one instruction is performed at a time; data is retrieved and then an operation is performed on it. In vector processing, the processor uses one instruction to operate on a whole vector at the same time (=parallel operation; SIMD, Single Instruction Multiple Data), see Figure 1.7. Processors that are designed to operate in parallel on multiple data are called vector processors in contrast to scalar processors that use a single data element at a time (a pure serial processing is SISD, Single Instruction Single Data).

1.9 Practical: Linux – *The* operating system in HPC

The area where Linux adaption is the largest is HPC where it is overwhelmingly the most used operating system. Based on 2017 and later data, Linux has 100% share of the TOP500 computers. We will discuss the Linux operating system in more detail and introduce its basic commands in

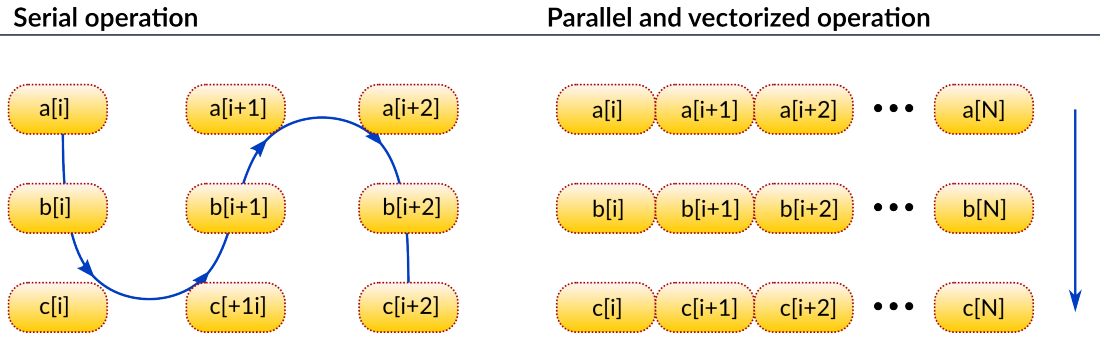


Figure 1.7: Left: In serial processing, one operation is performed at a time and one then moves on in a serial fashion. In vector processing operations are performed at the same time. One operates on a pipeline.¶

future lectures. This absolute dominance of Linux has some consequences, namely,

- Knowledge of Linux is vital since 100% of world's TOP500 computers run on Linux. Connections to computing clusters are only via a specific login or head node that also runs Linux. Your own personal computer can be Windows, Mac or Linux. Each of them have some advantages and can be used to connect to a HPC system via command line.
- Lots of the modern analysis tools work best on Linux. Since with HPC one often deals with very large amounts of data, processing can take anything from a few minutes to days. One of the niceties is that with Linux-based systems it is very easy to put such analysis processes on the background while using the computer for other tasks.

1.10 Practical: Hardware, software, user interface

We now discuss some very basic issues regarding hardware, software and user interface. Figure 1.8 summarize the relations. The operating system (OS) is an interface between the hardware and applications the user interface (UI) is the part of the OS that connects the user with the OS, that is, the UI is a virtual surface that connects the user to the computer. The two main types of user interfaces are the familiar Graphical User Interface (GUI). Another one is the Command Line Interface (CLI). While a GUI is handy for certain tasks and requires no knowledge of special commands like CLI does, it is not suitable for certain tasks espcially when efficency is needed. This is why CLI is needed. Even installation of simulation software becomes very cumbersome, if not impossible, without CLI.

Kernel is the heart of the operating system. Wikipedia description of kernel, 'The kernel is a computer program at the core of a computer's operating system with complete control over everything in the system', describes it very well. The above defines what kernel means in terms of operating systems and it should not be confused with *compute kernel*, the term is used in GPGPU computing. A compute kernel is the computationally intensive part that typically involves the inner loops.

1.11 Practical: Terminology – Double precision vs single precision

- **Single precision:** 32 bit. The data type is called single. A floating point number is represented using 32 bits: 1 bit for the sign, 8 bits for the exponent and 23 bits for the mantissa. This means that the range of numbers is $2^{-126} - 2^{127}$ or $\approx 1.18 \times 10^{-38} - 3.40 \times 10^{38}$ (single precision).

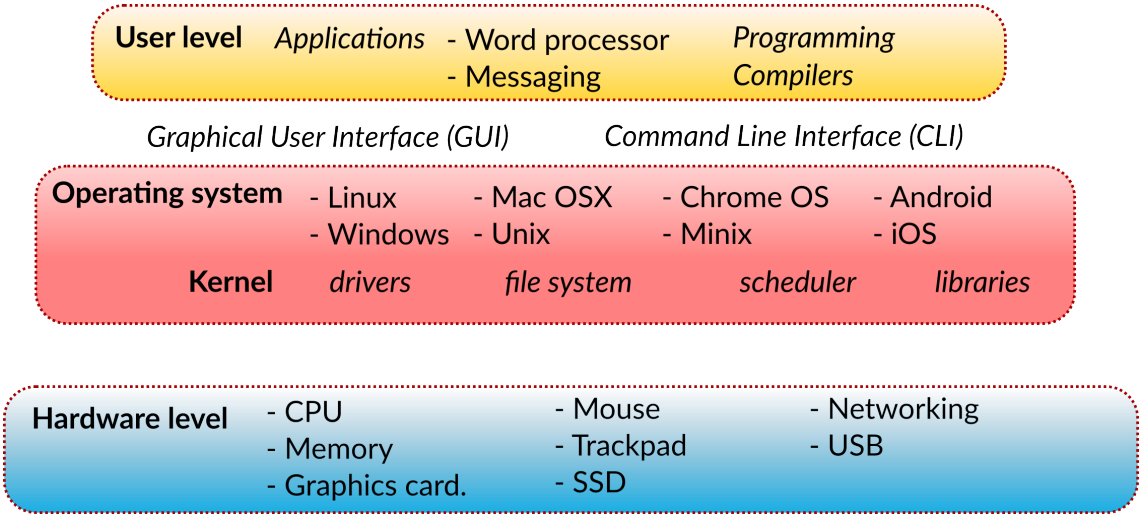


Figure 1.8: Basic relations between the hardware, operating system and the user level. The user interacts with the operating system via a GUI or CLI. The operating system then communicates with the hardware.

- **Double precision:** 64 bit. Data type is called double. A floating point number is represented using 64 bits: 1 bit for the sign, 11 bits for the exponent and 52 bits for the mantissa. The range of numbers is $2^{-1022} - 2^{1023}$.
- **Special cases:** overflow, not-a-number, subnormal number.

Most simulation software allow for compilation in either single or double precision.

1.12 Practical: Good practices

The list below repeats some of the matters discussed earlier, but it is well worth going through them again.

- Keep notes, that is have a lab book. That saves a lot of time and reduces the number of errors. Document the procedures and commands (and they can sometimes be somewhat tricky and finding them quickly without proper notes may be difficult or/and lead to errors. Lab book will help you to avoid mistakes and help you to speed up your work tremendously. One good way is to use something like a github document. That is easy to maintain and it is available from any computer. Other reasonable ways: Google docs, OneDrive, Dropbox, etc.
- Always remember to back up you critical files! There are great tools for doing that including GitHub, Zenodo and such.
- When running simulations, ensure that you will not fill your computer (estimate the amount of data that will be generated and is needed for analysis).
- Check for viruses
- Always, always, always visualize
- Always, always, always verify your simulations & system setup against known results from theory, other simulations and experiments.
- Automatize: Let the computer do the work. Use shell scripts, python, or comparable to let the computer to do the repetitive work.

1.13 Practical: Dangers for beginners in scientific computing

It may sound counter-intuitive, but the main problem of any computational modeling is the user, the computer will do what it is told to do and nothing else. If the instructions given by the users or programmer are wrong, or incomplete, the problem is not the computer but the person or persons who used/programmed the computer. To emphasize this fact, in one of our papers, "*The good, the bad and the user in soft matter simulations*" we wrote [6]:

> *It may sound absurd and somewhat provoking, but available simulation software (i.e., not home grown programs) becoming very user friendly and easy to obtain can almost be seen as a negative development!*

Not that long ago all simulations were based on house-written programs. Such programs had obvious limitations: Unlike a lot of the modern open-source software, those codes were not available to others, they were not maintained for longer periods of time, and they were often difficult to use for all but the one person who wrote code. It was also difficult to improve their performance due to very limited number (often just one) of developers. Modern codes are generally well maintained, have long-term stability, extensive error checking due to large number of users and they offer excellent performance that is impossible reach for any individual developer or even a small group.

Similarly to experiments, it is easy train a person to use the software and even produce data using the built-in analysis methods. That is, however, a precarious path. It is *absolutely imperative* to have a very strong background in the application field and its underlying theories and methods to be able to produce something meaningful. One should never use a software package as a black box!

1.14 Practical: Which programming language should I use/learn?

Depends on the needs, the level of knowledge and (projected) application(s). For example, Python is a general purpose language that can be used for many tasks and it is particularly useful for analyzing and visualizing data independent if the data is from simulations, experiments, online or some database. Python is very quick to learn and it has amazing amount of libraries and routines for almost any imaginable task, and the community is large and very supportive. Python is an *interpreted language*. Python is a language that has become a must learn independent of field and application, and it is particularly important in machine learning. While Python has become the *de facto* standard in data analysis and machine learning, its performance in terms of the demands of HPC makes it unsuitable for HPC applications, and languages such as C/C++, Fortran and CUDA need to be used.

C and C++ have a different nature than Python. They have to be *compiled* and they have a much more rigidly defined structure. C/C++ codes are great for writing high-performance simulation codes and, for example in the field of molecular simulations, Gromacs, LAMMPS and NAMD are written in C/C++. As for syntax, C/C++ and Python have lots of similarities.

Fortran is an older programming language and it was originally designed to be very efficient for numerical calculations. The name Fortran comes from *For*mula *Tran*slation. Although there has been a shift to C/C++, many HPC codes such as CP2K, Orca and Gaussian are written in Fortran.

Graphics processing units (GPU) are used increasingly in high performance computing. They use languages such as CUDA for NVIDIA GPUs, HIP for AMD GPUs and OpenCL that is GPU agnostic. In high performance computing CUDA is the dominant one while HIP is starting to gain ground. Syntactically, they have a very high resemblance to C/C++.

As for computers, the Linux operating system is overwhelmingly dominant in the world of high performance computing. As for programming, it can be done in any of the common environments, that is, Windows, macOS and Linux, or even Android or Chrome.

1.15 External links:

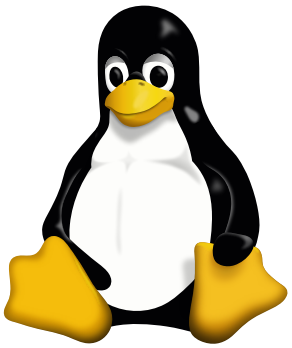
- Simulating Physics with Computers Richard P. Feynman
- Metropolis, Monte Carlo and the MANIAC
- Computing and the Manhattan Project
- A Trilogy on Errors in the History of Computing
- Commodore 64: Unboxing in 1982
- The lost 1984 video: Young Steve Jobs introduces the Macintosh
- Rampant software errors may undermine scientific results
- Bill Gates introduces Windows 3 and NT, 1991
- Tim Berners-Lee: How This Guy Invented the World Wide Web 30 Years Ago
- Deep Fake – should you believe what you see? This is possible on a normal consumer-grade GPU!
- AI at the Oscars: De-Aging and Other 'Digital Human' Special Effects
- Beauty in Science: Biophysicist Klaus Schulten & his Computational Microscope
- Three ways researchers can avoid common programming bugs
- Computational science: ...Error
- A Scientist's Nightmare: Software Problem Leads to Five Retractions
- Scientists Make Mistakes. I Made a Big One.
- IEEE Standard for Floating Point Numbers

1.16 Problems

Problem 1.1 Computing now and in the past: Search the literature/web and compare the computational power of a 1990's supercomputer, your laptop, an Android phone, gaming workstation. Be as specific as possible and provide references.

Chapter 2

First steps with Linux



The Dec 29, 2003 entry for Linux in the Urban Dictionary reads:

linux n. An OS that is awesome for geeky programmers. For everyone else, it is much too difficult to install and use.

Linux will always have a small installed base on desktop computers, because the only way it can become mainstream is if it loses the difficulty of installation and use. The only way to attain that is to completely change what linux is.

by truth teller December 29, 2003

Now it is time to discuss some of the Linux basics. This description does not aim to be complete, but rather practical in terms of pointing out the difference between Linux, Windows and macOS. The differences appear to - based on experience - a lot of confusion so we start from there. Unlike Windows and macOS, Linux is open-source software, thousands volunteer programmers participate in the development. As already discussed above, it is the absolutely dominant operating system in all HPC everywhere in the world.

In general Windows, and macOS, and Linux are three most prominent and widely used operating systems, each with its unique characteristics, history, and user base. For personal computers, Windows is the dominating OS with about 80 % share, followed my macOS with about 15 % and Linux with about 3 %. Below, we will briefly describe these operating systems, highlighting their key features and differences.

Linux, often referred to as the poster child of open-source software, is a Unix-like operating system kernel that forms the basis for numerous distributions (or distros) such as Ubuntu, Debian, and CentOS. It was created by the Finnish MSc student Linus Torvalds at the University of Helsinki in 1991 and has since grown into a robust and versatile ecosystem. Linux is known for its stability, security, and flexibility, making it a preferred choice for servers and a growing (albeit slowly) number of desktop users. One of its defining features is the open-source nature, which encourages

collaboration and customization. Linux supports a wide range of file systems, with `ext4` being one of the most commonly used. It is highly customizable, allowing users to tailor their environments to specific needs. Additionally, Linux has a rich command-line interface (CLI) that appeals to power users and system administrators. As discussed before, all of the world's top 500 supercomputers run on Linux.

Windows, developed by Microsoft, is the most widely used operating systems globally. It has a long history, dating back to the early 1980s, and has evolved through numerous iterations, with Windows 11 being the latest version. Windows is known for its graphical user interface (GUI) and extensive software compatibility. It dominates the desktop and laptop market, making it a go-to choice for many users, especially in corporate environments. The Windows NT (including Windows 11) family uses the NTFS (New Technology File System) as its default file system, providing advanced features like access control, encryption, and compression. Windows also supports a wide range of hardware and software, making it a versatile choice for various tasks.

macOS, developed by Apple Inc., is the operating system used exclusively on Apple's Macintosh computers. It has a reputation for its sleek and elegant user interface design, as well as its focus on user experience. macOS is built on a Unix-based (Free BSD) foundation, which provides a stable and secure environment. It has seen several iterations, with macOS Sonoma (14.2.1) being the most recent version in late 2023. The Apple File System (APFS) replaced HFS+ as the default file system, offering features like snapshot support, encryption, and optimization for solid-state drives (SSDs). macOS is known for its seamless integration with other Apple products and services, such as the iPhone, iPad, and iCloud.

Linux, Windows, and macOS represent a diverse spectrum of operating systems, each with its own strengths and appeal to different user groups. The choice of operating system often depends on individual preferences, specific use cases, and the hardware at hand, making each of them relevant and influential in the world of computing.

Before discussing Linux/Unix further, let's list some of the advantages and disadvantages.

2.1 General: Some advantages and disadvantages of Linux

Advantages of Linux

- Some of the advantages of Linux include:
- Excellent performance and low resource requirements (lightweight)
- Functionality
- Open source: One of the most significant advantages of Linux is that it is open source, which means anyone can view, modify, and distribute its source code freely. This fosters a vibrant community of developers and encourages innovation. Additionally, Linux distributions are typically free to download and use, making it cost-effective for individuals and organizations.
- Customizability, flexibility and portability: Linux offers a high degree of customization. Users can choose from a wide variety of Linux distributions (distros), desktop environments, and software packages, tailoring their system to specific needs. This flexibility makes Linux suitable for a broad range of use cases, from servers to embedded systems.
- Stability: Linux is renowned for its stability and reliability. It can run for extended periods without needing a reboot, making it a favored choice for servers and critical systems. This stability reduces downtime and maintenance costs.
- Security: Linux is inherently more secure than some other operating systems due to its Unix-like architecture. Its robust permission system, limited user privileges, and a lower

susceptibility to malware and viruses contribute to its strong security features. Frequent security updates from the community further enhance its safety
- Connects nicely with Android cell phones for file transfer (appears easier than on Windows).

Disadvantages of Linux

- Learning Curve: Linux can have a steep learning curve for users who are unfamiliar with Unix-like systems. The command-line interface (CLI) is powerful but may be intimidating for beginners. However, most Linux distributions offer graphical interfaces.
- Limited Commercial Software Support: While Linux offers many alternatives for popular commercial software, some applications are not available or may have limited support. This can be a drawback for users who rely on specific proprietary software packages; some software such as Microsoft Office and Adobe Photoshop are not available (although Progressive Web Applications are making a difference).
- Sometimes there is a lack of drivers for the very latest hardware: Although Linux supports a wide range of hardware, some manufacturers do not provide Linux drivers or support. This can lead to compatibility issues with certain devices, particularly graphics cards and Wi-Fi adapters.
- Fragmentation: The diversity of Linux distributions can lead to fragmentation. Compatibility and software availability may vary between distros, creating challenges for developers and users who want consistency across different Linux systems.
- Some software such as Microsoft Office and Adobe Photoshop are not available
- On laptops, batteries drain faster although there have been some very good recent developments

2.2 Linux: A brief overview

The still remaining most common misconception is that Linux is only for coders and geeks, and that is cumbersome to install and use. While in the early days of Linux, and to a degree even in the early 2000's, that was true, at this time that is no longer the case. Modern Linux distributions are very easy and quick to install, easy to maintain and extremely stable. For normal users, there is no need to use the feared command line or terminal.

The persistent myth of Linux being difficult to install and almost impossible to use for anyone but geeky programmers is very unfortunate. It is, as a matter fact, much faster and easier to install than, say, Windows or macOS (anyone who claims otherwise has not installed those two from scratch; in addition, the Windows and macOS updates tend to be very large: macOS update from Catalina to Big Sur is of the *meager* size of 12.18 GB that needs to be downloaded over the net, and even the Catalina security update is 2.84 GB. The user experience is also very similar to the other two main operating systems, and, importantly for the everyday user: there is no need for the terminal or the command line. Everything can be run in a menu-driven manner as with Windows and macOS. In our case that is different since the terminal window is the power user's best friend.

2.3 Who uses Linux / where is Linux used?

As discussed above, the desktop market share of Linux is not enormous, around 2-3% according to various sources. Although the real number may be somewhat higher since Linux is typically installed afterwards and thus difficult to track. For example, my computers have all shipped with Windows or macOS pre-installed yet with the exception of one, they have all been converted to either pure Linux or dual boot.

In High Performance Computing (HPC), Linux has absolute dominance: 100% - every single computer - on the TOP500 list of world's fastest computers uses Linux. As a practical consequence of that, one needs to know some command line and Linux to be able to use those resources. In addition, other operating systems such Android and Chrome OS are based on Linux.

- Entertainment: Almost all of the major film studios use Linux. This includes names such as Pixar and DreamWorks - as early as 1997 the movie Titanic was produced on Linux servers.
- Web Servers and Hosting: Linux is widely used as the foundation for web servers and hosting services. Popular web server software like Apache, Nginx, and Lighttpd run seamlessly on Linux, providing a robust platform for hosting websites, web applications, and content management systems (CMS) like WordPress.
- Cloud Computing and Virtualization: Linux is a core component of many cloud platforms and virtualization solutions, such as Amazon Web Services (AWS), Google Cloud Platform (GCP), and Kubernetes. Its scalability and resource management capabilities make it ideal for running virtual machines and containers.
- Scientific Research and Supercomputing: Linux is prevalent in scientific research and supercomputing environments. It powers high-performance computing clusters and supercomputers worldwide, enabling simulations, data analysis, and scientific discoveries in fields like astrophysics, biology, chemistry, and climate modeling. SpaceX and the Jet Propulsion laboratory use Linux.
- Data Centers and Server Farms: Large-scale data centers and server farms often rely on Linux due to its stability, scalability, and low resource footprint. Linux distributions like CentOS and Ubuntu Server are favored choices for building and managing server infrastructure.
- Embedded Systems and IoT: Linux is used extensively in embedded systems and Internet of Things (IoT) devices. Its adaptability allows it to run on a wide range of hardware, making it suitable for applications like smart appliances, industrial automation, and IoT gateways. This includes TV's (including manufacturers such as Samsung and LG), entertainment systems such as Kodi, vehicle information systems (manufacturers such as BMW) and such, and the list goes on.
- Mobile Devices and Android: Android, the most popular mobile operating system, is built on a Linux kernel. Linux's robustness and versatility are evident in the wide array of smartphones, tablets, and other mobile devices powered by Android.
- Network Infrastructure and Routing: Linux-based routers and network appliances play a vital role in managing internet traffic and providing network services. Linux's networking capabilities, including packet filtering and routing, are crucial for efficient data transmission.
- Cybersecurity and Ethical Hacking: Linux is a preferred choice for cybersecurity professionals and ethical hackers due to its security features and extensive toolkit for vulnerability assessment, penetration testing, and network monitoring.

2.4 Brief history of Linux

Linux is an independent operating system but it has its roots in Unix. Unix originated from the Bell Laboratories in 1969.

Linux was created by Linus Torvalds who was an MSc student at University of Helsinki (Finland) at the time. Torvalds was born Dec 28, 1969 in Helsinki, Finland. He is named after Linus Pauling or/and Linus of Peanuts. The first version of Linux was published by Torvalds in 1991 when he uploaded his operating system to FUNET, the Finnish University and Research Network ftp server for anyone to download.

2.4.1 Linus Torvalds: The creator of Linux

Linus Torvalds (Figure 2.1) is a Finnish-American computer programmer and software engineer who is best known for creating the Linux kernel, the core component of the Linux operating system. He was born on December 28, 1969, in Helsinki, Finland.

Figure 2.1: Linus Torvalds.

Linus Torvalds started working on the Linux kernel in 1991 when he was a computer science student at the University of Helsinki. Frustrated with the limitations of the Minix operating system, he began to develop his own Unix-like kernel as a hobby project. He posted a message on the Usenet newsgroup seeking feedback and collaboration, which marked the inception of the Linux project. Although the name of the OS is “Linux”, according to the story Torvalds himself did not give the name Linux. Rather, it was Ari Lemmke, an IT administrator at FUNET who gave the name "Linux" without asking Torvalds first; Torvalds had uploaded his new OS to the FUNET server using the name "Freax", but Lemmke didn’t like the name and changed it to Linux. The Linux kernel was released on August 25, 1991. His Masters Thesis at the University of Helsinki is entitled "Linux: A Portable Operating System". It may be the most influential Masters thesis in any field ever. In addition to Linux, he is also the creator of Git. He was awarded the Millenium Technology Prize in 2012; the Millenium Technology Prize is worth 1 million Euros. Previous recipients in the field of computers include Sir Tim Berners-Lee, the inventor of the World Wide Web. Torvalds has been named as one of the world’s most influential people by the Time Magazine several times (here a link to an article in the Time Magazine from 2004) Torvalds still has the ultimate control as what is included in the Linux kernel. He is also known to express himself quite sharply in the Linux discussion groups - no need to interpret whether something is positive or negative.

What sets Linus Torvalds apart is his decision to release the Linux kernel under an open-source license, specifically the GNU General Public License (GPL). This choice allowed others to freely view, modify, and distribute the source code, fostering a collaborative and global community of developers who contributed to the project. This open development model led to the rapid growth and success of Linux, and has been adopted for a very broad range of different projects in fields ranging from technology to humanities.

Linus Torvalds continued to oversee the Linux kernel development for many years, providing guidance and maintaining the project’s overall direction. While he is the original creator of the Linux kernel, he emphasizes that it is a collective effort and credits the thousands of contributors who have played a significant role in its development.

2.4.2 Additional external references to talks and videos by/of Linus Torvalds.

- The Code: Story of Linux documentary (subtitles in 12 languages)
- Computer History Museum: The Origins of Linux – Linus Torvalds (2008)
- Linus Torvalds talking with Aalto University (Helsinki) students on 23 October, 2012
- Linus Torvalds on his insults: respect should be earned (includes coarse language)
- The infamous NVIDIA talk: Linus Torvalds: Nvidia, F**k You!
- The above incident became infamous/famous but the full talk (at Aalto University, Helsinki) is very interesting: Aalto Talk with Linus Torvalds [Full-length]

2.5 From the beginnings to the current state of Linux, and Linux distributions

After its release, Linux began to gain traction, thanks in part to the efforts of Linux distributions like Slackware and Red Hat. These distributions bundled the Linux kernel with essential software packages, making it easier for users to install and use Linux on their computers. However, installation during the early days was time consuming and sometimes quite cumbersome and frustrating.

As Linux gained popularity toward the early 2000's , major technology companies recognized its potential. IBM, Oracle, and others started investing in Linux development and offering support for Linux-based solutions. This corporate support helped Linux gain credibility in the enterprise world. The 2000's marked a period of significant growth for Linux, and the broader open-source software movement. The GNU/Linux operating system, combining the Linux kernel with GNU software, became a standard platform for servers and supercomputers.

While Linux had been primarily associated with server environments, efforts like the GNOME and KDE desktop environments, as well as user-friendly distributions like Ubuntu (first release on Oct 16, 2004), made Linux increasingly viable for desktop and laptop users. Linux-powered Chromebooks have also gained popularity in education.

The 2010's marked what one call a mobile revolution as Android, an open-source mobile operating system built on the Linux kernel, was released in Sep 23, 2008 and emerged as a dominant force in the smartphone market. Equally importantly, the rise of containerization technologies like Docker and Kubernetes, which rely heavily on Linux, revolutionized software deployment and management. Linux is also the foundation of many cloud computing platforms, including AWS, Google Cloud, and Azure.

At this time (early 2024), there are a lot of distributions to choose from. Here, we will use Ubuntu-based Linux. While most of the instructions in this document are distribution agnostic, it may be easier to install one of the Ubuntu-based distributions as will be discussed below; the most apparent difference to the end user is that the different distributions have different package management systems and thus the precise commands as how something is done when we do command line installation may be different. For an advanced user that is not an issue. As for releases, there are also rolling releases and standard releases. Here, we use standard Long-Term Support (LTS) Ubuntu release (22.04 LTS to be precise, but most of the material has been tested with earlier releases including 20.04 LTS and 18.04 LTS); LTS releases are supported for five years, and are released every two years. The numbering, such 22.04 LTS, refers to year 2022, April, and Long-Term Support.

2.5.1 External links:

- Wikipedia: Comparison of Linux distributions

Chapter 3

Command line basics and installation of some necessary software

The city's central computer told you? R2D2, you know better than to trust a strange computer

C3PO.

In this course we will use some GUI driven software, software that is run through a web browser (Jupyter Lab/Notebook) and, what may come as a new experience, we will rely heavily on the command line terminal or console.

In general, there are two kinds of user interfaces:

- Graphical users interfaces (GUI)
- Command line interfaces (CLI)

The former one is quite obvious. GUIs are generally menu-driven and the operations consist of clicking and dragging-and-dropping. In Linux/Unix, the GUI is sometimes called X-windows and there are many different options. While convenient and intuitive for lots of tasks, the drag-and-drop operations and clicking are not enough for computing and we need a command line interface.

The command line provides the fundamental way to interact with the system:

- To perform tasks that are impossible using a GUI
- Create, access, copy, move and edit files and directories
- Execute commands
- And of particular interest here: To run computer simulations.

In large-scale HPC systems (such as the computing systems of the Digital Research Alliance of Canada), only command line is provided. It is used for:

- Scripting and creating new commands and aliases
- The command line is also fundamental for system administration

If you have used the basic terminal application in Windows, the Windows terminal is based on MS-DOS, whereas in macOS and Linux/Unix the terminal uses Linux/Unix commands. Importantly, the MS-DOS and Linux/Unix commands are not compatible. The MS-DOS based terminal is not useful for computation as it doesn't provide the necessary functionalities and tools. This means that

in the case you use Windows, you must install WSL (Windows subsystem for Linux), Linux in Virtual Machine or dual boot Windows/Linux, see Figure 3.1.

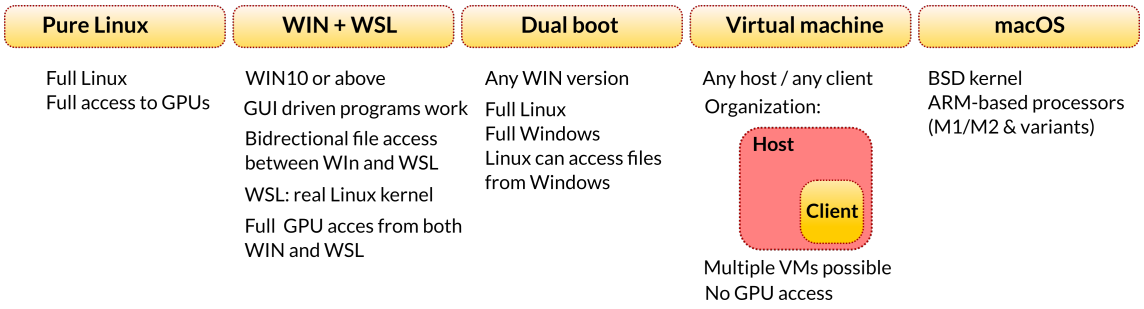


Figure 3.1: For the purposes of this course, one needs to have Linux, WSL or macOS. There are several different approaches. The figure shows different options and some of the relevant features. WSL is the fastest to set up. The virtual machine approach is versatile: any host-client combination can be used and one can have multiple clients. The downside is that enough memory is required since RAM is shared between the operating systems when two of them are run simultaneously. In addition, the virtual machine side has no access to possible GPUs. Since any host-client combination is possible, one can also have a Linux host and Windows client.

3.1 What do we need to do?

Important:

The responsibility is yours. When you install any software the responsibility is entirely yours. It is important to you back up your data and essential files. Doing that ensures that in case something goes wrong, you will be able to recover your previous work and software. While it is relatively rare that something goes wrong, one should take the necessary precautions. Recovering a crippled computer may be very hard or impossible. Remember also to run virus checks regularily.

For the purposes of this course, a CLI and some tools are needed. It is not possible to run simulation software from a GUI and a CLI is essential for efficient operations of many other tasks as well. Independent of the operating system:

- We need to a Linux/Unix command line terminal. It will be used extensively throughout the course.
- The CLI will be either Linux (for pure Linux as well as for WSL) or Mac's terminal (technically the macOS one is based on Free BSD (Unix) but that doesn't make any practical difference here).
- We need to learn the basics of how to use a CLI.

3.1.1 If you have a computer with Linux installed

All the basic tools are in place, just check that your computer is updated and that it has a C/C++ compiler. On Ubuntu-based systems, run

```
sudo apt update && sudo apt upgrade
```

to update, and for compilers,

```
sudo apt install gcc g++
```

3.1.2 If you have a Mac

Mac users already have a command line terminal. It can be found in

```
Finder -> Applications -> Utilities -> Terminal
```

That same location has also `XQuartz` that will be needed to run GUI based applications.

`Xcode` is required. Download `Xcode` from the official Apple Developer site and install. `Xcode` provides the compilers, libraries and other development tools. In addition, a package manager such as `Homebrew` or `MacPorts` is needed. Most of the examples are with `Homebrew`.

Check the Homebrew web site for details/options/possible updates, but in brief, this is how it is installed: After installing `Xcode`, open a terminal window and give the command

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

3.1.3 If you have a Windows computer

The following assumes Windows 10 on higher. The first decision to choose between Windows+WSL, dual boot and virtual machine (VM), see Figure 3.1. Installing WSL with Windows is by far the easiest and fastest choice, and WSL is supported by Microsoft. WSL is the recommended choice, but instructions for pure Linux, dual boot and virtual machine work as well.

Windows + WSL

This is an excellent option, the recommended choice here, as it is quick and easy, and provides full native Linux kernel. WSL also supports GPUs (if present) meaning that GPU computing is possible. Many popular software packages such as `Tensorflow` and `pyTorch` for machine learning, and Gromacs, NAMD and Amber for molecular simulation packages can use GPUs for simulations. GPUs can provide a significant speedup compared to CPUs.

Linux applications with GUI can also be run, for example, if desired, a wen browser can be run from the WSL (Linux) side. In addition, WSL provides bi-directional file access. That is, the Windows side of the computer can access files in WSL and vice versa.

To proceed, first check your Windows version and ensure that it higher than Windows 10 version 2004 (Build 19041 and higher). Windows 11 is, obviously, good. Second, with WSL one has choice between different Linux distributions (including Ubuntu, OpenSUSE, Kali, Debian, Arch Linux, etc.). Here, Ubuntu is used, but any other will work as well.

Follow the official instructions from Microsoft at How to install Linux on Windows with WSL. Note that the instructions mention also WSL1 and WSL2. If you have never had WSL before on your computer, then you will get the latest by default. Assuming that no WSL is present, the installation procedure is very simple: Open `Powershell` in administrator mode and give the command

```
wsl --install
```

This installs an Ubuntu based Linux distribution. The installation takes a few minutes. After the installation has completed you will have to reboot the computer. If the above doesn't work (this

may be the case with some Windows 10 setups), use the following to install the default Ubuntu Linux

```
wsl --install -d Ubuntu
```

The installation takes a few of minutes and at the end of it, the system will ask you for a new username and password. Then, for possible `wsl` updates, in `Powershell` run the command

```
wsl --update
```

With this, your `wsl` should now be fully up-to-date with the default Linux version, Ubuntu, installed. There are also many other options than Ubuntu, to see them (in `Powershell`) type

```
wsl --list --online
```

but we won't discuss them further.

Now, open an Ubuntu window (from your Windows menu), and in the command line Linux terminal run the commands

```
sudo apt update && sudo apt upgrade
```

for Linux updates. After this, your basic Ubuntu installation is ready. Note that if you have Nvidia GPUs, the graphics drivers are automatically installed.

For this course, install C/C++ compilers

```
sudo apt install gcc g++
```

Note: It is possible to install several separate Linux distributions with WSL. If you wish to do that, first check the available distributions by giving the following command in `Powershell`

```
wsl --list --online
```

and then

If you have an old WSL and want to upgrade

If the command

```
wsl --list
```

is not found, then it is highly likely that your `wsl` is old. You can check that using `Powershell`. Open it it in `admin` mode and type

```
wslconfig /l
```

If you want to uninstall it – if you do, back up all your data first as uninstalling will remove *everything* in your WSL partition – do the following

```
wslconfig /u <Distro name>
```

Try Linux without installing

If you are curious about installing Linux either as a dual boot or the only operating system, you can try it first using *Live USB* without installing. This allows you try it out and see if it works on your computer. When doing this, the operating system (Linux) runs from the USB stick making this a safe way to try out a Linux distribution. In addition, even if you already know that you want to install Linux, this step is important since it allows you to see that Wifi, bluetooth and other devices work properly, and to check if you like the user interface (there are many options). Here are the basic steps (the suggestions below are Ubuntu or Ubuntu-based for ease):

1. To make a Live USB you need a USB pen. 8 GB is recommended. If you want to try several distributions/flavors at the same time, just have 2 or more pens ready. Note that the USB pen needs to be erased (and formatted) in the process so make sure that you copy all the important files to another USB/SD/HD/SSD.
2. You need software that is able to create bootable USB drives. There are several options, one commonly used one is Rufus.
3. Browse the web pages of the different flavors/distributions and see which one pleases you visually. Functionally, they all contain the same features, but the interfaces and some other details vary. Common ones include the official Ubuntu, Ubuntu flavors (such as Kubuntu, Xubuntu, Ubuntu Kylin, Ubuntu Cinnamon, Ubuntu Mate and so on), SUSE Linux, Linux Mint, Arch Linux, Fedora and many others. Ubuntu is the most common. In the case of Ubuntu (and flavors), it is recommendable to choose an LTS (Long Term Support) version instead of the very latest one.
4. Once you have chosen a distribution/distributions, download the ISO image on your computer (note: installing Windows or macOS from scratch works essentially the same way using an ISO image).
5. Open the USB writer software (Rufus or other), and create a bootable USB stick from the ISO image you downloaded
6. With the new created USB stick inserted in the computer, reboot.

Pure Linux

Some users may prefer a pure Linux system. This is also a good option if there is an older unused computer around: Linux is not very resource hungry and it works well even on an older computer. As minimal requirements for the Ubuntu 20.04 LTS release, Canonical recommends a minimum of 4GB of RAM, 25 GB of HD, and 2 GHz dual core processors. This means that if there is an older unused computer hanging around, it will probably run Linux just fine. Practical examples: I have installed Linux (Ubuntu Budgie 20.04 LTS) on an 2007 iMac with 4GB memory and it runs fine. There are also special lightweight Linux versions (such as Puppy Linux) that run perfectly well even on older netbooks (I have tried some versions on an old Toshiba netbook with 2 GB of memory). The advantage of setting up Linux on an old unused computer (laptop or desktop) is that that provides and clean install and there is no danger of messing up with your current system.

To install pure Linux, follow the steps in the previous section for creating a bootable USB, try it, and select install after you have tried it out.

How about applications? While there are very good alternatives, there is no native MS Office or Adobe Acrobat for Linux. The situation is not that simple, however, as *Progressive Web Apps (PWAs)* have changed the situation. PWAs are applications that can be installed from the web browser (at this time at least MS Edge, Google Chrome and Vivaldi support them). For example, if one has an Office 365 subscription, it is possible to run Outlook, Powerpoint and MS Word as a PWA, and the same applies to many other software including Adobe Acrobat.

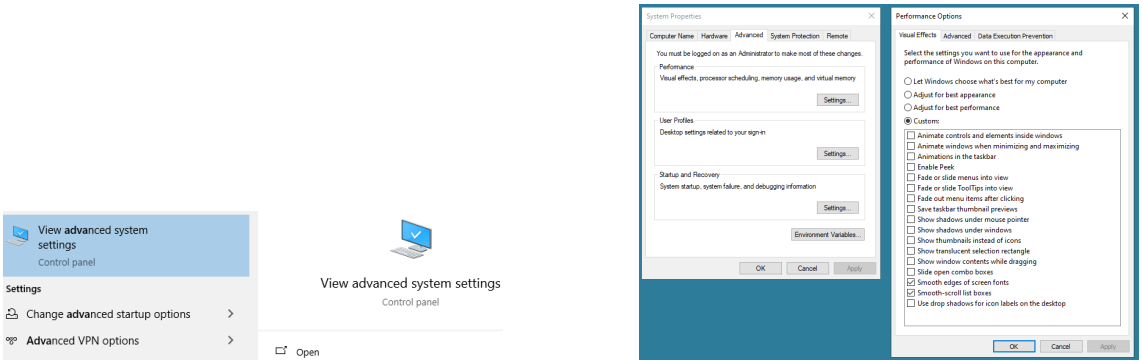


Figure 3.2: To speed up your Windows computer by turning off some unnecessary window decorations and animations, search for `Advanced system settings`. Once there, choose the tab `Advanced` and turn off all the options except the two smoothing options (edges of screen fonts and scroll list boxes; turning them off will give ragged-looking fonts). The speedup can be quite significant especially when running Windows through a Virtual Machine or if the Windows computer has only a modest amount memory and/or one of the less powerful processors.

Dual boot Windows + Linux

In this case Windows and Linux are separated with each of them being a full native installation. This also means that the storage (HD or SSD) must be divided between the two, that us, enough storage space is needed. Total 512 GB is enough as that gives enough space for both Windows and Linux. Note also that the Linux side can access the files on the Windows side but not vice versa (although there is software for that too).

Some manufacturers ship Windows installed with *BitLocker* turned on. If that is the case, things become more complex as the HD/SSD tends to be fragmented and it is not necessarily trivial to defragment it.

Linux in a virtual machine inside your current OS

The advantage of using a virtual machine (VM) is that it allows the installation of full Linux (or other operating systems, including different versions) in a contained environment without the need to repartition the hard drive/SSD. The main concern is the amount of RAM: since memory is shared between the two operating systems when the virtual machine is turned on, one needs to have enough RAM. Typically 16 GB seems to work fine, 12 GB is ok and 8 GB can get a bit sticky especially with more memory-hungry applications. Virtual machine doesn't allow access to GPUs (if you don't have discrete GPUs, then this is not an issue).

3.1.4 Speeding up your Windows

Whether you run just Windows, Windows + WSL or Virtual Machine with Windows as a guest, it is possible to speed up your windows with one simple change: turning of unnecessary window decorations such as shadows and animations, see Figure 3.2 for details.

3.2 File systems

Linux, macOS, and Windows are the most common operating systems, with their own features, and, importantly, each with its own file system. Overall, while all three operating systems have file systems designed to manage data storage, they differ in terms of file system types, case sensitivity,

support for links, permissions, and advanced features. Understanding these differences can be important when working with data across multiple platforms or when selecting a file system for specific use cases. One of the practical issues is that if one has, for example, a USB disk or an external SSD/HD, that has been formatted for one of the operating systems, it may not be readable in the others.

Below, we discuss some of the key differences between the Linux, macOS and Windows file systems.

3.2.1 Linux:

Ext4 (Fourth Extended File System) is one of the most commonly used file systems in Linux, there are also others such as ext2, ext3, and btrfs. Ext4 is the default file system for many Linux distributions, especially in the Ubuntu-based distributions. Of the practical differences to MS Windows is that Linux file systems are case-sensitive, which means `file.txt` and `File.txt` are different files. All the Linux file systems support symbolic links and hard links, which allow for file and directory references (these will be discussed in the next chapter). Access permissions are managed using a system that includes user, group, and others, with read, write, and execute permissions for each category – these will be discussed in the next chapter. Linux file systems often have journaling support (e.g., ext4) to help with data integrity and recovery in case of system crashes or power failures; journaling means keep track of changes that have not yet been committed to the file system. This provides protection against data loss when crashes or unexpected shutdowns occur.

3.2.2 MacOS:

HFS+ (Hierarchical File System Plus) was the traditional file system for macOS, but it has been largely replaced by APFS (Apple File System) since macOS High Sierra (10.13). APFS is designed to be optimized for SSDs and offers features like snapshot support, encryption, and space sharing between volumes. Like Linux, macOS file systems are also case-sensitive, but the default configuration is case-insensitive, which can lead to confusion when transferring files between systems. APFS supports hard links but does not support symbolic links to the same extent as Linux. Permissions are similar to Linux, with user, group, and others categories, and read, write, and execute permissions.

3.2.3 Windows:

NTFS (New Technology File System) is the default file system for Windows. FAT (File Allocation Table) and exFAT are also used in some cases, especially for removable storage. Windows file systems are typically case-insensitive, meaning `file.txt` and `File.txt` are treated as the same file. This can cause problems when copying or moving files between Windows and macOS or Linux/Unix. NTFS supports advanced features like access control lists (ACLs), encryption, compression, and disk quotas. Windows file systems also have symbolic links (since Windows Vista and Windows Server 2008), but it has no native support for hard links.

3.3 The Linux file system in brief

This section provides a brief introduction to the Linux file system. The emphasis on giving an idea of the file system and its structure to those who have little experience using Linux or have not used

Linux before.

Figure 3.3 shows a sketch of the basic directory/file structure. The top-level directory, called the root directory is denoted by `/`. The directories under it are referred with respect to it. For example, in `/home` the slash at the beginning refers to the root directory: thus, `/home` says that the directory `home` is located at the first level under the root directory. If we have a user `sam`, then his/her/their home directory would be located in `/home/sam`, that is, it is at the second level below the root directory.

Note that if you are running Linux using WSL, this is how your Linux part is organized. The organization is also very similar in macOS as it is a Unix-based system. One of the most notable practical differences is that the user directories on a Mac are located under `/User` instead of `/home`.

Important: If you are running pure Linux, Linux through a virtual machine or Mac macOS, you have two ways of accessing your files and the file system:

- Using the command line terminal (=console) window and
- using the file manager provided by your GUI. This is, of course, similar to Mac or Windows.

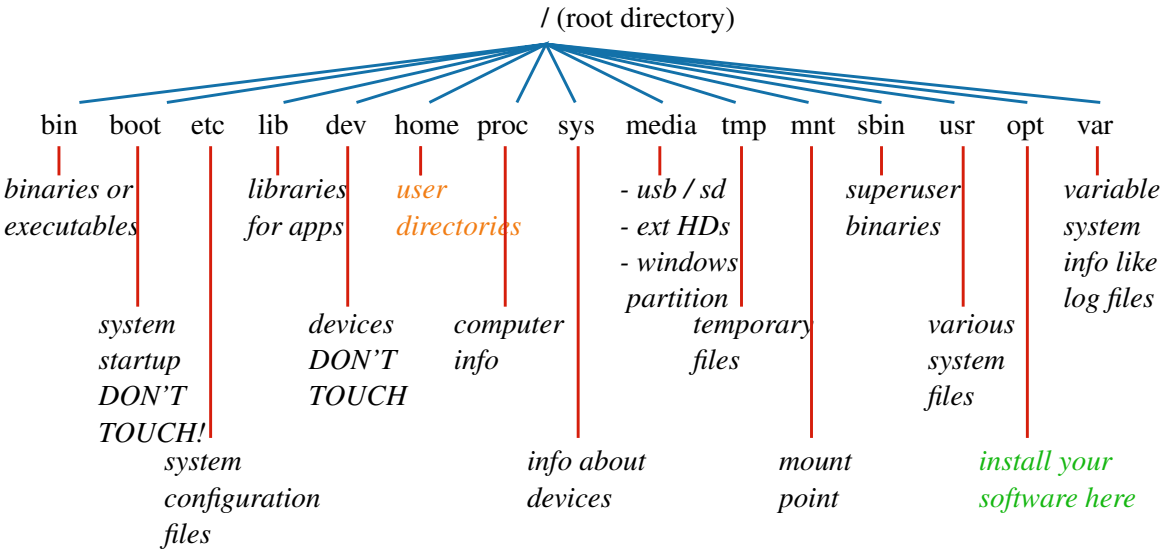


Figure 3.3: Sketch of a typical Linux file system. Orange: this is the location of user directories. For example, if your username is `sam`, this is where your personal files and directories are. The path to this directory would then be `/home/sam`. Green: The directory `/opt` is a good place to put your own software (that you compile).

3.3.1 More about directories

Here are brief explanations of some of the directories shown in Figure 3.3. Note that typically all of the directories in the figure contain subdirectories that are named according to their purposes.

`/bin`

This is where the binaries or executables are located. For example, when you use the terminal to, say, create a new directory, the executable is located in this directory. There is usually no reason to change anything in this directory manually. When installing software using a package manager, this is often the location where the executable is stored. The other common location is `/usr/bin`.

`/home`

The user directories are located under /home. For example, if there are two users /sam and /jane in the system, their personal home directories would be located at /home/sam and /home/jane, respectively; for example, their personal download, music, video etc. directories would be then located (and typically created automatically when a user is created) at /home/sam/Downloads, /home/sam/Music, /home/jane/Downloads, and so on.

/usr and /usr/bin

The name of this directory is confusing since it refers to user. However, as described above, the user home directories are kept under /home. Historically, /usr was the location for the users' home directories. In all modern installations it contains various system files and directories. The directory /usr/bin is a bit special since software (non-operating system related) is put here. For example, web browsers such as Firefox, Google Chrome or graphics software are usually installed here instead of /bin which contains mostly OS-related executables.

/boot

This is the location where the system startup files are kept. Unless you know exactly what you are doing, do not touch any of the files in this directory. Doing so may render your system unbootable and unusable, and it may be very difficult to repair. On the positive side, there is no danger in changing these files accidentally since one has to be superuser to do so.

/etc

This may be the most confusing of the system directory names. As the name suggest, it comes from et cetera, Latin for and the rest. Historically, the files and directories that didn't fit well under other directory categories were put here and hence the name. Nowadays, /etc contains configuration files for the system and some applications. For example, the password-containing (encrypted) passwd file is located in there.

/sbin

This directory contains executable that are not needed by normal users, but the superuser only and hence the name /sbin instead of /bin.

/media

When a USB pen, external hard drive or SD card is connected, this is the location where it shows in the file system

/mnt

In older systems USB devices and SD cards were usually mounted in this directory but now it is more common to use the directory /media as described above. This directory can be used to mount devices manually.

/tmp

All sorts of temporary files stored here by the system.

/dev, /sys and /proc

These directories contain devices and system information and there is usually no need to do anything regarding these.

/lib

This directory contains the various libraries that the different software uses.

`/opt`

This is a directory that can be used to compile and keep various software.

`/var`

Variable files, that is, files whose content changes are kept here. This includes files that are spooled for printing (in `/var/spool` and, very importantly, different log files in `/var/log`. The files include `syslog`, `auth.log` and they include information about functioning of the different devices such as Wifi and bluetooth, users logging in and out and failed attempts to login, and so on. When something goes wrong, these files usually contain the information that can help to fix problems.

3.3.2 Hidden or the so-called dot-files

All operating systems contain hidden files and directories. They contain files and directories that keep information about the system configuration. In both Linux and macOS, these hidden files are the so-called dot-files.

How to see the hidden files:

We will discuss them more later, but you can make them visible (try it out but there is no reason to keep them visible all the time) from your GUI file manager's options in both Linux and Windows.

On Mac, there is no direct access to them through the file manager, but you can make them visible in Finder by clicking `command-shift-dot`.

3.3.3 Additional info:

Important:

As always, keep backups of your own personal directories and data.

3.4 Two user types: superuser and normal

"There are only two industries that refer to their customers as 'users'."
- Edward Tufte, Professor of Political Science, Computer Science and Statistics, Yale.

3.4.1 Normal user

Normal users do not have privileges to change systems setting or install software at the system level, just in their own private directories. This also means that they cannot accidentally or otherwise delete or modify anything in the system and wreck it. Even if one has superuser privileges ones should always operate as a normal user and use the superuser rights only when absolutely necessary. By default, that is, when you login, you are a normal users. This applies to both Windows and Linux. Superuser rights must be called explicitly in both systems.

3.4.2 Superuser

For comparison: On Windows, the Linux/Unix superuser, also called `root`, account corresponds to the Administrator account. The Administrator in Windows and the superuser have the privileges to

modify the system files, add and delete users, and install software to the system (normal users can typically install some software in their private directories).

3.4.3 Important: What not to do

Although this may sound trivial, it is impossible to overemphasize that you should never use the superuser privileges when they are not needed for system maintenance. Performing normal operations as the superuser has several risks, the most obvious one is that as a superuser you can delete and modify system files that may wreck your system and make it hard or impossible to recover without full re-installation. Normal users cannot do such damage. The worst damage a normal users can do is to destroy their own files and directories but that does not wreck the system. The situation is the same in all operating systems. Just to give an example: A superuser or a malicious intruder who has somehow acquired the superuser password can wipe out the full system with one single command. This brings in the next important issue.

3.4.4 Important: Password and security

Passwords should always be chosen such that they are safe. The superuser password is particularly important since if anyone has that, they will be able to control and take over the computer or computer system.

Just for curiosity: The term superuser is a real term.

We will refer to these terms and issues in what follows.

3.5 External links

- Random computer stuff: Random notes on various computer issues, installation for Windows, Mac and Linux.

Chapter 4

Editing ASCII files

When programming, editing system files, using LaTeX, and so on, one needs an ASCII editor. MS Word and such can absolutely not be used as they use their own proprietary formats – editing and saving system files in formats other than plain ASCII may render your system unusable. Below are a few notes regarding ASCII editors. There are many options some of which are listed. `vi` is the canonical ASCII editor that comes with all Linux/Unix-based operating systems including macOS. One should know the basics of `vi` even if some other editors are used for programming and other tasks.

ASCII, or what is commonly called as 'plain text' is this context, stands for "American Standard Code for Information Interchange." It is a character encoding standard that was developed in the early 1960s to represent text and control characters in computers, communication equipment, and other devices that use text. ASCII assigns a unique numerical value (code) to each character, allowing computers to store, transmit, and display text using a standardized set of codes. ASCII includes a set of 128 characters, which include both *printable* and *control characters*. The printable characters are the ones you see on your keyboard, such as letters (A-Z, a-z), digits (0-9), punctuation marks, and special symbols like the dollar sign ($) and percent symbol (%). Control characters are non-printing characters used for various control functions, such as carriage return (CR), line feed (LF), and tabulation (TAB). ASCII does not include accented characters such as ü, Å, é, ç and so on. The extended character set ISO 8859-1 includes them as well. Importantly, ASCII is compatible with most computer systems and programming languages. This compatibility makes it a fundamental building block for text processing in computing.

4.1 The `vi` editor

For anyone coming directly from GUI-based systems, the `vi` editor typically appears very cumbersome. It is, however, very lightweight and it is automatically installed with more or less all Linux/Unix based systems including Mac's terminal application. On the more grave side, if things go badly wrong and some files need to be fixed, `vi` is usually the choice. Even the live distributions include it. On the light side, despite its simplicity it is very powerful and can be used to edit programs, LaTeXdocuments etc. and for many it is the editor of choice.

The usual obstacle for the first time users is to understand that `vi` has two different operating modes: 1) *insert mode* and 2) *command mode*. To edit any document, one has to be in the *insert mode*. When one opens `vi`, it always starts in the *command mode*.

In command mode: You cannot insert text, only commands. This includes deleting characters

(see below), saving the document, quitting the editor and so on. You can also use the arrow keys to move to a different location the file. The arrow keys cannot be used for this purpose in the insert mode.

In insert mode: You can type in any text. But you cannot delete characters. For that, you must switch back to the command mode. This tends to be the most difficult concept at the beginning.

Appendix A provides a detailed discussion of the `vi` editor.

4.2 ASCII editors other than `vi`

Note that independent of your operating system, programs (independent of the programming language), system files (independent of the operating system), LaTeXfiles and some other must be edited with text editors that don't impose any formatting or hidden characters. Below is a list of some possibilities.

4.2.1 Atom

Atom is a very powerful editor with lots of plugins for many tasks. It is available for Windows, Linux and Mac. It also has Git control which may be important depending on your preferences.

4.2.2 Visual Studio Code

Visual Studio Code is from Microsoft and it is free. It is available for Mac, Windows and Linux.

4.2.3 pico and nano

One (or both) of them come installed with Linux and Mac terminal. They are run from the terminal and are very easy to use.

4.2.4 Emacs

Is a very powerful editor that is available for Linux, Mac and Windows. It has a bit of a learning curve, but it depending on your longer terms needs it may be worth it. It has excellent support for various programming languages and LaTeX.

In Linux there are also others

Such as `gedit`, `kate` and so forth. The typically even have GUIs.

Chapter 5

Command line: Some basic & useful commands

They have computers, and they may have other weapons of mass destruction.

The 78th United States attorney general Janet Reno

This chapter lists some of the most important and useful commands with examples, and provides a very brief introduction to *environment variables* and *automating tasks using shell scripts*. Additionally, some notes regarding the difference between Linux (`bash` shell) and macOS (`zsh`) are provided.

Warning:

When there is a warning, take it very seriously as there are a few commands that are dangerous!

5.1 Shell

Shell is the interface between the user and the operating system, that is, it is a user interface for accessing an operating system's services. It can be understood as a layer that interprets user commands and translates them into actions the computer's operating system can perform. There are two main types of shells: command-line interfaces (CLI) and graphical user interfaces (GUI). Here, the focus is on CLI.

The CLI is a text-based interface where users interact with the software by typing commands into a console or terminal. This type of shell is often favored by more advanced users due to its powerful capabilities and efficiency for certain types of tasks. In a CLI, everything from file manipulation, software installation, system monitoring, and programming can be done through text commands. It offers a more direct and less resource-intensive way of interacting with the operating system compared to GUIs.

There are several different types of shell, for example:

- `sh` (Bourne Shell): The original Unix shell. It was developed by Stephen Bourne at AT&T's Bell Labs in the late 1970s.
- `bash` (Bourne Again Shell): This is the shell that is most commonly used in Linux (and WSL). Developed by Brian Fox, released in 1989 as part of the GNU project. `bash` includes advanced programming features that are absent in `sh`, like arrays, select loop constructs, and advanced string manipulation capabilities. `bash` has an improved command-line interface, including features like command completion, command history, and the ability to edit the command line using keyboard shortcuts, and it supports a more extensive set of environment variables and shell options that provide greater control and customization of the user environment. `bash` is largely compatible with `sh` scripts, as it was designed to be backward compatible with the Bourne shell. Importantly, `bash` supports job control, a feature that allows users to suspend and resume execution of commands, which is not available in the traditional `sh`.
- The macOS terminal uses `zsh` (the Z Shell) from macOS Catalina onwards and `bash` prior to Catalina. It was created by a Princeton student Paul Falstad in 1990, and it can be considered as extended `bash` shell. `zsh` has advanced interactive features, customization, and extensive plugin system. The choice between `bash` and `zsh` largely depends on the user's preference, the need for advanced interactive features, and the specific environment they are working in.
- Others: `csh` (the C Shell), `tcsh` (improved C Shell; it was created for the TENEX operating system and the letter T comes from that)

In the following we assume `bash` shell unless otherwise mentioned. In some cases, differences between `bash` and `zsh` are pointed out.

To check out which shell you are running, open a CLI and type

```
echo $SHELL
```

5.2 How to get help: The commands `man` and `apropos`

As the very first thing, let's list a couple of methods how to get help using the command line. To get help on help, try

```
man man
```

It is instructive to check each of the commands below using `man`. If for some reason there is no man-page (that's how the manual pages are often called) for a command, that probably means that full documentation was not installed.

`apropos:` You know what you want to do but cannot remember the command.

This can return a long list of commands, but it provides a lot of information. For example, to get to know what kind of copying commands are available, try

```
apropos copy
```

As you can see from the output, the list is very comprehensive.

5.3 Who am I and who is in the system?

`who:` Who is logged in the system and the sessions they have.

```
man whoami
```

This shows also the users logged in the system, since when and some other information. The command `who` has lots of options (check using `man who`) including

`who -a` Shows all the users logged in, the last time the system was booted and some other information.

`who -b` Shows when the systems was booted the last time.

Example 5.1

```
sam@linux:~$ whoami
sam
```

Example 5.2

```
sam@linux:~$ who -b
        system boot  Nov 12 14:54
```

Example 5.3

```
sam@linux:~$ who -a
        system boot  Nov 12 14:54
        run-level 5  Nov 12 14:54
sam   tty1         Nov 12 15:21 (:0)
sam   pts/1        Nov 12 15:21 (:0)
sam   pts/2        Nov 12 15:21 (:0)
```

5.4 What is the name of my system and related

Below are some of the useful system information commands. There are several more for various purposes, but here only some the more common ones are listed.

`hostname`: Show (can also used to change) hostname (=the name of your computer).

The command `hostname` has several options (check using man hostname) including

`hostname -I -a`: Shows your IP addresses.

macOS (`zsh`) note: This command also exists on Mac, but the options are different, check using `man hostname`

Example 5.4

```
sam@linux:~$ hostname
linux
```

`uname:` Shows system information.

This is a very useful command and it has several options (check using `man uname`) including

`uname -a` Shows all information.

`uname -r` Shows the kernel release.

`uname -o` Shows the operating system.

Example 5.5

```
sam@linux:~$ uname
 Linux
```

Example 5.6

```
sam@linux:~$ uname -a
Linux linux 5.4.0-53-generic #59-Ubuntu SMP Wed Oct 21 09:38:44 UTC 2020 x86_64 x86_64 x86_64 GNU/Linux
```

`lscpu`: Shows information about your CPUs (list CPU).

Has several options (check using `man lscpu`).

macOS (`zsh`) note: This command doesn't exist on Mac by default. Instead, you can use the command

```
system_profiler SPHardwareDataType
```

macOS (`zsh`) note: It is possible to get the Linux command + some others by installing the package `util-linux` using Homebrew:

```
brew install util-linux
```

Example 5.7

```
sam@linux:~$ lscpu
Architecture:                    x86_64
CPU op-mode(s):                  32-bit, 64-bit
Byte Order:                      Little Endian
Address sizes:                   39 bits physical, 48 bits virtual
CPU(s):                          12
On-line CPU(s) list:             0-11
Thread(s) per core:              2
Core(s) per socket:              6
Socket(s):                       1
NUMA node(s):                    1
Vendor ID:                       GenuineIntel
CPU family:                      6
Model:                           158
Model name:                      Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz
Stepping:                        10
CPU MHz:                         4266.032
CPU max MHz:                     4500.0000
CPU min MHz:                     800.0000
BogoMIPS:                        5199.98
Virtualization:                  VT-x
L1d cache:                       192 KiB
L1i cache:                       192 KiB
L2 cache:                        1.5 MiB
L3 cache:                        12 MiB
NUMA node0 CPU(s):               0-11
Vulnerability Itlb multihit:     KVM: Mitigation: Split huge pages
```

`lsb_release`: Shows distribution specific information. LSB = Linux Standard Base

Has options (check using man `lsb_release`).

`lsb_release -v` Shows the version.

`lsb_release -a` Shows all information.

macOS (`zsh`) note: This command doesn't exist on macOS.

Example 5.8

```
sam@linux:~$ lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 20.04.1 LTS
Release:        20.04
Codename:       focal
```

5.5 File / directory operations

This section focuses on various file operations, such as how to how to see the current work directory, how to list, create, copy and delete files and directories and how to move between directories. The concepts of file permissions and ownership are also discussed.

5.5.1 Some special conventions related to files and directories

- Single dot (`.`) refers to the current directory.
- Double dots (`..`) refers to the directory at the next higher level.
- Tilde (`~`) refers to home directory.
- Hidden files start with a dot (`.`).
- Hyphen (`-`) refers to the previous directory where you were.

5.5.2 Home directory and work directory / current directory

`pwd`: Show where I am.

The command `pwd` prints the current directory on screen (also called work directory - `pwd` = print work directory). There are not man options for pwd (check using `man pwd`). The `pwd` command shows the full path.

Example 5.9

```
sam@linux:~$ pwd
/home/sam
```

This means that the user `sam` is currently in his/her home directory.

5.5.3 Show the files and directories

`ls`: Show the files and directories, that is, the contents of current directory

The command `ls` has a lot of useful options as one can see by using `man ls`. Here are some (the option `-l` is used in connection with most of them since it is very useful to see the file/directory details):

`ls -l`	Long listing, shows the file/directory properties including size, time of last modification, owner and access rights.
`ls -lt`	The option `-t` lists the files ordered such that the most recently modified is listed first. This is very useful with the option `-l`
`ls -ltr`	Using the previous with the option `-r` lists the files in reverse order: such that the most recently modified is listed last.
`ls -la`	The option `-t` lists all the files including the hidden files that start with a dot.
`ls -lS`	The option `-S` lists the files with the largest one first.
`ls -lh`	The option `-h` lists the files such that size is in a human readable format. This means sizes are given in the form 1k, 1M, 1G. This is very useful.

Note: The dot directories. When using `ls -la`, there are two directories that have the names dot (`.`) and (`..`) listed. As described above, dot refers to the current directory and double dot to the one directly above. These can be used when moving around directories as will be done below.

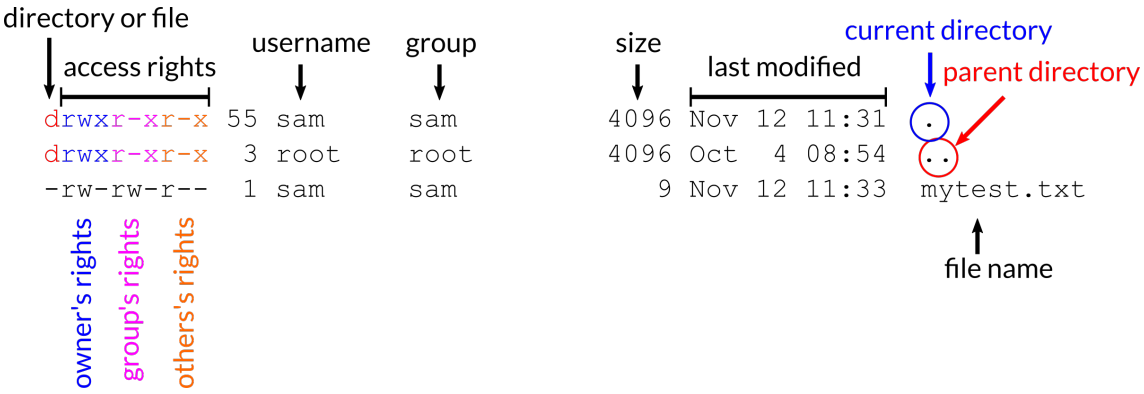


Figure 5.1: When getting the long listing with the file and other details, this is how it could look like. The dot directories are also shown. The access rights come in three groups, those of the user's, group's and other's. `r`=right to read, `x`=right to execute, `w`=right to write, `-`=no right's given. Dash in the first column means file while `d` stands for directory.

Note: File details when using the long listing.

`tree`: Show the directory structure.

This is a neat tool that can be used to view the directory structure. The command `tree` may already be in your system. To check if that is the case, open a terminal window and type

```
which tree
```

If you see something like

```
/usr/bin/tree
```

Then the command is already installed. If that is not the case, you can do it as follows in an Ubuntu/Debian-based systems:

```
sudo apt install tree
```

Let's check how it works. The command gives the directory tree (as in the Figure) with respect to the argument. If the argument is the root directory `/`, the command

```
tree /
```

prints the directory tree starting from the root level on the screen. You can stop it by pressing `ctrl-C`

Note: The keystroke `ctrl-C` can always be used to stop a process than runs on your screen. It is very useful. There are other ones as well but they will be introduced later.

If the above would be all there is to tree, then it wouldn't be super useful. But fortunately it takes different optional arguments. One particularly useful one is the option `-L` number that defines the level. For example,

```
tree -L 1 /
```

shows only one level below the root directory and

```
tree -L 1 -t /
```

provides listing that is ordered from the ones that was modified the longest time ago to the latest one.

To get more information about tree, simply type `man tree` on the terminal screen.

macOS (`zsh`) note: If this command is not on your Mac, you can install it using Homebrew:

```
brew install tree
```

5.6 Wildcards

Below is a list of common wildcards. They are very useful when searching, listing copying and so on. These should not be mixed up with the conventions listed above. There is a distinction between wildcards and regular expressions, but we are not going to discuss the latter here.

`*` match any number (zero or more) of characters. Example: `ls a*` lists all the files and directories starting with the letter `a` (small `a` only, capitalization matters).

`?` Match a single character. Example: `ls a?v*` lists files that start with the letter a, the second character can be any character, the third character is small `v` followed by any number of arbitrary characters.

`[ ]` range of characters. For example, `A\[abc\]D`, matches strings that start with the letter `A`, end to the letter D and have any combination of the characters `abc` between them. Example: `ls a[dvc]*` lists files that start with an `a`, then have any combination of `dvc` followed by any number of any characters.

`!` The exclamation mark is logical not. When used in example `A\[!abc\]D`, matches strings that start with the letter `A`, end to the letter `D` but doesn't have any of `abc` between them. Example: `ls a[!dvc]*` lists files that start with an `a`, then have no combination of `dvc` followed by any number of any characters.

5.6.1 Moving between directories

Now that we know how to see where we are, and list the files and directories, let's look at how to move between directories.

`cd`: Change directory.

`cd` means change directory. If you want to move, for example, into your downloads directory (it is assumed that you are in your home directory), type

```
cd Downloads
```

Capitalization. Capitalization matters, usually the downloads directory is spelled with a capital but you can of course check that by using the ls command.

Using tilde (`~`) to get your home directory. Tilde is refers to your home directory and it is really handy. For example, to move your home directory from wherever you are, typing

```
cd ~
```

will take you home. Or, if you want to move from some remote directory to the downloads directory, simply type

```
cd ~/Downloads
```

The slash character is needed since it refers to a directory.

Using plain cd to get your home directory. Simply using

```
cd
```

moves you back to your home directory.

Using double dots `cd ..` to move up one level. Simply using

```
cd ..
```

moves you up one level. For example, if you are in the downloads directory, `cd ..` moves back to your home directory since it is one level above, that is, it is the parent directory.

```
pushd: Switch easily between two (or more) directories.
```

The command `pushd` adds a directory (or directories) in the directory stack and allows you to move easily between them. In the simplest case, if you are currently located in your home directory, just type

```
pushd ~Downloads
```

This adds the directory Downloads to the stack and moves you there. Typing

```
pushd
```

takes you back home or whatever is the other directory in the stack (there can be more than one). Executing `pushd` once more takes you to `Downloads`. This is very handy when moving between directories that have long paths.

How to see which directories are included in the stack?

```
dirs -v
```

5.6.2 Creating and deleting files and directories

Now that we know how to see where we are and list the files and directories, let's look at how to create and delete files and directories.

`touch`: Create an empty file.

Doesn't sound like much of a command but in practise this it is very useful. In addition to generating new empty files, `touch` can also be used to change the time when a file has been modified. This can be very helpful if the system (like some HPC systems) automatically deletes files older than some specified time if they are in some particular directories (such as temporary work directories).

To create a file called `my_text_file.txt`, try:

```
touch my\_test\_file.txt
```

`mkdir`: Create a new directory. To create a new empty directory, called `my_test_directory`, try:

```
mkdir my_test_directory
```

`rmdir`: Delete a directory.

To delete a directory, it must be empty (this is a good safeguard). If the directory is not empty, `rmdir` won't do anything. Since the newly created directory is empty, let's delete it:

```
rmdir my_test_directory
```

`rm`: Delete files.

The command `rm` is (obviously) very useful but also potentially very dangerous especially some of the options. The dangerous options are listed separately with a warning.

The command `rm` does not remove directories, empty or not, unless the corresponding option is given, see below. Here are some useful options `rm` takes:

`rm -v` Verbose. Does not prompt but tells what was done.
`rm -i` Interactive remove, prompt for every file being removed.
`rm -d` Removes empty directories.

Warning:

Be very careful with the following and **DO NOT EXECUTE THEM NOW**.

`rm *` **Deletes absolutely everything in the current directory**. There is no way to recover any of the files unless there is a separate backup.

`rm -r *` **Deletes absolutely everything *recursively***. That is, everything including possible subdirectories are deleted. As the most drastic example, the superuser can delete the whole system if this command is accidentally executed at the root directory. In such as case, the is fully deleted and nothing can be recovered.

5.6.3 Copying, renaming and moving files and directories

Now that we know how to see where we are and list the files and directories, let's look at how to create and delete files and directories.

`cp`: Copy files or/and directories.

The command `cp` has lots of useful options, check using `man ls`. The wildcards listed above are also very useful with this command. Here are some useful options:

`cp -p` This option preserves the date last modified (time stamps), ownership and mode. A very useful option.
`cp -i` Interactive copying. Asks if an existing file should be overwritten or kept.
`cp -n` Doesn't allow overwriting existing files.
`cp -u` If a file with the same name exists, it is overwritten only of the source file is newer than the existing file at the destination.
`cp -r` Copies recursively, that is, copies also subdirectories.

`mv`: Move or rename files and directories.

The command `mv` can be used with both files and directories. Below are some useful options (very similar to `cp`):

`mv -i` Interactive mode. Very useful to avoid overwriting files or/and directories.
`mv -n` Prevents overwriting existing files/directories.
`mv -u` If a file/directory with the same name exists, it is overwritten only of the source is newer than the destination.

Example 5.10

Let's 1) create an empty file, but first check that it is not there, 2) copy it, 3) rename the original, 4) list the files using a wildcard, 4) remove the original, 5) remove the copy as well, and finally 6) check that both of the files have been deleted.

```
sam@linux:~$ ls test.txt
ls: cannot access 'test.txt': No such file or directory
sam@linux:~$ touch test.txt
sam@linux:~$ ls test.txt
test.txt
sam@linux:~$ cp test.txt test-copy.txt
sam@linux:~$ ls test*
test.txt test-copy.txt
sam@linux:~$ mv test.txt test-original.txt
sam@linux:~$ ls test*
test-original.txt test-copy.txt
sam@linux:~$ rm test-original.txt
sam@linux:~$ rm test-copy.txt
sam@linux:~$ ls test*
ls: cannot access 'test*': No such file or directory
```

5.7 What is in your files - seeing file contents

At this time we don't have files with content but since we will needs these commands shortly, let's list them briefly and elaborate when we have files with content.

`cat` Concatenate files and print on screen (or standard output to be more precise).
`head` Show the first 10 lines of a file.
`tail` Show the last 10 lines of a file.
`more` Show the file contents screenful-by-screenful.
`less` Opposite of more.

5.8 Find files and directories

`find:` Find files and directories.

The `find` command is very powerful and it has lots of options and different ways to use them. It also finds files and directories very fast. Here, the basic usage with some options are provided. This is how the command works:

```
find [where to search] [criterion for searching (name, type, date of modification...)] [-options] [what to
    find]
```

The command find is somewhat more complex than the ones above. It's usage is illustrated with examples:

`find . -name myfile.txt` Searches the current directory and any directory below it for a file or directory with the precise name `myfile.txt`.

`find . -name "myfile*"`	Searches the current directory and any directory below it for a file or directory that start(s) with `myfile`.
`find . -type d -name MYDIR`	Searches the current directory and any directory below it for a directory (`-type d`) with the precise name `MYDIR`.

Example 5.11

Since at this time you don't have (assuming you have a new installation) much on your computer, let's search for something we know have:

```
sam@linux:~$ find . -type d -name Downloads
./Downloads
```

Example 5.12

Let's us a wildcard:

```
sam@linux:~$ find . -type d -name "Down"
./Downloads
```

5.9 Compare files, extract information

`diff`: Compare the contents of files Usage:

```
diff file1 file2
```

Example 5.13

Assume that you have two files, one of the is an older version of some text and the other newer. To see the difference (=changes), use

```
diff old.txt new.txt
```

`grep`: Extract lines with a given pattern from output

Example: List all the files with details and in chronological order, but print on screen only the ones that have the pattern `txt` in them:

```
ls -lt | grep txt
```

5.10 Linking files

`ln`: Link two files.

There are two types of links: 1) hard link and 2) soft (or symbolic or symlink) link. A *hard link* is essentially another name for an existing file on the same file system. Both the original file name and the hard link point directly to the same underlying `inode` (the file system's internal representation of a file). A *soft link* is a reference or pointer to another file or directory. Unlike a hard link, a symbolic link does not point to an `inode` but instead contains the path to the linked file or directory. This is the most common type of link, and thus the most common option is `-s` that makes a symbolic or soft link. If the original file is deleted, moved, or renamed, a soft link becomes a 'dangling' link that points to a non-existent file.

Example 5.14

Link, using a symlink, the file `new.txt` to the file `target.txt`. This means, for example, that if you delete `link.txt` the file that has the actual content (`target.txt`) remains intact or if you edit `link.txt`, the changes will be put in `target.txt` (similarly, if you directly edit `target.txt`, the changes appear immediately in `link.txt`). This is a very handy command.

```
ln -s target.txt link.txt
```

5.11 Processes: View, stop, modify, and set priority

`ps`: Change the execution priority.

This is a very useful command. Here are some examples

Show all processes:

```
ps -e
```

Show your active processes:

```
ps -a
```

View all the process (switch `-e`) with details (switch `-f`), and filter (`grep`) your processes from them:

```
ps -fe | grep ${USER}
```

The above syntax has a few very useful points: 1) the command `grep` (Global Regular Expression Print) is a tool for searching text and displaying lines that *match a specified pattern*. 2) The above uses *the pipe* (`|`). The pipe is used for creating a series of commands that are linked by their standard streams (outputs). The pipe takes the standard output (`stdout`) of one command (here, `ps -fe`) and feeds it directly as the standard input (`stdin`) to another command (here, `grep ${USER}`). This ability to chain commands together is a fundamental aspect of Unix and Linux shell scripting and command-line usage, as it enables the creation of complex command-line operations from simpler individual commands. 3) The above uses the *environment variable* `$USER`. Environment variables provide a way to share configuration settings between multiple applications and processes in Unix-like operating systems. We will discuss them more below.

`top`: Show a live view of the processes in the system.

Opens a screen (on your terminal, see Figure 5.2) that shows the processes on your computer and, for example, how much memory and CPU they take, and what is their priority (the column PR). This is a very handy command. You can quit the screen by typing `q`.

`kill`: Kill/end processes.

Example 5.15

The following kills the process number `11220`

```
kill -9 11220
```

Example 5.16

The following kills all of your process and logs you out

```
top - 09:26:46 up 8 days, 1 min,  2 users,  load average: 9.21, 4.92, 3.32
Tasks: 433 total,   8 running, 425 sleeping,   0 stopped,   0 zombie
%Cpu(s): 47.6 us,  9.6 sy,  0.0 ni, 42.3 id,  0.4 wa,  0.0 hi,  0.2 si,  0.0 st
MiB Mem :  31926.3 total,   7696.5 free,  11777.5 used,  12452.3 buff/cache
MiB Swap:      0.0 total,      0.0 free,      0.0 used.  16923.6 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
 266334 mka       20   0 1155.4g 445316 125840 S  96.0   1.4 815:01.40 msedge
 641859 mka       20   0 1141.9g 851708 158540 R  96.0   2.6   1:10.42 Vivaldi-Content
 641972 mka       20   0 1131.3g 924320 107072 R  74.5   2.8   0:47.87 Vivaldi-Content
   7591 mka       20   0   33.2g 710376 289900 S  71.2   2.2  47:27.18 vivaldi-bin
 641943 mka       20   0 1131.3g 216152 111072 S  36.4   0.7   0:10.08 Vivaldi-Content
   7667 mka       20   0 1135.5g 306732 128224 S  32.5   0.9  21:33.65 Vivaldi-Extens
   3416 mka       -2   0 2975232 303676 195204 S  31.1   0.9 100:33.68 kwin_wayland
   3951 mka       20   0   33.4g 233444 119992 S  27.2   0.7 326:49.65 msedge
   3552 mka       20   0 5199460 557620 157632 S  17.9   1.7   5:53.59 plasmashell
   7641 mka       20   0   33.4g 275048 153124 S  17.2   0.8 110:11.14 Vivaldi-Gpu
   1139 root      20   0 2161504 113488  11904 S  14.6   0.3  30:11.90 DisplayLinkMana
   7643 mka       20   0   32.4g 170148 107528 S  12.6   0.5  10:50.71 Vivaldi-Util
 425228 mka       20   0 1155.4g 173604 120664 S  10.3   0.5 163:30.92 msedge
 641999 mka       20   0 1131.3g 150756  99712 S   6.0   0.5   0:03.40 Vivaldi-Content
 642734 mka       20   0 1131.3g 159192 114976 S   6.0   0.5   0:05.61 Vivaldi-Content
   3472 mka       20   0 1361420 123472  77696 R   5.6   0.4  39:14.54 Xwayland
 643121 mka       20   0  736440  94620  73984 S   5.6   0.3   0:00.17 spectacle
```

Figure 5.2: Output of the command `top`. The upper part of the screen shows general information about the system and processes, such as uptime, number of users, how much CPU and memory is being and used and so on. The lower part shows the processes that use the most CPU. In this case, the web browsers (Vivaldi and MS Edge) take a lot of CPU.

```
kill -9 -1
```

`nice`: Change the execution/scheduling priority.

Priority `-20` is the highest and +19 the lowest priority in terms of CPU allocation. This is important since, when you run simulations or do extensive data analysis, you should set the priority of such processes to `+15 - +19`. When you do that, the process has a lower priority meaning that you can also use the computer do something else when needed (such as editing text etc.), but when you are not using it for other tasks, your simulation or analysis process will take full resources.

Example 5.17

This is how to set a priority of a process to the value +15:

```
nice -n +15 command_name
```

If the process is already running, it can be reset using the command `renice`

Example 5.18

This is how to reset the priority of an ongoing process (see above how to get the process ID):

```
renice -n +15 -p process_id
```

5.12 File permissions and ownership

`chmod`: Change the access rights.

This is a very useful command.

`chown`: Change ownership.

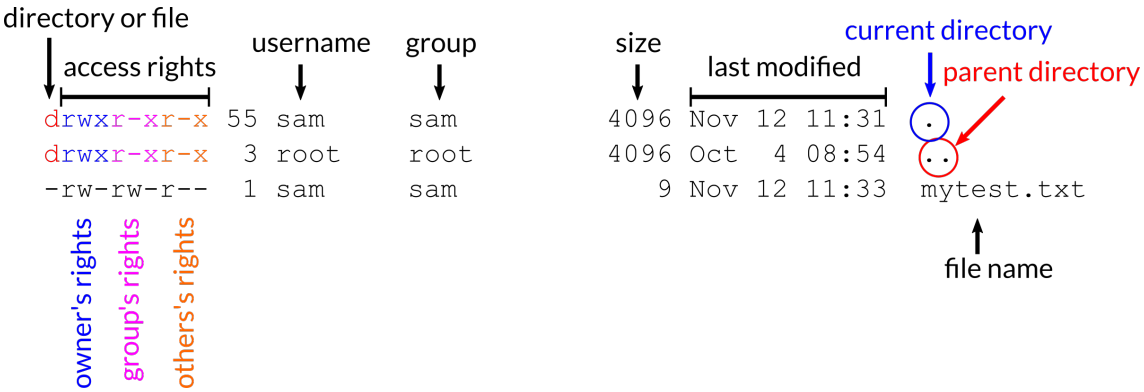


Figure 5.3: When getting the long listing with the file and other details, this is how it could look like. The dot directories are also shown. The access rights come in three groups, those of the user's, group's and other's. `r`=right to read, `x`=right to execute, `w`=right to write, `-`=no right's given. Dash in the first column means file while `d` stands for directory.

Example 5.19

Assume that you edited a file `text.txt` as the superuser and now you want to change the ownership to user `sam` who belongs to the group `users` (you can see the user and the group by using the option `-l` with `ls`):

```
sudo chown sam:users text.txt
```

5.13 Environment variables

As discussed abobe, environment variables provide a way to share configuration settings between multiple applications and processes in Unix-like operating systems. Environment variables are fundamental for the configuration and operation of the shell environment and the applications running in it. They provide a flexible way to influence the behavior of shell processes and scripts, allowing for a customizable and dynamic computing environment. Understanding and effectively using these variables can significantly enhance the user's ability to *automate and streamline tasks*. Common environment variables include,

`PATH`	A list of directories the shell searches for commands.
`HOME`	The path to the current user's home directory (e.g., `/home/sam`).
`USER`	The name of the current user.
`SHELL`	The path to the current user's shell (e.g., `/bin/bash`).
`UID`	The numeric user ID of the current user.

One can view the environment variables individually by using the `echo` command and the dollar sign (`$`; indicates a veriable) in front of the variable.

Example 5.20

```
echo $PATH.
```

Another method is using the command

`printenv`: Show all the environment variables.

Example 5.21

See all the environment variables that have the text `PATH` in them (capitalization matters):

```
printenv |  grep PATH
```

5.14 Scripting: Automating tasks

Scripting provides way to automate tasks, manage system operations, and process data. A script is a plain text file containing a series of commands that would normally be typed at the command line. These scripts can include loops, conditionals, variables, and functions, making them versatile for a wide range of tasks – scripting is a fundamental skill for power users of Unix-like systems. Scripting works in all modern shells including `bash`, `zsh`, `tcsh` and so on.

The best to see this is by examples. The following script This displays the date, current directory and files in it

Example 5.22

```
#####################################################################
# The character # comments out whatever is after it
#
#!/bin/bash      # this calls the bash interpreter.
#
# This script displays the date, current directory and files in it.

echo "Date and Time: "     # The command 'echo' prints on screen
date                       # Calling the command 'date'
echo "Current Directory: "
pwd
echo "Files in the Directory: "
ls
```

Write this, using an ASCII editor such as `vi`, in a file (below, `test_script.sh`. Then, this can be executed using the command

```
bash test_script.sh
```

Another way of executing it is by changing the file itself executable using the `chmod` command,

```
chmod u+x test_script.sh
```

Then, it can executed using

```
./test_script.sh
```

Example 5.23

This is a more complex example demonstrating the uses of conditionals (`if`-statements), `for-loops`, variables and pattern matching. Save the below to a file and execute the same way as above.

```
#####################################################################
#!/bin/bash
#
# Script to check for files that end '.txt' in a specified directory
# This example uses if-statements, for loops and pattern matching.
#
#####################################################################
```

```
DIRECTORY=$1 # Directory to check, passed as an argument

if [ -z "$DIRECTORY" ]; then
    echo "No directory specified. Using the current directory."
    DIRECTORY="."
fi

TXT_FOUND=0

echo "Checking for '.txt' files in $DIRECTORY..."

for FILE in "$DIRECTORY"/*.txt; do
    if [ -e "$FILE" ]; then
        echo "Found: $FILE"
        TXT_FOUND=1
    fi
done

if [ $TXT_FOUND -eq 0 ]; then
    echo "No '.txt' files found in $DIRECTORY."
fi
```

5.15 Optional: Create work directory for the course

Open a terminal window and type

```
cd
```

This changes the directory to your home directory.

Then, give the command

```
mkdir SciComp9502B
```

This directory can be used for course related data (probably a good idea to create subdirectories in it for clarity), scrips and programs.

5.16 Problems

Problem 5.1 Allowing only for a single press of the `Return` key (=the command/string of commands has to be given on one line only and not as a series of commands), find out all the information regarding your CPU and its properties.

Problem 5.2 What is the command that can list all files starting with the letter `n` and the third letter being also `n` in the directory `/bin`?

Problem 5.3 Allowing only for a single press of the `Return` key (=the command/string of commands has to be given on one line only and not as a series of commands), give the command that lists all the files and directories under the directory `/usr` such that the full path is visible (=information where they are located in the directory structure shows up).

Problem 5.4 Allowing only for a single press of the `Return key` (=the command/string of commands has to be given on one line only and not as a series of commands), give the command that lists all the files that start with either the letter `z` or the letter `y` under the directory `/bin` in such a way that listing ordered from the smallest to the largest file and the file sizes are listed in K, M, G etc.

Problem 5.5 Important: Document each step and have screen shots when editing the files and printing the outputs of commands. First ensure that you are in your home directory. If you created the course directory as discussed in the Linux module move there or, alternatively, create a new directory for this exercise (it is a good idea since files and directories will be created). Check both your username and path, and create a new directory called `TESTING` inside the course directory (or the directory that you created). Move inside that directory and create two new empty files without opening the files with an editor. Call the files `mytext_1.txt` and `mytext_2.txt`. Using the `vi` editor (instructions for `vi` are provided as an Appeninx in the course notes), edit the files such that the first one has the text:

```
This is my first file.
```

And the second file should have the text:

```
This is my second file.
```

Make sure that you have saved the files. Using a single command and without opening the files, compare the differences between the two files.

Problem 5.6 Using a single command, append the following text in `mytext_1.txt`:

```
This text goes in mytext_1.
```

Repeat the comparison between `mytext_1.txt` and `mytext_2.txt` from the previous problem.

Problem 5.7 Link a new file `mytext_temp.txt` to `mytext_1.txt`. Show also that the linking worked and show the contents of the new file. Note: This is not the same as copying using `cp`. Use `vi` to edit `mytext_1.txt` and add the following text in it:

```
Written after linking.
```

Show the contents (on screen; take a screenshot) of `mytext_tmp.txt` on screen without opening `vi`.

Problem 5.8 Delete the files that have names starting `mytex` such a way that you have to confirm each deletion. Then move to the main course directory and delete the directory `TESTING`.

Problem 5.9 Open the `vi` editor. Then, open a new terminal window and keep `vi` running in the other. Next, in the new window, list all the processes that `vi` has. Finally, using command line in the second terminal (=not using `vi`), terminate `vi` and show that it has been terminated.

Problem 5.10 Using a single command, find the information about your current display and system language.

Problem 5.11 Using only single line (commands can be chained), list the processes in descending order of memory usage.

Chapter 6

Installation of Python and Jupyter Lab

Computers are useless. They can only give you answers.

Pablo Picasso

We will install Python 3. The previous version, Python 2, has been officially deprecated and it is no longer maintained. While Python 3 and Python 2 are compatible to a large extent, there are differences and incompatibilities. This should be kept in mind especially when searching for information from the internet or when using books that provide examples using Python 2.

6.1 Installation

Independent of the operating system, there are two main options:

1. `Anaconda` (or `Conda`)-based installation and
2. `pip` (Python Package Installer).

Both are good choices but there are differences. `Conda` installation gives immediately a very large number of packages including the most relevant ones needed in this course. It is a very convenient choice and the preferred one here. Anaconda has a both a graphical user interface as well as command line (`conda`). In addition, both Jupyter Lab and Jupyter Notebook are installed automatically with `Anaconda`. `pip`, however, gets new packages usually faster but all packages must be separately installed and there is no graphical user interface. This also means that Jupyter must be installed separately. With pip one also has to be more careful with package conflicts and it is generally a good idea to use so called virtual environment to avoid problems.

In brief, `Anaconda` is easier. In addition to Python, Jupyter Notebook and Lab will be installed immediately. Anaconda is available for Windows, macOS and Linux. `pip` may be your choice if you have already experience with Python but `Anaconda` is recommended for the current purposes.

Note regarding Windows vs WSL: The `Anaconda` installer works great in Windows and it also works in WSL through the Linux installation process. It may be more convenient to use the Anaconda installer in Windows, although that is a matter of personal choice. In addition, WSL has a basic Python installation present (not the full `Anaconda` distribution, though) in case needed.

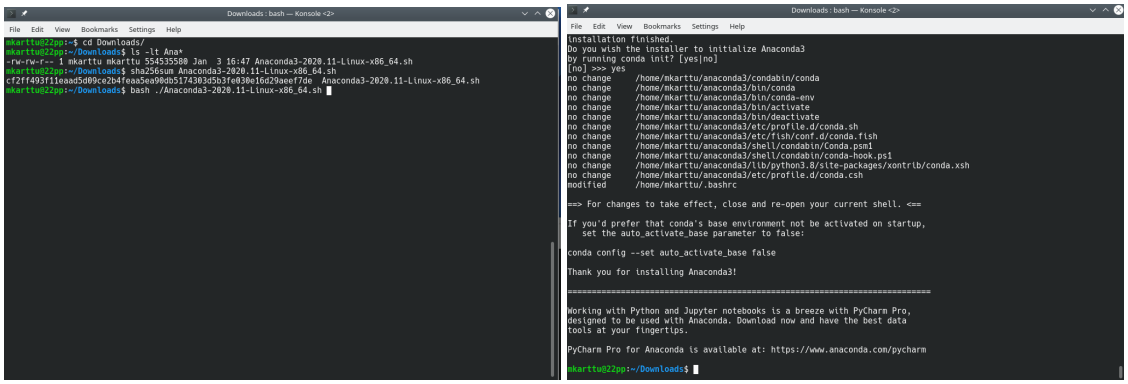


Figure 6.1: Screenshots from the beginning and the end of Python installation using the Anaconda graphical installer.

6.1.1 Anaconda in WSL and Linux

With Linux and WSL you already have Python installed. It may be, however, more convenient to install Anaconda as it comes with almost all possible necessary tools. pip installation is also provided below as one of the options.

This is the recommended way.

1. Go to the Anaconda Installation page
2. The page lists the pre-requisites that need to be installed. If you are using an Ubuntu based installation, simply open terminal window and copy to install the dependencies. Here's also how you can install the pre-requisites:

```
apt-get install libgl1-mesa-glx libegl1-mesa libxrandr2 libxrandr2 libxss1 libxcursor1 libxcomposite1
     libasound2 libxi6 libxtst6
```

3. Then download the Anaconda Installer from Anaconda Installation page and follow the instructions.
4. As the last step of the installation, verify your installation. Do not skip this step!
5. For this step, open a new terminal and type `conda list`. If you get a list of routines, your installation is complete.
6. See screenshots from the installation process.
7. Note that Jupyter Lab and Jupyter Notebook are now present

6.1.2 pip in WSL and Linux

6.1.3 Anaconda in Windows

On Windows 10/11 the easiest option is to install Anaconda. Note that this installs Anaconda on the Windows side only.

1. Go to the Anaconda Installation page
2. Select the option for Windows and follow the instructions on the page.
3. The installation will provide you with the option to install PyCharm. It is optional, decide based on your needs.
4. As the last step of the installation, verify your installation. Do not skip this step!
5. After verification, open the Anaconda Navigator from the menu.

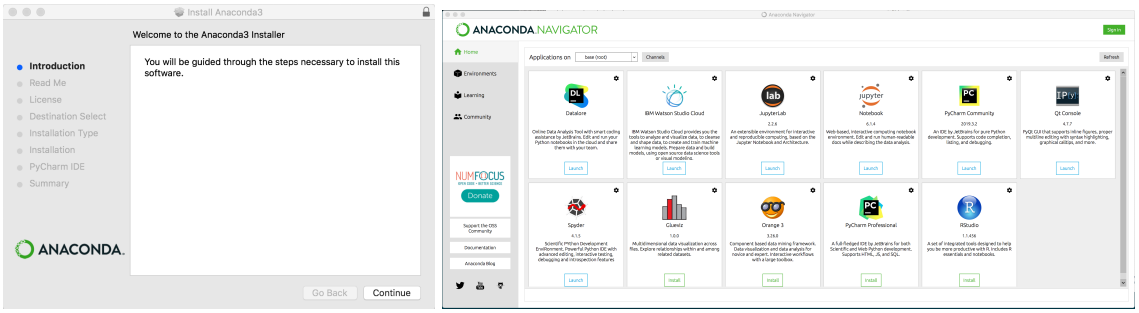


Figure 6.2: Screenshots from the beginning and the end of Python installation using the `Anaconda` graphical installer.

6.1.4 `Anaconda` on macOS

Recommended: `Anaconda` graphical installer

See also Figure 6.2.

1. Go to the `Anaconda` Installation page
2. Select the option for macOS and follow the instructions on the page.
3. As the last step of the installation, verify your installation. Do not skip this step!
4. After verification, open the Anaconda Navigator from the menu.

Option: `Anaconda` using command line

If, for some reason, the graphical installer doesn't appeal to you, it is also possible to install `Anaconda` using command line:

- The `Anaconda` Installation page offers also the option to install using command line. If that is what you prefer, then choose this option instead of the graphical installer.

Option: Using the distribution from the Python home page

`Anaconda` is recommended, but this is also an option:

- Download from the Python Home page and follow the instructions.

Option: Pip using `Homebrew`

It is possible to install Python, `pip` and other tools using `Homebrew`. We will not cover this here, Anaconda installation is recommended.

- Homebrew and Python

6.1.5 Keep Python up-to-date

Updating all packages is very easy (these commands need to be given in the command line terminal),

```
conda update --all
```

To update conda:

```
conda update conda
```

Install packages:

```
conda install <package name>
```

6.2 Python virtual environments

Important:

Using virtual environments is highly recommended. Using them eliminates lots of potential errors, and problems and version conflicts with package updates.

There is a vast number of python packages both distributed with the basic installation and contributed by the users and various projects. When packages are updated or/and changed, compatibility may break. That is not a problem between packages that belong to the main distribution but some of the contributed ones, including ones that may be your own ones, may also need to be modified to be compatible with the new distribution. The solution to this is the so-called virtual environment.

Using virtual environments is highly recommended. Using them eliminates lots of potential errors and problems with package updates.

The idea of a virtual environment is to provide a sandbox - an isolated environment - for a project. This allows the project to have its own dependencies without interfering with other projects. Here is a practical example from the context of molecular simualtions and data analysis (this was the case in Nov 2020 but may have changed since then): The widely used molecular simulations analysis package MDAnalysis requires a specific version of the very popular plotting package Matplotlib. The latest Matplotlib version (=the the latest at the time of writing this) leads to some unexpected errors. Using virtual environments helps to circumvent the problem since creating an environment specific for MDAnalysis allows it to have its own version of Matplotlib. This type of isses are quite common and using virtual environments helps to avoid problems - tracking such problems can sometimes be quite difficult.

In the example above, we'd simply created a separate virtual environment. This could be extended to any other situations. For example, let's assume that we two projects, ProjectA and ProjectB. Let's further assume that they have different and possibly conflicting dependencies. We simply need to create separete virtual environments for both ProjectA and ProjectB, and are good to go.

There are no limits to the number of environments.

6.3 Installation of virtual environments

Here's how to create virtual environments with `conda` using the command line terminal. This works in Linux, WSL and Mac. Mac and Windows have also the Anaconda GUI but using command line is quicker. Note that `conda` stores the information about environments in `$HOME/anaconda3/envs/`. In practise that means that environments can activated and deactivated very conveniently (this is different from `pip`).

6.3.1 Virtual environments using `conda` and command line

1. Check the version and (possible) existing environments (command line):

```
conda -V
```

2. Check if `conda` needs to be updated:

```
conda update conda
```

3. List all of your existing `conda` environments:

```
conda env list
```

4. If you want to see the list of packages installed in your current virtual environment:

```
conda list
```

5. Create a new virtual environment. To use your current version of Python:

```
conda create -n my_new_environment
```

6. Alternatively, if you want to use some specific version of Python, instead use

```
conda create -n my_new_environment python=x.y
```

   and replace `x.y` by the version number accordingly
7. Activate the new virtual environment

```
conda activate  my_new_environment
```

   After this, the name of your new environment will appear in front of the prompt on your terminal.
8. Install the packages that you want for the environment called `my_new_environment`

```
conda install -n my_new_environment <package name>
```

   Note: the option `-n` specifies the name of the environment. That allows for easy installation of packages into any environment. Leaving `-n` out installs in the current environment.
9. Deactivate the virtual environment

```
conda deactivate
```

   After this, the text base will appear in front of the command prompt on your terminal.

If/when you want to remove a virtual environment:

```
conda env remove -n  my_new_environment
```

6.3.2 Virtual environments Anaconda Navigator

1. Open the Anaconda Navigator
2. Click on the tab named 'Environments'
3. Click on Create at the bottom of the Navigator menu

6.3.3 Virtual environments using `pip` and command line

`pip` works a bit differently, but using it is equally easy. First, install the package `virtualenv` using

```
python3 -m pip install --user virtualenv
```

This gives `venv` for setting up virtual environments.

There is one important thing to notice here and this is where `pip` is different from `conda`. Using

pip creates a virtual environment and its directories at the location where you currently are. In the example below, the creation of the new environment that is called my_new_env means that a directory called my_new_env is created at the current location. This is less convenient than with conda. You can activate a virtual environment from any location but with pip, you need to know where it is located.

- The command to create a new virtual environment (we call it here my_new_env):

```
python3 -m venv my_new_env
```

- After its creation, the environment must be activated by the command source

```
source my_new_env/bin/activate
```

- To check that the virtual environment is indeed activated, type

```
which python
```

- Deactivation is easier: Independent of your location, simply type

```
deactivate
```

6.4 Installation of Jupyter Notebook / Jupyter Lab

Jupyter Notebook/Lab can be described as a web browser-based document that has the ability to combine live code with plotting, interactions, text and multimedia. In addition to Python, it can handle many other programming languages as well. It is better to call it a platform or an environment rather than a document since it provides an interactive browser-based environment for scientific computing, analysis and documentation. Jupyter notebooks have the file extension .ipynb. The files are in ASCII format and do not depend on the operating system one uses. Importantly, Jupyter notebooks provide the possibility to create live lab books that are reproducible. Jupyter notebooks are used extensively in a broad range of fields such as analysis of simulation data, machine learning, digital humanities, meteorology, and so on.

The applications that can open, execute and save Jupyter notebooks are called Jupyter Notebook and Jupyter Lab. The former is older but has more extensions for handling different types of data.

There is also Jupyter Book that uses markdown and Jupyter notebooks to create live textbooks that can even execute code. This document was made using Jupyter Book. There are also several other related projects, see the Project Jupyter page. Jupyter is open source software.

6.4.1 Jupyter Lab or Notebook?

There are two applications that can open Jupyter notebook (.ipynb) files, Jupyter Notebook and Jupyter Lab. Below are some of the main differences between the two:

- Both of them run in a web browser.
- Jupyter Lab is newer. Due that, not all the extensions that work with Jupyter Notebook work with Jupyter Lab. This is, however, changing rapidly.
- The user interface of Jupyter Lab is very versatile. One can open simultaneously many notebook (.ipynb) files, even terminal windows and stand-alone Python (or other programming language) windows.
- With Jupyter Notebook one works with a single notebook (.ipynb) file at a time.

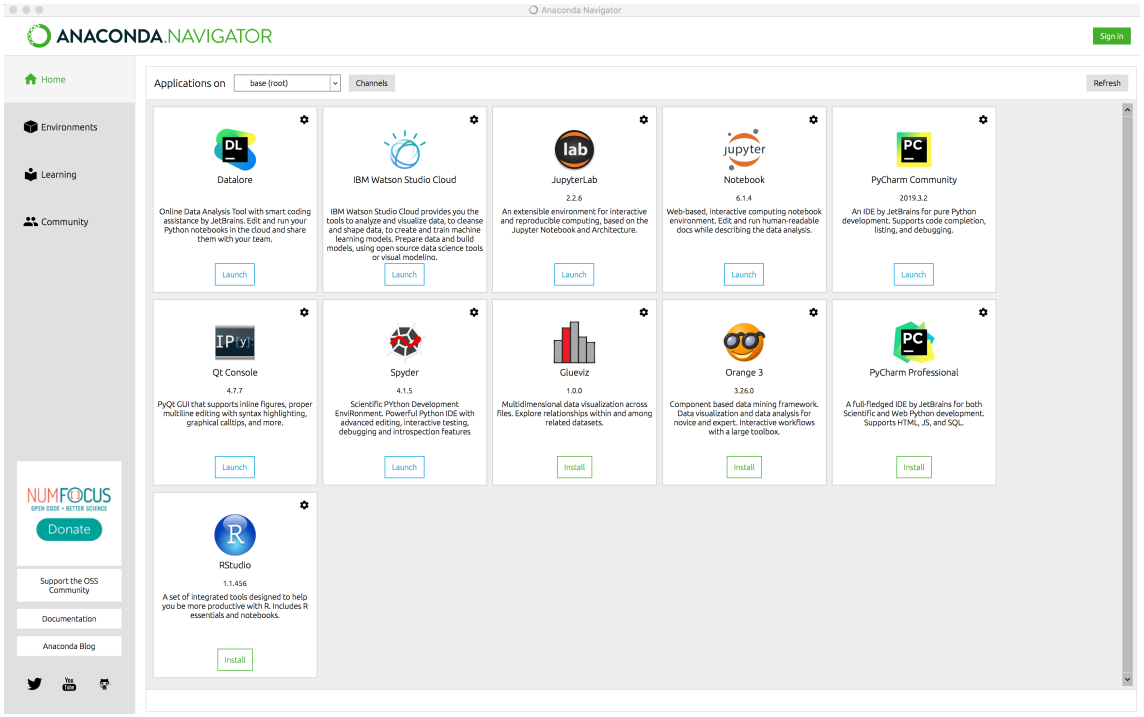


Figure 6.3: Anaconda Navigator in Mac and Windows. Simply clicking `Launch` on Jupyter Lab or Jupyter Notebook starts it. Similarly, one can start any of the other applications that come bundled with Anaconda.

So which one to choose? Here, we prefer Jupyter Lab. However, everything works in Jupyter Notebook as well so the choice is yours. For the advanced users, Jupyter Notebook has some features that are not yet available on Jupyter Lab such as the use of variables in markdown cells. We won't be using such features here and it is highly likely that they get implemented in Jupyter Lab as well.

6.4.2 How to start Jupyter Notebook / Lab?

Independent of your operating system, launching Jupyter Lab (we'll use Jupyter Lab from now on) opens a new web browser or tab using your default browser.

On Windows and Mac

Simply run the Anaconda Navigator from your program menu. The interface, see Figure 6.3, is the same in both cases.

On Linux and WSL

On Linux and WSL/WSL2, Jupyter Lab and Jupyter Notebook must be started on the command line. The following commands start Jupyter Lab and Jupyter Notebook, respectively.

```
jupyter-lab
jupyter-notebook
```

If needed, you can get help with the following commands:

```
jupyter-lab --help
jupyter-notebook --help
```

6.4.3 How to update Jupyter Lab

On command line, check the version you have:

```
jupyter-lab --version
```

With `conda`

There are two options:

1. Updating for the base environment (=no virtual environment activated)

   ```
   conda update jupyterlab
   ```

2. Updating inside a virtual environment: First, activate the environment and then run the update:

   ```
   conda avtivate my_virtual_environment
   conda update jupyterlab
   ```

With `pip`

```
pip install --upgrade jupyterlab
```

6.4.4 Jupyter interface

The initial web browser window is identical in all operating systems. Note that the Jupyter window (or tab) is not connected to internet. Instead, it runs *locally* on your computer. The figure below shows the initial browser window when starting Jupyter Lab. There are a few points to here:

The web address: `localhost:8888/lab`. This is a *local address*, this means that the application (Jupyter Lab) is running on your computer (`localhost`) and connected to `port 8888`. The window has tabs on the left hand side for listing files, extensions, help etc. The top menus offer functionality such as starting and stopping kernels, running selected cells and so on. The icons on top of the left hand side pane have the options to open new Launchers, create new folders, upload files and refresh the current list. In the figure below, no files are present in the current directory (the list is empty).

The initial Launcher Window in a web browser is shown in Figure 6.4:

6.4.5 Code and markdown cells

Clicking the Python 3 notebook icon opens the Jupyter workspace window. It is the same in all operating systems. The figure below shows some of the important features.

One of the very useful features on Jupyter notebooks is that there are different types of cells (the white area inside the blue rectangle). The two basic types are markdown cells and code cells, see Figure 6.5. The cell type can be easily changed from the dropdown menu as indicated in the figure. This separation makes it very convenient and easy to write interactive, reproducible and well-documented notebooks.

Code cells

As the name suggests, these are the cells in which one writes (live) code. Here, we use Python 3 and that is also clearly indicated in the workspace. Once code is written, it has to be executed. One can

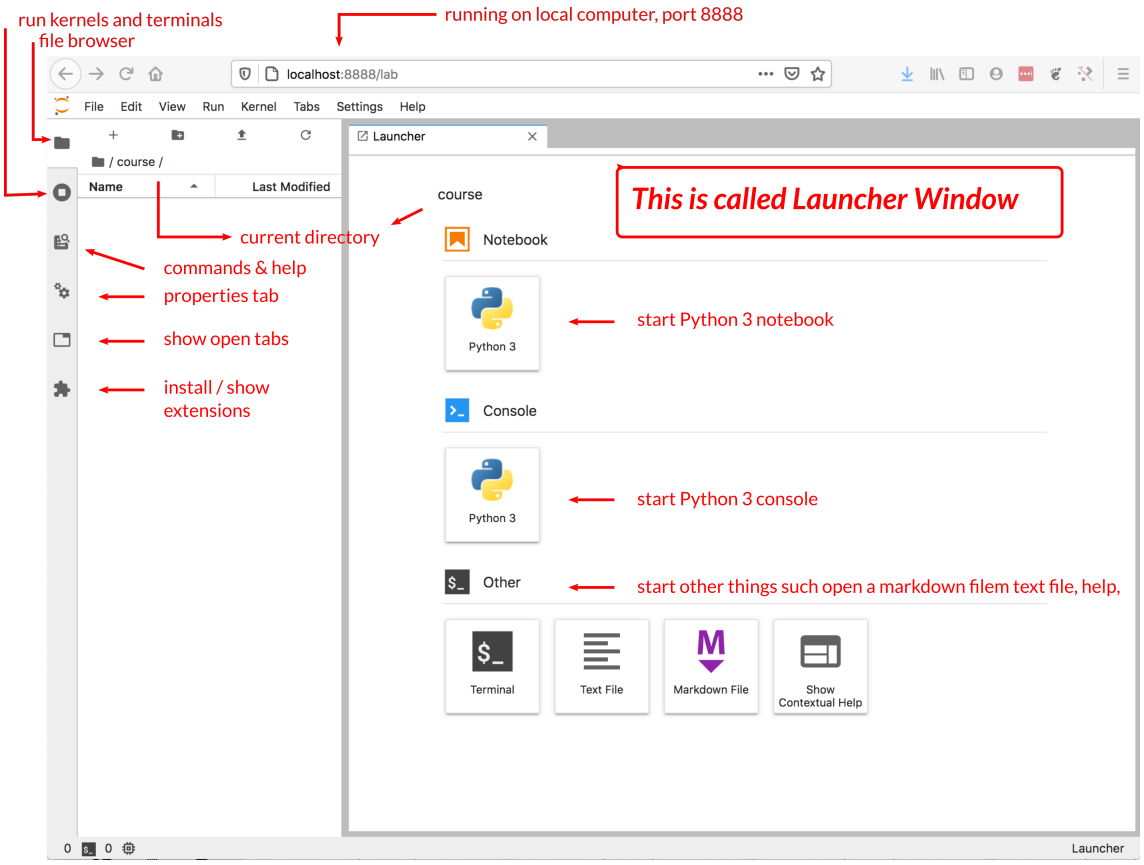


Figure 6.4: The initial Launcher Window in a web browser. This is the same in all operating systems.

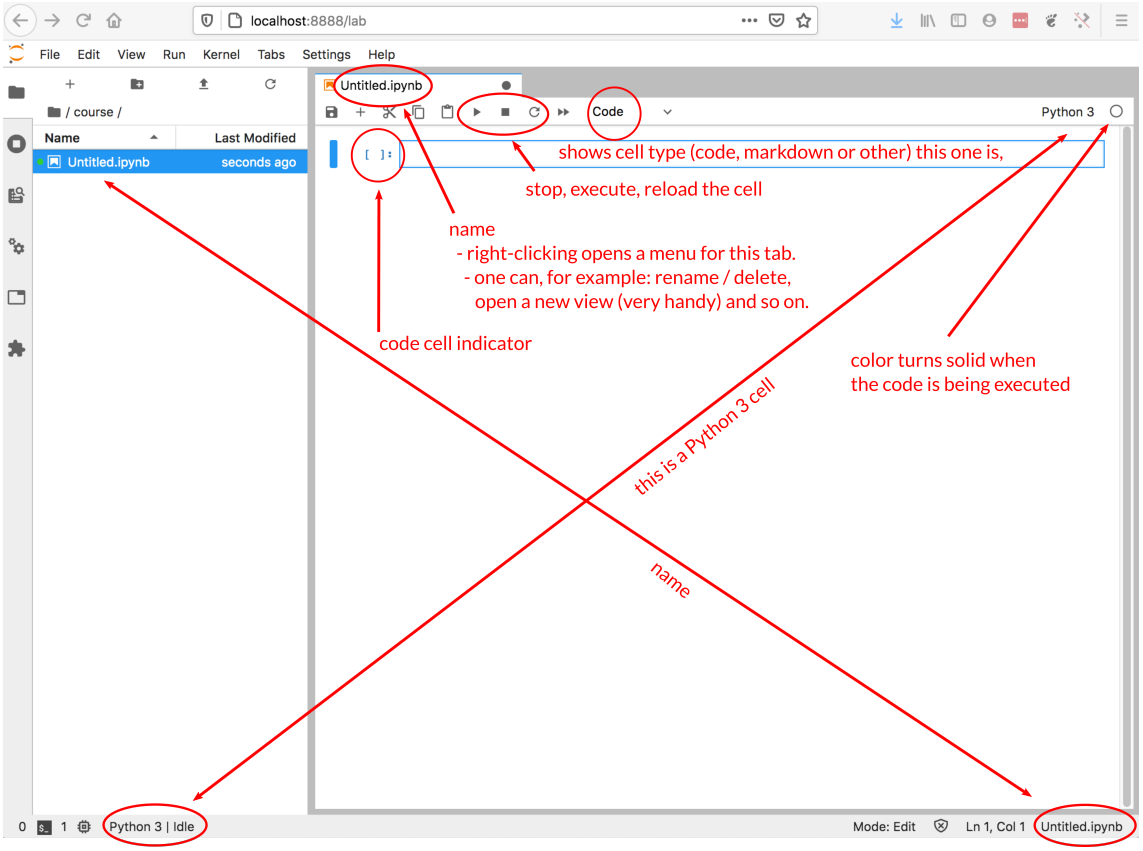


Figure 6.5: Jupyter Lab window in a web browser with the most important features indicated.

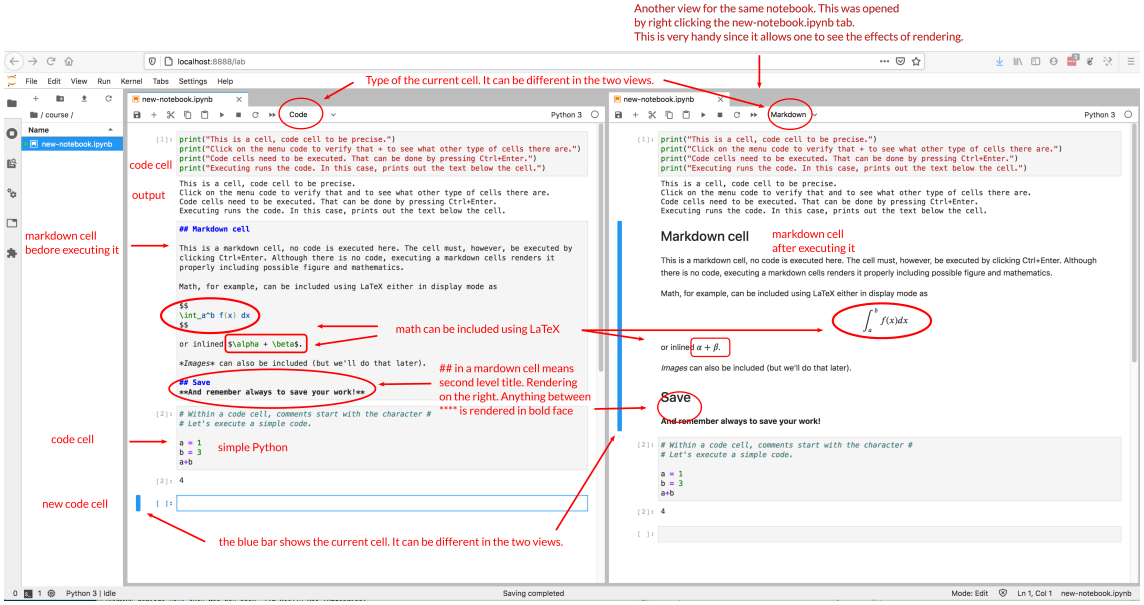


Figure 6.6: Jupyter Lab window showing both code and markdown with LaTeXincluded.

use the icons or the 'Run' and 'Kernel' tabs. The 'Kernel' tab contains several very useful options including 'Restart Kernel and Clear All Outputs'. This is very useful if one wants to make sure that everything is consistent (remember, the cells are sequential and sometimes cell-wise execution of commands using different variables may lead to problems).

Code cells can also be executed by clicking `Ctrl-Enter` (this does not advance the cursor to the next cell) or `Shift-Enter` (advances the cursor to the next cell).

Markdown cells

Markdown means that one can enter normal text, but there is no executable code. *Markdown* is language that is simple and quick to use for documentation (link to the preference is provided in the header). It allows for titles, lists, figures, mathematics and so on. Even markdown cells must be executed for the text (and other markup such as mathematics) is rendered properly. Execution works just like for code cells.

Below is a figure that demonstrates some of the above matters and also the multi-window feature of Jupyter Lab (if it is too small, click on it to see it full size).

6.4.6 Remember to save your work

There is much more to the Jupyter notebooks (see the links in the header) but the above provides the fundamentals. When working with them, *always remember to save your work.*

Chapter 7

On computation and codes

Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.

B.W. Kernighan & P.J. Plauger in The Elements of Programming Style.

Before we start discussing computing, modelling and implementing the different algorithms and methods, let us briefly list a few general issues on design, programming, computing and codes. The remarks below do not depend on the programming language.

7.1 Well-written programs: Desirable properties

Independent of the chosen programming language, there are several matters that should be considered. It is impossible to write an exhaustive checklist, but the purpose of the short lists below is to bring some of the important issues to the reader's attention. In all programming, one should be careful and pay attention to detail, and remember debug, debug, and debug. Some of the desirable properties for any code are

- **Portability:** Codes should be written in such a way that they are portable, i.e., they must work correctly on different computer architectures.
- **Readability:** A good code should include comments and avoid unnecessary tricks.
- **Simplicity:** The purpose is to solve problems, not to write the most beautiful code.
- **Parallelizability:** For a code that requires long run times, parallelizability may be important to reduce execution time. Here, we will not focus on that issue, though.

7.2 Programming languages

There are many, many programming languages. There is no ultimate language, but one has to choose the language based on pragmatic arguments. However, Python has become a very popular language for a great number of tasks from data analysis to banking and to scientific computing,

and C/C++ has become the most common general purpose language. Fortran is still being used for computationally intensive tasks - that is the purpose that it was originally designed for (Fortran = FORmula TRANslation). For web-based solutions, javascript is extremely common.

7.3 A few notes on coding and related matters

Perhaps to most important piece of advice is the following quote:

> *First, solve the problem. Then, write the code.*
>
> —John Johnson

In brief, the above is the essence of good design: Do your background work, solve the problem, find the important determinants and constraints, and when possible find analytical solutions that you can use as your benchmarks. If your code fails to reproduce them, either your program, your algorithm or/and your general approach to solve it is/are wrong. Find what is essential to your problem, do not try to include everything possible (=avoid irrelevant details): If your task is to optimize the wind resistance of a new car model, its shape is the major factor. Including color options or the interior have no role and should not be considered. As another example, consider a database for course grades: The essential information consists of names, students numbers, grades and comparable, but including details such as hair color, height and weight are irrelevant. Apply the

- KISS (keep it simple, stupid) and
- YAGNI (your aren't gonna need it)

principles.

Good design consists of finding the essential parameters and variables and avoiding unnecessary complexity. Beginning programmers and computer modellers often try to include too Remember: Garbage in, garbage out (GIGO).

Below are some practical hints to speed up programming and design:

- **Pseudocode:** Write out the details of the algorithm you wish to implement as pseudocode. It makes tracing of errors much easier and programming much faster. Pseudocode is independent of the chosen programming language.
- **Divide the problem to smaller parts:** When possible, divide the problem to clear smaller parts and test those parts separately.
- **Modularize:** Write subroutines (=subprograms) for self-contained routines. These subroutines are then called by the main program to perform their tasks. These subroutines may be tested and verified individually, which is a great advantage.
- **Clarity:** In general, try to program in a clear manner avoiding unnecessary complications.
- **Variable names:** Use descriptive variable names. It makes your code much for readable. Do not recycle variables which have been named in a descriptive fashion as that will make following the code really hard. You can (and should) easily recycle variables such as loop indices (i, j, k). It is often a good idea to call temporary variables as temp or something like that.
- **Comments:** Include generous and precise comments in your code. What does this subroutine do? What are the main variables? Have any tricks been applied? It is also often clearer to comment blocks of code and avoid line-by-line commenting as it decreases readability.
- **Status:** Mark the status of your codes clearly. Is the program, subroutine or function fully functional or tested. How has it been tested? When? By whom?

- **Print:** When writing and testing different parts of the code, print out results, also intermediate results, as they can be used later for comparison and to help to debug the code.
- **Keep a lab-book:** Keep notes when you program and test your codes. Print out the results, and keep a diary - a lab-book - in which you detail what kind of changes you did and what was the reason for those changes or corrections and their effect. Proper logging will save enormous amount of time afterwards.
- **Debug: Debug, debug, debug, and even after that, debug.** It is often quicker to write the first version of a code than to debug it. Even the best of programmers do, however, produce certain number of errors. It has been quoted that the 'industry average' for professional programmers is 15-50 errors per 1,000 lines of code. See, for example, Rampant software errors may undermine scientific results by David A.W. Soergel
- **Test cases:** Test your code against known results whenever possible. In particular, pay attention to boundary conditions as they tend to be a common source for a lot of errors,
- **Program libraries:** Use program libraries when possible. They have usually been tested tested by numerous users, and typically optimized for performance and accuracy. Keep in mind, though, that using them is not always straightforward (e.g., wrapping in the case FFTs is sometimes somewhat involved) and one should always perform one's own tests and verifications as even the libraries may contain errors.

7.4 A few additional details for efficient programming

In addition to the above design issues, here are a few often overlooked practical issues

Summations: One can minimize roundoff errors if summations are performed in the order of increasing magnitude (add smaller numbers first).

Avoid mixing of types: Do not add floating point numbers and integers. Instead, use a type conversion (functions such as REAL in Fortran) when performing such an operation. REAL(i) converts an integer i into a floating point number and it can then be used in operations with other floating point numbers. Although Python does not have type declarations, this issue still matters: For example, consider the difference between 1/3 and 1.0/3.0. Python 2 (no longer maintained but lots of libraries and codes are still using Python 2) and Python 3 handle these expression differently!

Memory access: Make sure you access computer memory in the most efficient way. For example, in the case nested loops, the first index runs fastest in Fortran, whereas in C/C++ and Python it is the last one. This can make a huge difference in performance.

Chapter 8

Introduction to Python

There are only two kinds of programming languages: those people always bitch about and those nobody uses.

Bjarne Stroustrup, developer of C++

Since its introduction in 1998, Python has become one of the word's most popular programming languages and often required for jobs that involve any programming.

Python is modern a interpreted language, as opposed to compiled programming languages such as C, C++, Fortran, Pascal, and so on. Being an interpreted language means that a Python interpreter reads the code line-by-line as it is executed. In contrast, a compiled language has to be run through a compiler to produce an executable program. The compiled executable works only on the platform where it was compiled, but a code that is interpreted is more portable between different platforms.

Python was originally created by Guido van Rossum, a Dutch programmer. The first Python implementation was done in December 1989 by van Rossum – to keep himself busy, as he says. Current version of Python is 3 and, importantly, Python 2 has been deprecated as of 2020. Importantly, Python 3 is not fully backward compatible with Python 2.

Figure 8.1: Guido van Rossum.

Below, some properties of Python are listed. If they do not sound familiar, do not worry as we will learn them along the course.

Properties:

- General purpose and Jupyter provides interactivity, free and open source
- *Dynamically typed* (as opposed to *statically typed*). This means that one doesn't have to declare variables.
- Object oriented
- Portable (Linux, Unix, macOS, Windows, mobile phones...)
- Versatile. Python is used in an enormous number of different applications, commercially, in research and in industry
- Easy to learn. This is indeed the case! The barrier to start using Python

is very low.
- Extensible, it uses modules. The number of available modules is very large.

Disadvantages:

- Speed can be an issue since Python is an interpreted language. However, there are efficient libraries and the possibility use tools such as Cython and even GPUs.

Famous examples/applications/users:

- YouTube, reddit, Yahoo, Google (Gmail, Groups, Maps), CERN, NASA
- Part of the core components in Linux distributions
- Information security industry
- Applications such as Abaqus, Gimp, Inkscape, Spotify...

Huge number of modules and libraries. Examples:

- SciPy, Biopython, `numpy`, `matplotlib`, Sage, Tensorflow, Chempy, MDAnalysis, pyEMMA, RDKit, nglview, DeepChem

Used a scripting language in:

- Blender, Gimp, Inkscape, Totem, SPSS, PyMOL

Other:

- Jython: compiler that produces Java byte code from a Python code
- PyS60: Symbian [obsolete now] phones
- Python for Android

It is hard to tell exactly how popular Python is (since it is not easy to define a good metric), but in both StackExchange and Github it ranks among the most popular ones. When it comes to programming and programming languages, it is good to keep in mind the following quotation:

8.1 Python basics – the necessary prerequisites

Let's get into Python. As mentioned above, it is very easy to start using it. We have to, however, understand a few basic things. The aim here is not programming in a broad sense but rather to use Python for various simple tasks. Topic such as object oriented programming will not be covered, the emphasis is put on simple data analysis an providing code snippets that can be re-used, for example, in analyses of experimental and computational data, in producing high quality plots for manuscripts, theses, talks and posters, and in using modules and packages for tasks such as Markov analysis and machine learning. This part gives a very brief introduction and jump start to Python. That is done in practise in mind: This section shows some plotting techniques for data generated inside Python as well as for data that is read from a file/files. A few of the most important modules are briefly introduced: `matplotlib` and `seaborn` for plotting, `numpy` for handling array data and pandas for statistical analyses. More will introduced as we progress. One important point is that Python is a very versatile language and the number built-in and contributed modules is enormous. It is impossible to know all, or even a reasonable fraction, of them. Instead, it is very important to know how to search for information and how to use it.

Methods are common and their applicability spans throughout the fields. For example, the Monte Carlo method is used to optimize public transportation schedules, design of integrated circuits, used in modeling quantum phenomena, to mention a few applications. Similarly, machine learning is

used in everything from image processing to analyzing protein conformations and optimization methods are used in virtually all fields.

8.2 How to use the rest of this Chapter

The best way to learn is to try out the code snippets. That can be done in Jupyter notebook (that is the assumption here) or using `spyder` or such, or even plain Python terminal. When using Jupyter Lab, one should be aware that once a kernel has been executed, the definitions become available throughout the notebook and that is sometimes a bit deceiving for tracking dependencies. In such a case, interrupting and re-running the kernel should clarify the situation.

Some execution cells have pieces that are commented out. In some cases that is done to avoid long outputs, but it is very good to try them out.

Since we have already installed and open Jupyter Lab, let's do that again. Open Jupyter Lab from your program menu or command line depending which operating system or/and installation method you used. The code snippets below are to be executed in Jupyter Lab.

8.3 Install Python virtual environment

It is a very good idea to use Python virtual environments. See Section 6.3 if you have not already installed virtual environments.

8.4 Getting help

At this point you should have Jupyter Lab open and a notebook running in it. Remember to give the notebook a name and save it regularly.

Below is an easy way to get help. We will elaborate on it later.

```
# Important: Try the command in the Code cells in your own notebook.
# The hashtag (#) character starts a comment.
# Note that if a comment is put in the middle of a Code cell, everything that is after it
# is ignored by the interpreter.

help()
```

8.5 Variables and data types

Below, we will discuss the basics of variables in Python. The intention is not to cover all aspects, but enough to get going and to be able to use the concepts efficiently. Note: a good way to name your variables is to use descriptive variable names. There are some restrictions on variable naming and those restrictions will be discussed below.

Unlike the usual programming languages such as C, C++ and Fortran, Python is dynamically typed. In statically typed languages such as C/C++ and Fortran, the types variables must be known and fixed and cannot be changed. In dynamically typed languages such as Python and javascript that is not the case. This means that in addition to the value, the type of a variable may be changed. The second notable difference to languages such as C/C++ is that in Python one not have does declare variables. The type and value of a variable are assigned when the variable used the first time. This is best illustrated by an example (the variable names are arbitrary, the part that indicates the variable type is used for clarity):

```
a_int        = 2                                    # integer
a_float      = 2.0                                  # float
a_string     = 'Hello world'                        # string
a_tuple      = ('Learning','Python',(24,1,2023))    # tuple
a_list       = [1.0, 2.0, 3.0]                      # list
a_dictionary = {'Monday':'lundi','Tuesday':'mardi'} # dictionary
```

Note also that variables are assigned by using the assignment operator `=`. In logical comparisons (True/False), the comparison operator `==` is used.

To see the value of the variable, we must use the command `print` (if you have used Python 2 before or searched help from the net, notice that in Python 3 brackets are required):

```
print(a_int,a_float)
print(a_string)
print(a_tuple)
print(a_list)
print(a_dictionary)
```

You can also just type the variable to see its value (but only one at a time):

```
a_string
```

Here are a few things to notice:

- Hashtag starts a comment.
- Alignment at the beginning of the line matters.Try shifting any of the lines above by one and you will get an error message.
- The value of the variable is not echoed (=printed on screen) after it has been assigned. - The above are the most commonly used data types in Python. Python has many other useful data types including Fortran-like intrinsic type for complex numbers. Below is a list of some of them with an example of how they are used. We will discuss some of them in more detail in a moment. More information is available at the Python web page.

8.5.1 Basic variable types:

1. integer: `a_int = 1`
2. float: `a_float = 1.0`
3. complex: `a_complex = 1.0+1.0j`
4. string: `a_string = 'barley'`

```
a_int = 1
a_float = 1.0
a_complex = 1.0+1.0j
a_string = 'barley'
```

8.5.2 Compound variable types:

1. list: `a_list = [1.0,2.0,'hops']`
2. tuple: `a_tuple = (1.0,2.0,'hops')`
3. dictionary: `a_dictionary = 'hops1':'saaz','hops2':'spalt'`

```
a_list = [1.0,2.0,'hops']
a_tuple = (1.0,2.0,'hops')
a_dictionary = {'hops1':'saaz','hops2':'spalt'}
```

Execute the above variable assignments by putting them to a code cell in Jupyter. If you do that and print them out, you will notice that the variables have changed since their first definition above.

```
print(a_int,a_float)
print(a_string)
print(a_tuple)
print(a_list)
print(a_dictionary)
```

As you notice, tuple, list and dictionary use different brackets and for the type dictionary the syntax relates the two values separated by a semicolon and comma separates list items. We will discuss these data types in more detail below.

8.5.3 Mutable and immutable data types:

There is one important issue that should be mentioned here: Data types are divided into *mutable* and *immutable*. The content of a mutable variable can be changed. Dictionary and list are *mutable data types*. String, integer, float and tuple are *immutable data types*. The key to understanding the difference is to understand that everything in Python is an object. We will discuss this issue with examples when we progress.

8.6 Keywords, variable names and first touch of import

Python is very flexible with variable names but there are some restrictions. Here is a brief summary:

- allowed: variable names can be of arbitrary length
- allowed: both letters and numbers
- allowed: the character underscore (_)
- not allowed: the character @
- not allowed: spaces
- distinction: there is a distinction (like in C/C++) between upper and lower case. Both are allowed.
- restriction: variable names must start with letter
- not allowed: variable names cannot be one of the keywords (see below)

There are recommendations for good programming style in Python.

We can import the module keyword as given below and check the. There are an enormous amount of modules for various purposes. This import mechanism makes Python extremely flexible. We will use this mechanism extensively and discuss matters as they arise. For now, take this as an example how to import a module and how to call them. keyword is a built-in module. Modules typically contain functions or methods and they called using the dot operator (.). Below, we import the module keyword and then call the function kwlist from it.

import: the command import provides a mechanism to import or bring in libraries, modules, packages or your own code that is in a separate file. This is very powerful and it is the main mechanism for bringing in libraries and modules to your code. In the next cell, we use the command import keyword to import module called keyword. That module provides the list of the reserved Python keywords. It also provides a mechanism to check if a word belongs to reserved Python keywords. There are also other ways for importing but the command import is the most common one.

```
## These are Python keywords that are not allowed as variable names.

import keyword                                          # import keyword module. It is a built-in one.
print (keyword.kwlist)                                  # Print out the list of reserved keywords.
print ('The number of keywords:', len(keyword.kwlist))  # How many keywords are there?.
print ('The keywords are: \n',type(keyword.kwlist))     # Just for curiosity, what the data type?
```

```
test_word = "coffee"                                    # define a variable for testing
print("Is",test_word,"a Python keyword?", keyword.iskeyword(test_word))
```

One question that may come to mind is: How did you know the commands `keyword.kwlist` and `keyword.iskeyword`? The help command tells it. In addition, when called a function from a module, one needs to use the name of the module like we did above (for example: `keyword.kwlist`).

Some other obvious questions are: 1) what are the built-in functions, 2) how to know what the methods associated with each of the modules are and 3) how does one get more modules. There is help. Uncomment the ones below one-by-one and execute

```
## Uncomment one-by-one or run these in different cells

#help("keyword")     # help on the module 'keyword'. Try also help("keywords")
#help("builtins")    # help on build-ins
#help("modules")     # this will list all the modules that are installed (can be massive)
```

8.7 Integers & floats

Integers are the simplest data type. Let's assign some integers (when the decimal point is not given, the type is defined as an integer), change the value of one of them and then perform a simple division:

```
a=2
b=4
c=3
print(a,b,c)
```

```
a   = 7
d   = a/c
dd  = a%c
ddd = a//c
print(a,d,dd,ddd)
```

The above seems quite mundane, but there is one important matter: The variable `d` is formed by dividing `a` by `c`. Both `a` and `c` are integers but `d` is a float. This may not see remarkable but in Python 2 as well as many in some other languages division of two integers (modulo operation) gives an integer. In Python 3 this is not the case and this can cause some trouble when converting from old Python 2 code to Python 3: In Python 2, Fortran and C/C++, the division `a/c` would yield 2 instead of 2.3333. The operator `%` gives the remainder and the operator `//` takes the modulo of the quotient. Note also the following:

```
aa = a+1
print(aa, type(aa))
bb = aa+1.0
print(aa, type(aa),type(bb))
```

Here we have printed out the *value* and *type*. Adding and integer to an integer yields an integer but adding a float to an integer yields a float. As discussed above, Python is *dynamically typed* and this is an example of *type reassignment*. This is not possible in C, C++ or Fortran; with those programming languages you may or may not get an error message during compilation (depending on your compiler).

8.8 Strings and the slicing operator

Strings are, as the name suggests, strings of characters. In Python, they defined by single quotes (double quotes work). Note that if a number is within a string Python sees it as a character (text) and not as a number. Recall the string variable we used above:

```
a_string = 'barley'    # One can use either simple or double quotes
print(a_string)
print(type(a_string))
```

We can add strings together, or concatenate them, using the operator +:

```
a_string = a_string + ' is important for brewing'
print(a_string)
```

Note also that that if you execute the cell several times, it keeps adding to itself (try it out, just re-execute the Jupyter cell for a few times. Let's re-assign the variable (in case you did re-execute the Jupyter cell). This is how to change line (notice the space or lack of it after \n:

```
a_string = 'barley'
a_string = a_string + '\nis important for brewing' + '\n and hops are needed too'
print(a_string)
```

Since quotations are used to define strings, one needs to escape them if one wants to use quotation marks as part of the string:

```
b_string = "Barley is \"good\" for you."
print(b_string)
```

We can also access elements of a string very easily by calling the index. Like in C/C++ (and unlike in Fortran), indexing starts from zero. To call the second element,

```
print(a_string[1])
```

Slicing:

There is more to strings: We can use the slicing operator to access parts of a string. The syntax is

```
varname[start:stop:step]
```

Let's try this:

```
print('Original string:', a_string,'\n\n')
print('Sliced string:',a_string[0:9],'\n')
print('Sliced string with a step of two:',a_string[0:9:2])
```

Slicing is very useful in many tasks. Here's a short summary how it works and check the you understand the output:

```
b_string = 'La Femme is a French band'
print(b_string)        # print the variable
print(b_string[0:4])   # print the 5 first characters
print(b_string[5:])    # print elements from the 6th element till the end
print(b_string[0:6:2]) # the the 7 first characters skipping every second
print(b_string[1::2])  # print every 2nd element starting from the second (index 1)
```

There is one more thing to notice. Strings are immutable data types. This means that once we define a string, its elements cannot be changed. This is best illustrated by an example. Let's try to delete the second element:

```
del b_string[1]
```

It doesn't work since strings are immutable. We can, of course delete the whole variable. Let's do that and try to print and we see that the error message is a results of the fact that after deletion the variable `b_string` no longer exists:

```
del b_string
print (b_string)
```

8.9 Lists

Unlike strings, lists are mutable and we can change their elements, and we can append more elements. This is best demonstrated by examples. Let's first generate a list and print out the values of its elements. Lists are defined using square brackets:

```
a = [1.0,2.0,3.0]                                   # generate a list & print
print("a and its length:", a,len(a))                # number of elements (length)
b = [1.0,2,'rhubarb']                               # elements in a list do not have to be of same type
print("b and its 3rd element:", b,b[2])             # print b and its third element
empty = []                                          # create an empty list.
print("empty list:", empty)                         # Print empty
listinlist = [1.0,2.0,['barley','hops']]            # we can also have list as an element of a list
print("List in list & 3rd element:", listinlist,listinlist[2])
   # Prints the variable and its third element (it is a list!)
b[2] = 15                                           # Change the third element of b
print("b after changing the 3rd element:", b)
print("slice 0th to 3rd element:", b[0:2])          # Slicing works the same as with strings
print("more slices:", b[0:3:2])
print(b[0::2])
b.append(11)                                        # We can use the method append() to add value to the list
print("b after appending to it:", b)                # Shows that number 11 was added (appended) at the end
```

8.10 Tuples - ordered immutable lists of elements

Tuples are sometimes confusing, probably because they seem similar to lists. The term has its origin in mathematics (set theory) and it can be defined as an *ordered set of elements*. N-tuple is an ordered list of N elements. If you are a database person, 'record' is roughly the same. Tuples in Python are *immutable*, their elements cannot be changed once they have been created unlike was the case with lists. This is a useful feature in certain situations. Here are some practical points concerning tuples:

- Indexing starts from zero (the same as with lists)
- It is not possible to add (append/extend) or remove (remove/pop) elements from a tuple.
- Tuples are a safe way to store data (when appropriate). This is where immutability is the key.
- Slicing works the same as with lists
- When assigning a tuple, one uses the regular parenthesis
- Since they are immutable, it is faster to retrieve information from a tuple than a list
- There is a conversion routine between tuples and lists
- Elements in a tuple can be mixed: You can have floats, integers, strings, lists, etc.

```
days = ('lundi','mardi','mercredi','jeudi','vendredi','samedi','dimanche')
print(days)             # print the tuple
print(days[1])          # print element two
#days[1] = tuesday      # Elements cannot be changed (immutable). This gives an error message
```

Remove the comment on the last line and execute. We cannot change the elements individually. We can, of course, reassign the tuple:

```
days = ('Monday','Tuesday','Wednesday','Thursday','Friday','Saturday','Sunday')
print(days)                    # check that everything was reassigned
print(days[0:3])               # slicing works as before
print(days[0:7:2])             # print every second element staring from the first one
#days = days.append('new day') # comment out and try. Tuples are immutable -> we can't add to them
```

8.11 Dictionaries

This may sound like an odd variable, but it is very useful. Dictionaries are *key-value pairs*. The most obvious examples are a dictionary and a phone book, but this data type is very useful for many purposes. Curly brackets are used when defining a dictionary. Below, we use the dot notation to call the function `keys()` to list all the keys of a dictionary.

```
#-- Define some key-value pairs:
engfra={'Monday':'lundi','Tuesday':'mardi','Wednesday':'mercredi'}
print(type(engfra))      # check the data type
print(engfra['Monday'])  # print out the value corresponding to the key Monday
print(engfra['Monday'])
print(engfra.keys())     # This is how you can list keywords of a dictionary
```

8.12 Combined variables

It is also possible to define *combined variables*. This is very useful in many applications. Let's illustrate this with an example. Remember that curly braces, angular brackets and normal brackets have very specific meanings as discussed above.

```
# define a combined variable. Pay attention to the different brackets

music = {
    'Drake': [
        'Toosie slide',
        'Hotline Bling',
        'Laugh Now Cry Later'
    ],
    'Justin Bieber': [
        'Sorry',
        'Yummy',
        'Lonely'
    ],
    'La Femme': [
        'Ou va la monde',
        'Hypsoline',
        'It\'s time to wake up'
    ]
}

print(type(music))                      # Check the variable type
print(music.keys())                     # Let's print the keys
print(music['Drake'])                   # print out the values associated with the key 'Drake'
print(music['Justin Bieber'][1])        # print out the second value associated with the key 'Justin Bieber'
print(music.values())                   # Instead of keys, we can also print the values only
music['La Femme'][0] = 'Sur la planche' # Since the value for each key is a list, we can change the values
print(music['La Femme'])                # Check that the substitution was made
```

8.13 Checking who's who: `type`, `len`, `id`

Here is a list of some commands that are helpful in inspecting variables. We have already used some of them above

```
print(type(1))       # what is type of the variable
print(type(1.0))     # notice the difference to the above
print(type(engfra))  # the one from above
print(id(engfra))    # identity of a variable. Gives the unique identifier of the variable (not memory address)
print(len(engfra))   # length of a variable
```

8.14 Type conversion

Like in C/C++ and Fortran, Python has type conversion functions. Some of them are different from their C/C++ and Fortran counterparts. Try them out.

Conversion	Action
`int(x)`	Converts `x` to an integer
`str(x)`	Converts `x` to a string
`float(x)`	Converts `x` to a float
`complex(x)`	Converts `x` to a complex number
`hex(x)`	Converts `x` to a hexadecimal

Table 8.1: Basic type conversion operations in Python.

Need help? It is very easy. Execute the cell below and it will tell what methods the class complex (which belongs to the module `builtins`) has:

```
help("complex")
```

8.14.1 Operators

Basic Python does not include many mathematical operators or functions. More or less all functions have to called in by importing a module or modules and we will discuss that in a moment. The arithmetic operators that are included in basic Python are the usual ones and the order of execution follows the common rules:

`+`	addition
`*`	multiplication
`**`	exponentiation
`-`	subtraction
`/`	division
`%`	modular division

Python has also C-style assignment operators:

`a+=b`	for `a=a+b`
`a*=b`	for `a=a*b`
`a**=b`	for `a=a**b`
`a-=b`	for `a=a-b`
`a/=b`	for `a=a/b`
`a%=b`	for `a=a%b`

And the usual relational operators: `<`, `>`, `<=`, `>=`, `==`, `!=`

Need help? It is very easy: Just try for example

```
help("+")
```

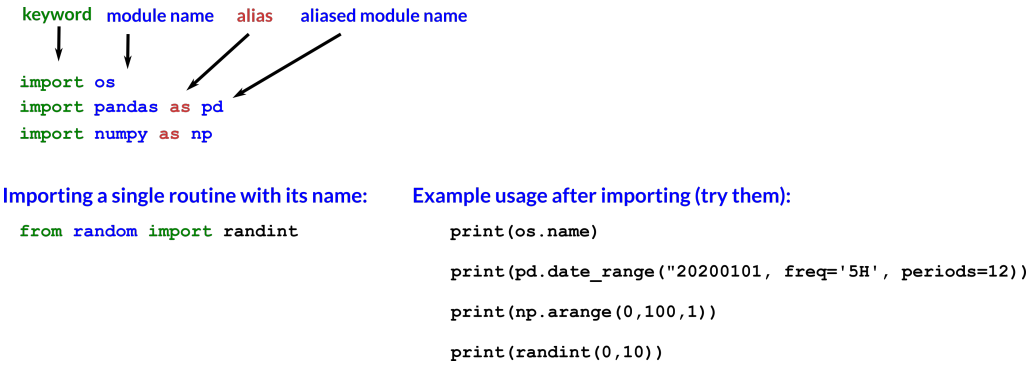


Figure 8.2: Basic terminology related to importing modules in Python. Numpy is library that provides lots of efficient array/matrix operations and numerical operations and it is used extensively in lots of applications. Aliasing using `as`; technically speaking this is not true aliasing but rather binding (in this case `np`) to the namespace (in this case `numpy`)) but that is beyond the point here.

8.15 Importing modules

Above, we used `import` to load a module, and we then used the *dot notation* to access a *function* from it. This is what we did:

```
import keyword                # import keyword module
print (keyword.kwlist)        # Print out the list of reserved keywords.
```

As briefly explained above, the module `keyword` is a built-in one. It provides the list of reserved keywords. In the above, `kwlist` is *function of the module*. In general, one accesses the *functions* or *methods* of a *module* using the *dot operator*. We will be using this extensively. The figure shows some of the basics and terminology. In the figure, an *alias* is created when importing module `numpy`.

...and don't forget to check what the modules in the figure do:

```
help("random.randint")
```

8.16 Plotting in Python

It's time to do some plotting. The example below is organized as follows:

- generate simple data and for plotting
- plot the data using `matplotlib`.
- while `matplotlib` can generate beautiful plots, we import `Seaborn` to decorate the plot
- replot the data with `matplotlib` and `Seaborn` decorations. This produces a publication-quality plot
- examples also include how to include and select plot title, loglog plot, markers and linestyle, and use LaTeXin labels

There are also other options to do plotting such as Plotly Express. Here, we use `matplotlib` as it is probably the most common way and it is very easy for Matlab users to use `matplotlib` due to syntactic similarity.

This also provides an introduction to `numpy`, loops and plotting options.

The following is very handy way for generating arrays:

```
np.arange(start=1, stop=10, step=3)
```

Let's use it to generate data for plotting

```
import numpy as np
import matplotlib.pyplot as plt

#---- Create the data:

x_data = np.arange(0.1,10,0.2)    # generate some simple data starting in the range 0.1-10 with steps of 0.2
print(x_data)                     # Let's check the data
y_data = x_data**2                # Let's use the data to generate a simple function.

#---- Create a quick plot using matplotlib:
#
#     A pretty publication-quality plot is created below after this one
#     using decorations, colors etc. from Seaborn.
#
#     Linestyles and markers are more or less the same as in Matlab. See:
#     https://matplotlib.org/2.0.2/api/markers_api.html#module-matplotlib.markers
#     https://matplotlib.org/2.0.2/api/lines_api.html#matplotlib.lines.Line2D.set_linestyle

plt.title('One can have a title too')
plt.plot(x_data,y_data, color='black', linestyle=':', marker='p')
plt.xlabel("x-title")
plt.ylabel("y-title")
# plt.legend(loc='center right')

plt.show()
```

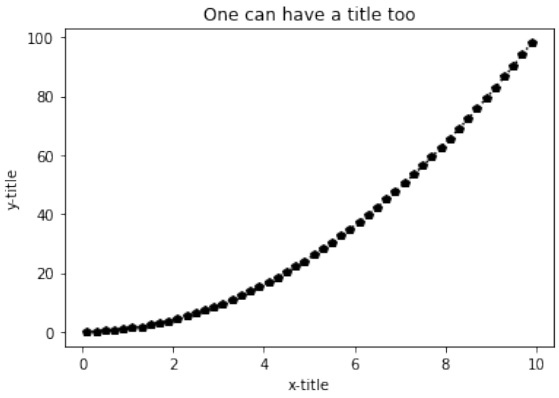


Figure 8.3: A basic plot using `matplotlib`.

The above shows how easy and quick it is to generate plots from data and to add titles and labels, and to change line styles and markers. Let's take this a bit further. Seaborn is based on matplotlib and it allows changing styles, color palettes and other decorations very easily. In the next cell, we import Seaborn and define background, grid, fonts and line widths. These definitions have been used to create figures for publications and the line widths and other styles have been selected such that the plots look good when printed.

The best way to use the cell below is to save it separately and then import it. If you plan to use these or some other settings in this same manner, that is what you should do.

```
#=============================================================
#
# Import Seaborn to make a pretty plot.
#
# Seaborn is excellent for controlling decorations, line widths
# color palettes, etc.
#
#   - Define background for the plot
#   - Define if a grid is used (background)
#   - Define font families
```

```
#   - Define line widths
#
# Ideally, this should be put in a separate file and imported,
# but is included here as an example.
#============================================================

import seaborn as sns
from matplotlib import rc


sns.set()
sns.set_context("paper", font_scale=1.5, rc={"lines.linewidth": 2.0})

#---- This allows the use of LaTeX + the use sans-serif fonts also for tick labels:

rc('text', usetex=True)
rc('text.latex', preamble=r'\usepackage{cmbright}')

# There are 5 presents for background: darkgrid, whitegrid, dark, white, and ticks
# Define how ticks are placed and define font families

sns.set_style("ticks")
sns.set_style("whitegrid",
 {'axes.edgecolor': 'black',
 'axes.grid': True,
 'axes.axisbelow': True,
 'axes.labelcolor': '.15',
 'grid.color': '0.9',
 'grid.linestyle': '-',
 'xtick.direction': 'in',
 'ytick.direction': 'in',
 'xtick.bottom': True,
 'xtick.top': True,
 'ytick.left': True,
 'ytick.right': True,
 'font.family': ['sans-serif'],
 'font.sans-serif': [
  'Liberation Sans',
  'Bitstream Vera Sans',
  'sans-serif'],})
```

The cell below plots the data using matplotlib with the seaborn settings to make the above plot pretty with nice and consistent fonts, control of tick marks, legends and so on. The line widths were determined in the previous cell so that if the plot is included in an article or thesis, everything should look good. At the end, there is an option to save as png and svg. While png is good for importing plots into Word, LaTeX or LibreOffice, svg (=Scalable Vector Graphics) is scale free. That is, it can be used in posters without loss of resolution and it can be edited (without the need to replot), say, if one wants to change the line type or color. svg files can be edited using the excellent inkscape program (free and available for Windows, Mac and Linux) and then saved to png or any other format if necessary. Note also that svg can be directly imported in LibreOffice files and also in Jupyter Labs/Notebook and on web sites. It's usually a good idea so save in both formats but if one needs to pick one, svg is a much more flexible choice.

```
#============================================================================
#
# Let's make the plot pretty with settings from above.
#
#  - Produces a publication quality plot for an article or thesis.
#============================================================================

#---- Choose Seaborn colour palette.
#     This is the only Seaborn piece in this cell, all the rest would work without it.
#     The previous cell does all the decorations and that's what Seaborn did for us.
#
#     More palettes: See: https://seaborn.pydata.org/tutorial/color_palettes.html

sns.set_palette("tab10")

#---- This allows modification of tick positions (& a few other things)

from matplotlib.ticker import (AutoMinorLocator, MultipleLocator)
fig, ax = plt.subplots(1,1)

# Set x and y-minor ticks:

ax.yaxis.set_minor_locator(MultipleLocator(5.0))
ax.xaxis.set_minor_locator(MultipleLocator(1.0))

#---- Plot data. One can use LaTeX for legends (label gets picked up by
#     the legend part below).
#     Linestyles and markers are more or less the same as in Matlab. See:
#     https://matplotlib.org/2.0.2/api/markers_api.html#module-matplotlib.markers
#     https://matplotlib.org/2.0.2/api/lines_api.html#matplotlib.lines.Line2D.set_linestyle
```

```
plt.plot(x_data,y_data, label='$x^2$', linestyle=':', marker='D')

#---- Axis labels and legend:

plt.xlabel("x-axis")
plt.ylabel("y-axis")
plt.legend(title="Function",loc='center right',fontsize='small',fancybox=True)

#---- It's very easy to plot vertical and horizontal lines for example
#     for the average values or put them in any arbitrary position.
#     Linestyles are the same as in Matlab

plt.axvline(x=np.mean(x_data), color='black', linestyle=':')
plt.axhline(y=np.mean(y_data), color='black', linestyle='--')

#---- You can add text using the coordinate positions (x,y)
#     The example also shows how to concatenate text from extracted data

plt.text(0, 80, 'Curve: $y=x^2$\nAverage x value is: '+str(np.around(np.mean(x_data),decimals=2)), rotation=0)

#---- Uncomment to save. First give a name. Then a high resolution png and svg files are generated

#printfile = 'testplot'
#plt.savefig(printfile+'.svg')
#plt.savefig(printfile+'.png',dpi=300)

#---- Show

plt.show()

#---- One can plot more so let's generate a log-log plot:

plt.title('One can have a title too')
plt.loglog(x_data,y_data, label='$x^2$', linestyle='-.')
plt.legend(title="Function",loc='upper left',fontsize='small',fancybox=True)

plt.show()
```

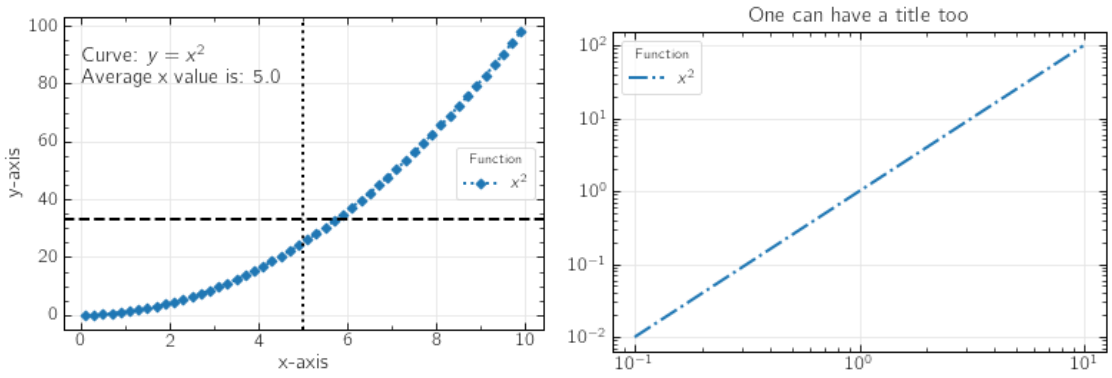


Figure 8.4: A basic plot using `matplotlib`.

8.17 Variation to the theme: plots xkcd style

Not that this is useful, but it is fun: One can eve plot using the XKCD comics style. To make the plots look more authentic, install the humor sans font (not required, though) from the terminal window using the command sudo apt install -y fonts-humor-sans

Let's plot the same plot as above.

```
rc('text', usetex=False)        # Using LaTeX doesn't work with xkcd style so let's turn it off

#-- Note: we use the 'with' since we don't want to make this permanent (only in this context)

with plt.xkcd(scale=1, length=100, randomness=1):   # This turns on the xkcd mode. It has 3 parameters
#with plt.xkcd():                                   # Use default values for the parameters
    plt.xlabel("X-AXIS")
    plt.ylabel("Y-AXIS")

    plt.axvline(x=np.mean(x_data), color='black', linestyle=':')
    plt.axhline(y=np.mean(y_data), color='black', linestyle='--')
```

```
plt.title('One can have a title too')
plt.plot(x_data,y_data, label=r'$x^2$', linestyle='-.') # Note that after turning of LaTeX, this is the
     way we can have it/
plt.legend(title="Function",loc='upper left',fontsize='small',fancybox=True)

plt.show()
```

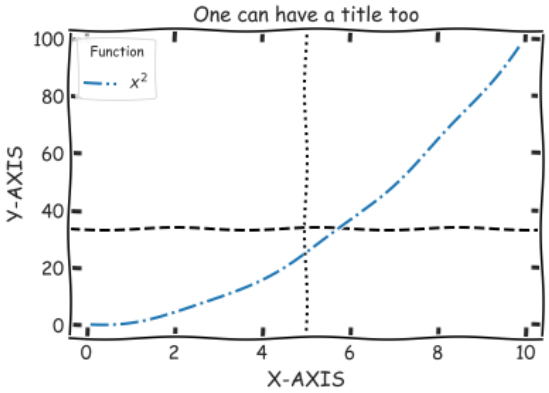


Figure 8.5: XKCD-style plot using `matplotlib` and `seaborn`.

8.18 Plotting with data from your computer

The following shows one way of reading data from a file that has data in columns separated by spaces. Seaborn is again used for pretty plots. This also demonstrates numpy and pandas, two very useful modules. This is not very elegant but aim is to demonstrate plotting using data stored in a file, how to use a for loop and how one can get some quick statistics.

We will use two approaches: 1) Direct reading and plotting, and 2) reading the data, converting it to pandas dataframe and then plotting. The second part is shown since pandas dataframe is a very useful (and very commonly used) way of handling data. There are many other ways to plot by calling Seaborn directly, using scatter etc.

Data: It is assumed that data is in `$HOME/test_plot/test_data.dat`

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import os

from helpers import pretty_plots
pretty_plots.pretty_plot(pal='tab10',size='paper')

#---- Get home directory. Demonstrates the module os

home_dir = os.environ['HOME']
print('My home directory is:', home_dir)

#---- Input data: x,y1,y2 data is in this file ----------

data_1 = home_dir+'/test_plot/test_data.dat'

#---- Let's assign arrays and --------------------
#     read the data from the file line-by-line into the variables:
#     Floats are read from a line and appended to the arrays.
#     The data file is opened for reading only.

x, y_1, y_2 =   [], [], []

for line in open(data_1, 'r'):
  values = [float(s) for s in line.split()]
  x.append(values[0])
  y_1.append(values[1])
  y_2.append(values[2])
```

```
#---- Let's create a pandas dataframe from the data
#     Note really necessary, but demonstrates pandas

df = pd.DataFrame({
    "time in ns"    : x,
    "measurement 1" : y_1,
    "measurement 2" : y_2
})

#---- To write the data as a csv file with the column headers
#     give a name for the out file and uncomment:

#df.to_csv(outfile, index=False)

#---- This is one reason why dataframes are so handy. Let's print it on screen,
#     easy to see variables and get statistics:

print('\nDataframe:\n')
print(df)

#---- Compute the averages of the two data columns using numpy and print them on screen:

y_1_mean = df['measurement 1'].mean()
y_2_mean = df['measurement 2'].mean()

print('\n')
print(y_1_mean)
print(y_2_mean)
print('\n')

aa=df[['measurement 1','measurement 2']]

#--- print some more statistics

print(aa.dropna().describe())

#---- Create a simple plot of the data in one figure
#     A pretty plot is created below after this one.
#     Note: ax provides a handle to share the axis


ax = df.plot(x='time in ns',y='measurement 1', s=40, kind='scatter', color='tab:blue', label='Data column 1')
df.plot(x='time in ns',y='measurement 2', ax=ax, s=40, kind='scatter', color='tab:orange', label='Data column
     2')
df.plot(x='time in ns',y='measurement 1', ax=ax, kind='line', label='')
df.plot(x='time in ns',y='measurement 2', ax=ax, kind='line', label='')


plt.ylabel("measurement data")
plt.legend(loc='center right')

#---- Plot lines for the average values

plt.axhline(y=y_1_mean, color='black', linestyle=':')
plt.axhline(y=y_2_mean, color='black', linestyle='--')

#---- Plot the average values in the figure.
#     Note: Since LaTeX is enabled, the underscore character needs to be escaped using backslash

plt.text(1, 1.5, 'Average y\_1: '+str(np.around(y_1_mean,decimals=1)), rotation=0)
plt.text(1, 1.35, 'Average y\_2: '+str(np.around(y_2_mean,decimals=1)),rotation=0)


plt.show()
```

The output:

```
My home directory is: /home/mkarttu

Dataframe:

   time in ns  measurement 1  measurement 2
0         1.0           1.00           2.00
1         2.0           1.10           1.90
2         3.0           0.90           1.95
3         4.0           1.20           2.10
4         5.0           1.02           2.05

1.044
2.0

       measurement 1  measurement 2
count       5.000000       5.000000
mean        1.044000       2.000000
std         0.112606       0.079057
min         0.900000       1.900000
25%         1.000000       1.950000
50%         1.020000       2.000000
75%         1.100000       2.050000
max         1.200000       2.100000
```

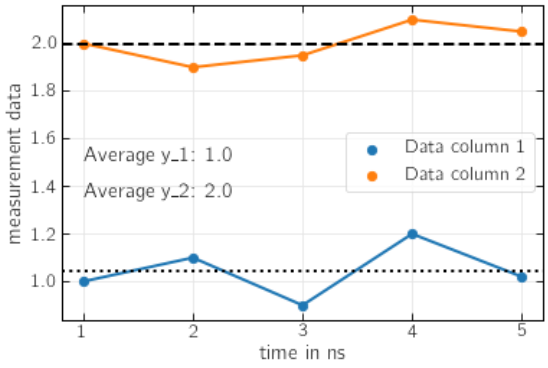


Figure 8.6: Plot using `matplotlib` and `seaborn`, and data from `$HOME/test_plot/test_data.dat`.

8.19 A very brief intro to loops and conditionals

8.19.1 `for` loop

We already saw an example of this. But here is the `for` loop in brief. Note intendation, it is critical.

1. In example 1, we woop over a range of values, pring out, append values to a list, convert list to a numpy array. Why the last? If/when one needs to perform numerical calculations numpy has lots of optimized routines. As an example, perform an element-wise numpy operation. Why not use lists? Lists are much slower but they are convenient for reading in data and some other stuff.
2. In example 2, the loop goes over the characters of a string. This is closely related to what was done above when we read lines using a for loop.

```
#--------------------------------
# Example 1.

import numpy as np
a = 0                                           # We already used the variable a above so let's initialize
     it.
b = []                                          # initalize list b and check that it is empty.
print(b)

for i in range (11,17):
    a = a+i
    b.append(i)                                 # Append values to the list
    print('The value of a is',a,'index:', i)

b_new = np.array(b)                             # Convert b to a numpy array.
print(b, type(b))                               # Let's check the original and converted
print(b_new, type(b_new))
print(np.sqrt(b_new))
```

The output:

```
[]
The value of a is 11 index: 11
The value of a is 23 index: 12
The value of a is 36 index: 13
The value of a is 50 index: 14
The value of a is 65 index: 15
The value of a is 81 index: 16
[11, 12, 13, 14, 15, 16] <class 'list'>
[11 12 13 14 15 16] <class 'numpy.ndarray'>
[3.31662479 3.46410162 3.60555128 3.74165739 3.87298335 4.        ]
```

```
#-----------------------------------------------------------------------------------------------
# Example 2.

for i in "Ferme les yeux est imagine-toi par Soprano et Blacko.":
    print(i, end='')  # the part end='' prints without changing line. Comment it out and see.
```

The output:

```
Ferme les yeux est imagine-toi par Soprano et Blacko.
```

8.19.2 `while` loops and breaking/stopping a loop

The `while` loops is used for executing a loop while some (logical) condition is true. Here is also an example of a simple `if` statement.

```
#-----------------------------------------------------
# Example. A simple while loop and a break statement

import numpy as np

x = np.arange(0,10,1)
i = 0
while i < max(x):
    i = i+x[2]
    print(i)
    if i == 4:
        print('Need a break?')
        break
```

8.19.3 `if` - `elif` - `else` statement

This is used to test a condition. Note that it is critical to have proper indentation.

```
#---------------------------------------------------------------------
# Example.
#
#   Let's generate random integers between 0 and 100.
#   The argument size gives the length of the array (vary and re-run).

import numpy as np

x = np.random.randint(100,size=(2))
j = 0
for i in x:
    if x[j] > 6 and x[j] < 10:
        print('Condition 1:',j,x[j])
    elif x[j] > 30 and x[j] < 40:
        print('Condition 2:',j, x[j])
    else:
        print('Condition 3:',j, x[j])
    j = j+1
```

The output:

```
Condition 2: 0 39
Condition 3: 1 58
```

8.20 What was not covered

The above covers the basics and provides a starting point for being able to produce and analyze data. The main thing that was not covered is object oriented programming. For those who are interested, here is some further reading:

Object-Oriented Programming (OOP) in Python 3

8.21 Problems

Problem 8.1 In Section 1.11 we briefly discussed floating point numbers and their representation on a computer. Find the Python commands that give the smallest and largest floating point numbers (one for each), and a single Python command that gives information about floats. Remember to print out the outputs!

Appendix A

The `vi` editor

The `vi` editor is the basic ASCII text editor that comes with essentially all Linux/Unix systems. It was released already in 1976 (by Bill Joy) and, despite its age, `vi` remains one of the standard tools. The name `vi` comes from visual editor. `vi` works on the command line and does not have a GUI. It is very lightweight, minimalistic and powerful.

`vi` differs from most editors (with or without GUI) in that it has two modes (this is why it is a modal editor): 1) *command mode* and 2) *editing or insert mode*. As the names suggest, these modes serve for different purposes. In the *command mode*, one can enter commands such as `copy`, `paste`, `save`, `quit` and so on. Editing text is not possible in the command mode. For editing text, one has to enter the *editing or insert mode*. That mode is exclusively for editing text. For saving the file that is being edited, one has to switch to the *command mode*. Understanding the two modes is the key to using the `vi` editor.

A.1 Why `vi`

First, since it ships with all Linux/Unix systems, it is always present without any additional installations. Second, is extremely lightweight. Those two factors are very important in the case if/when something goes wrong with the system and some of the system files need to be edited: In such situation GUI based editors (even if they are ASCII editors) may be impossible to use. The `vi` editor is sometimes referred to as the system admin's best friend. Although switching between the modes may sound cumbersome, it is very easy to get used to that and `vi` is a favorite of many.

A.2 Switching between the *command* and *editing* modes

- When `vi` is started, it is always in the command mode.
- Commands are *case sensitive*
- To switch from the *command mode* to the *insert mode*: press the key `i`.
- Switch from the *insert mode* to the *command mode*: press the `esc` key.

A.3 Starting `vi`

To open a file or to create a new file, simply type

```
vi filename.txt
```

If filename.txt exists it will be simply opened. If it does not exist, it will be created. Figure A.1 shows a screenshot of the vi editor opened from the command line and a few lines added in the new file. Figure A.2 shows a screenshot of how to search for a string in vi and Figure A.3 shows

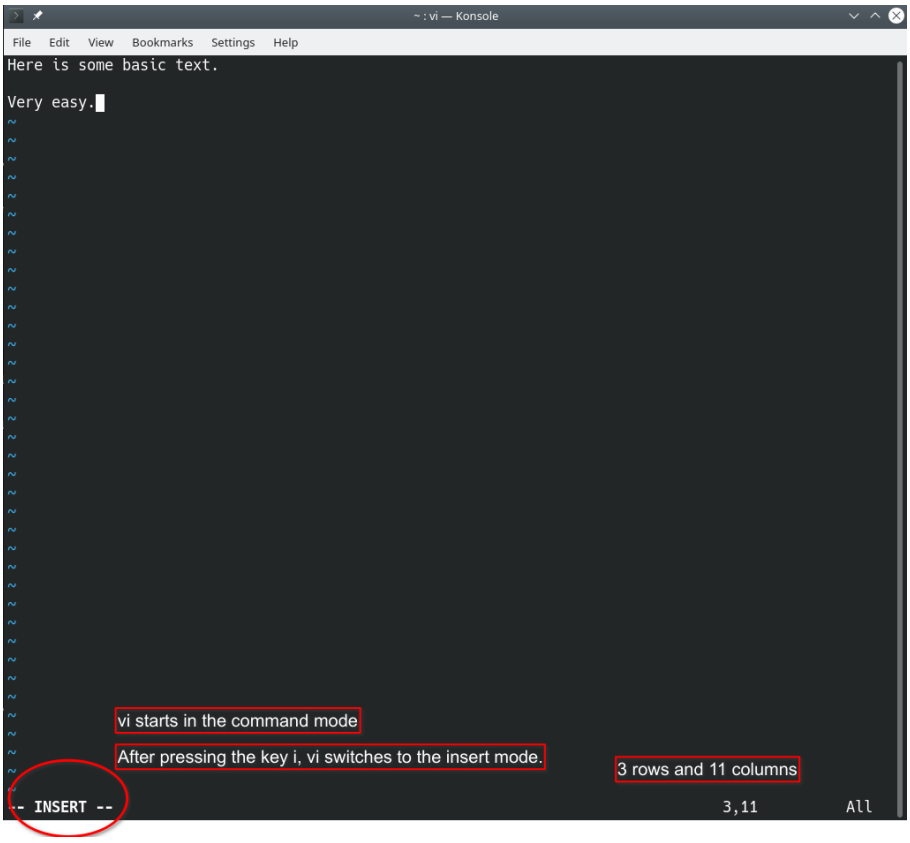


Figure A.1: The vi editor opened with a few lines of text added. The text INSERT at the left hand corner of the screen indicates that vi is in the *edit* or *insert* mode.

how to save the file and exit the editor.

A.4 vi command summary

Below is a summary of some of the most common commands. Open a test file and try them.

COMMAND	FUNCTION	NOTES
vi filename.txt	edit/create a file called filename.txt	On the command line terminal. Starts vi in *command mode*
i	Switch to *insert mode*	Shows the text -- INSERT -- at the bottom of the screen
esc	Switch to *command mode*	
x	delete character to the right of the cursor	
yy	copy a line (of text)	
p	paste below the current line (of text)	
4x	delete 4 characters (to the right of the cursor)	

COMMAND	FUNCTION	NOTES
X	delete character to the left of the cursor	
dw	delete word to the right of the cursor	
2dw	delete 2 words to the right of the cursor	
dd	delete the current line	
2dd	delete 2 (current and the next) lines	
D	delete all characters from the cursor to end of the line	
o	insert an empty line below the cursor	Switches to *insert mode*
O	insert an empty line above the cursor	Switches to *insert mode*
r	replace the character under the cursor	Switches to *insert mode*
a	append text to the right of the cursor	Switches to *insert mode*
A	append to end the line	
cw	replace the word	Switches to *insert mode*
3cw	replace 3 words (to the right; leaves the rest the line after the two words intact)	Switches to *insert mode*
C	delete the text from cursor to end of line	Switches to *insert mode*
J	join the line below to the current line	Switches to *insert mode*. The cursor can be anywhere on the line
3J	join the 3 following lines below to the current line	Switches to *insert mode*. The cursor can be anywhere on the line
u	undo	Switches to *insert mode*
ZZ	save the file and quit	Switches to *insert mode*
:w	save the file & continue to edit	Switches to *insert mode*
:w newname.txt	save the file with name newname.txt	Switches to *insert mode*
:r name.txt	read text into the current file from file named name.txt	Switches to *insert mode*
:wq	save the file and quit	Switches to *insert mode*
:q!	discard all changes since last save and quit	Switches to *insert mode*
w	move forward word-by-word	
b	move backward word-by-word	
$	move to the end of the current line	
0	move to the beginning of the current line	
H	go to the top line of the current screen	
M	go to the middle line of the current screen	
b	go to the beginning of the word	
e	go to the end of the word	
:num	show the line number	
L	go to last line of the current screen	
G	go to the last line of the file	
1G	go to first line of the file	
Ctrl-f	scroll forward one screen	
Ctrl-b	scroll backward one screen	
Ctrl-d	scroll down one-half screen	
Ctrl-u	scroll up one-half screen	

COMMAND	FUNCTION	NOTES
:/ *keyword* + return	search for *keyword* in the current text	To repeat the search forward press n, backward: N
:s/ *keyword* + return	search for *keyword* in the current line	To repeat the search forward press n, backward: N
:s/word1/word2 + return	search for word1 on the current line AND replace it by word2	in *command mode*.
:%s/word1/word2 + return	search for all occurrences of word1 AND replace them by word2	in *command mode*.

A.5 vi cursor movements:

Instead of using the arrows, the cursors (in *command mode*) can be moved using the following keys:

COMMAND	FUNCTION
h	move left one space
j	move down one line
k	move up one line
l	move right one space

A.6 vi in readonly-mode

vi can also be used in *readonly-mode*. The search and movement functions work as listed above, but editing and saving is not possible in the readonly mode. Here's how:

```
vi -R filename.txt
```

A.7 vim: The basic vi improved

vim is an improved vi and a very popular ASCII text editor - vi and vim are arguably the most popular text editor in Linux/Unix. vim adds several features to the basic vim and all of the above commands work also in vim. It comes (just like the basic vi) with most (if not all) current Linux/Unix systems as well as Mac.

vim stands for *Vi IMproved*. It adds features to the basic vi and is highly customizable. It also has many plugins including git integration and a scripting language. vim is available for essentially all operating systems including iOS and Android.

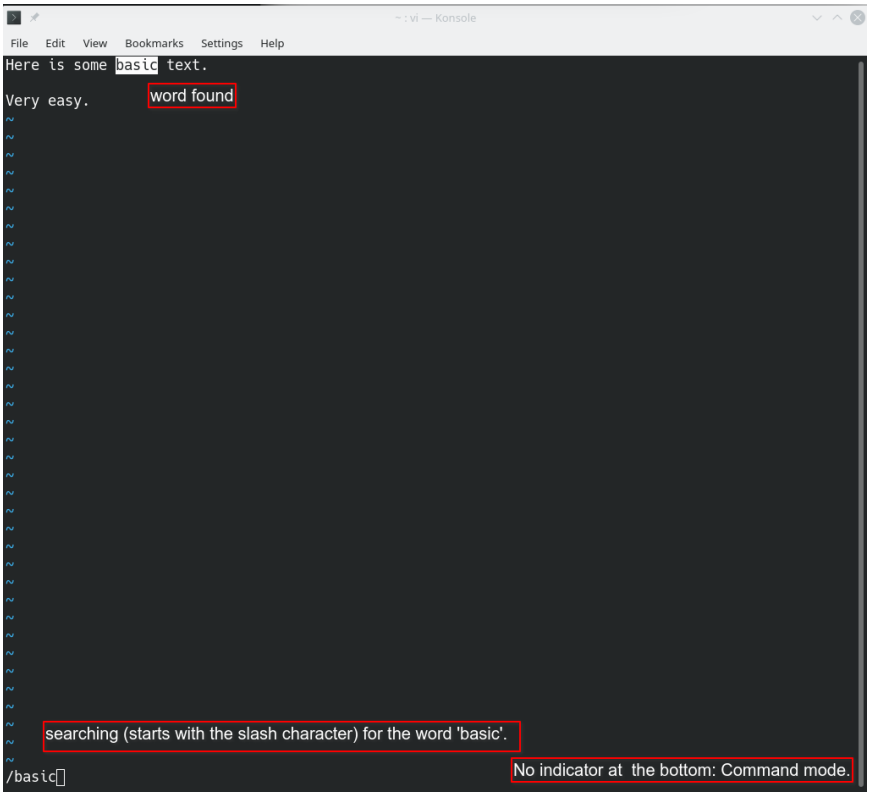


Figure A.2: Searching for string in vi. The editor is in the *command mode*.

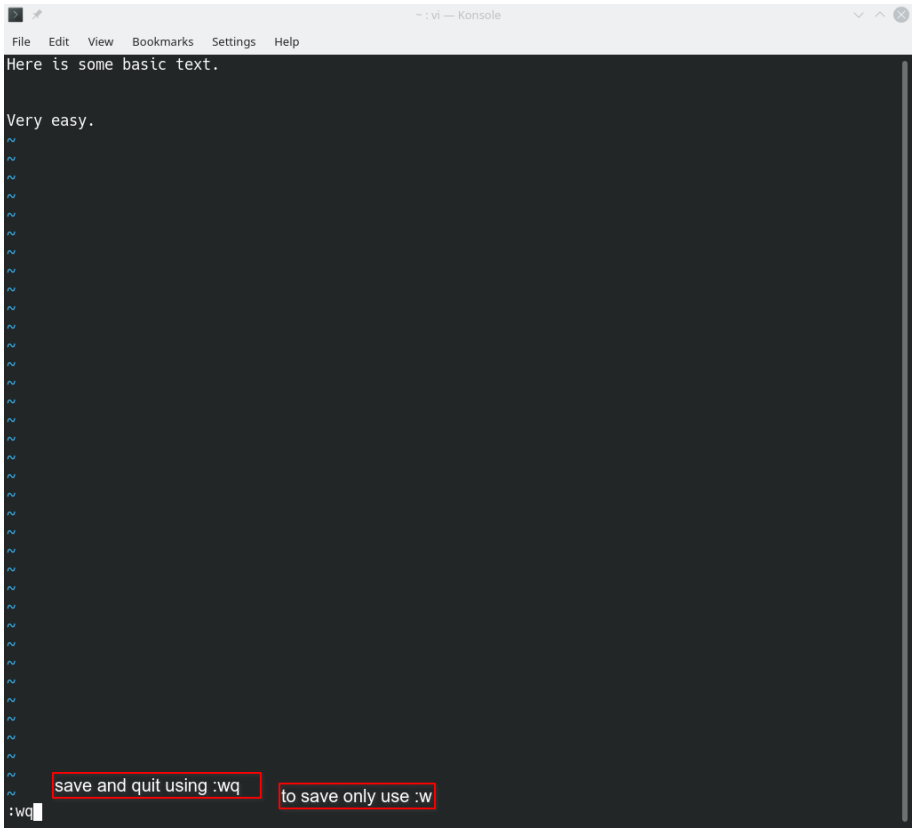


Figure A.3: Saving the file and exiting the vi editor. vi is in the command mode.

Appendix B

Software used to create these notes

- Linux: Ubuntu 22.04 LTS, KDE Neon USer Edition (Ubuntu 22.04 LTS based); macOS: Catalina, Big Sur; Windows 10/11 and WSL.
- Jupyter Lab
- Inkscape for (vector) graphics.
- Gimp for graphics (bitmaps).
- Python with `matplotlib` and `seaborn` for plots.
- LaTeX for typesetting
- `Visual Studio`, `emacs` and `vi` for editing ASCII files
- Overleaf for LaTeX document management

Bibliography

[1]N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller, "Equation of state calculations by fast computing machines", J. Chem. Phys. **21**, 1087 (1953).

[2]P. D. Walker and P. G. Mezey, "A new computational microscope for molecules: high resolution MEDLA images of taxol and HIV-1 protease, using additive electron density fragmentation principles and fuzzy set methods", J. Math. Chem. **17**, 203–234 (1995).

[3]Press, William H., S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical recipes in C++: The art of scientific computing* (Cambridge University Press, Cambridge, Cambridgeshire, New York, 2002).

[4]E. H. Lee, J. Hsin, M. Sotomayor, G. Comellas, and K. Schulten, "Discovery through the computational microscope", Structure **17**, 1295–1306 (2009).

[5]G. H. Golub and C. F. Van Loan, *Matrix computations* (JHU Press, Feb. 2013).

[6]J. Wong-ekkabut and M. Karttunen, "The good, the bad and the user in soft matter simulations", Biochimica et Biophysica Acta (BBA) - Biomembranes **1858**, 2529–2538 (2016).