Tools for Scientific Computing

A. Latina, CERN

Abstract
A large multitude of scientific computing tools is available today. This article gives an overview of available tools and explains the main application fields. In addition, basic principles of number representations in computing and the resulting truncation errors are treated. The selection of tools is for those students, who work in the field of accelerator beam dynamics.

Keywords
CAS, CERN accelerator school, Tools for Scientific computing

1 Introduction
In this lesson, we will outline good practice in scientific computing and guide the novice through the multitude of tools available. We will describe the main tools and explain which tool should be used for a specific purpose, dispelling common misconceptions.

We will suggest reference readings and clarify important aspects of numerical stability to help avoid making bad but unfortunately common mistakes. Numerical stability is the basic knowledge of every computational scientist.

We will exclusively refer to free and open-source software running on Linux or other Unix-like operating systems. Also, we will unveil powerful shell commands that can speed up simulations, facilitate data processing, and in short, increase your scientific throughput.

1.1 Floating-point numbers
Computers store numbers not with infinite precision but rather in some approximation that can be packed into a fixed number of bits. One commonly encounters integers and floating-point numbers.

The technical standard for floating-point arithmetic, IEEE Standard for Floating-Point Arithmetic (IEEE-754), was established in 1985 by the Institute of Electrical and Electronics Engineers (IEEE). The standard addressed many problems found in the diverse floating-point implementations that made them difficult to use reliably and portably.

1.1.1 Error, accuracy, stability
Arithmetic between numbers in integer representation is exact, if the answer is not outside the range of integers that can be represented. Real numbers use a floating-point representation IEEE-754, where a number is represented internally in scientific binary format, by a sign bit (interpreted as plus or minus), an exact integer exponent, and an exact positive integer mantissa (or fraction), such that a number is written as:

\[ \text{value} = (-1)^{\text{sign}} \times 2^{\text{exponent}} \times 1.M \]

Single-precision floating point numbers use 8 bits for the exponent (therefore one can have exponents between -128 and 127) and 23 bits for the mantissa, \( M \). Double-precision floating point numbers use 11 bits for the exponent (one can have exponents between -1024 and 1023) and 52 bits for the mantissa, \( M \).

The complete range of the positive normal floating-point numbers in single-precision format is:

\[ s_{\text{min}} = 2^{-126} \approx 1.17 \times 10^{-38}, \]
\[ s_{\text{max}} = 2^{127} \approx 3.4 \times 10^{38}. \]

In double-precision format the range is:
\[ d_{\text{min}} = 2^{-1022} \approx 2 \times 10^{-308}, \]
\[ d_{\text{max}} = 2^{1024} \approx 2 \times 10^{308}. \]

Some CPUs internally store floating point numbers in even higher precision: 80-bit in extended precision, and 128-bit in quadrupole precision. In C++ quadrupole precision may be specified using the `long double` type, but this is not required by the language (which only requires `long double` to be at least as precise as double), nor is it common. For instance, gcc implements `long double` as extended precision.

### 1.1.2 Machine accuracy and round-off error

The smallest floating-point number which, when added to the floating-point number 1.0, produces a floating-point result different from 1.0 is termed the **machine accuracy** \( \varepsilon_m \). For single precision
\[ \varepsilon_m \approx 3 \cdot 10^{-8}, \]
for double precision
\[ \varepsilon_m \approx 2 \cdot 10^{-16}. \]

It is important to understand that \( \varepsilon_m \) is not the smallest floating-point number that can be represented on a machine. That number depends on how many bits there are in the exponent, while \( \varepsilon_m \) depends on how many bits there are in the mantissa.

The **round-off error**, also called rounding error, is the difference between the exact result and the result obtained using finite-precision, rounded arithmetic. Round-off errors accumulate with increasing amounts of calculation. As an example of round-off error, see the representation of the number 0.1 in figure 3.

Notice that the number of meaningful digits that can be stored in a floating-point arithmetic, is approximately equal to:
\[
\text{number of digits} \approx \frac{\text{number of bits in the mantissa}}{\log_2(10)} \approx \frac{\text{number of bits in the mantissa}}{3}.\]
Fig. 3: Internal representation of the number 0.1 in floating point, single-precision. Notice the “error of conversion”, or round-off error.

Round-off errors accumulate with increasing amounts of calculation. If, in the course of obtaining a calculated value, one performs $N$ such arithmetic operations, one might end up having a total round-off error on the order of $\sqrt{N}\epsilon_m$ (when lucky), if the round-off errors come in randomly up or down. (The square root comes from a random-walk.)

1.1.3 Truncation error

Many numerical algorithms compute “discrete” approximations to some desired “continuous” quantity. For example, an integral is evaluated numerically by computing a function at a discrete set of points, rather than at “every” point. In cases like this, there is an adjustable parameter, e.g., the number of points or of terms, such that the “true” answer is obtained only when that parameter goes to infinity. Any practical calculation is done with a finite, but sufficiently large, choice of that parameter.

The discrepancy between the true answer and the answer obtained in a practical calculation is called the truncation error. Truncation error would persist even on a hypothetical “perfect” computer that had an infinitely accurate representation and no round-off error. As a general rule there is not much that a programmer can do about round-off error, other than to choose algorithms that do not magnify it unnecessarily (see discussion of “stability” below). Truncation error, on the other hand, is entirely under the programmer’s control. In fact, it is only a slight exaggeration to say that clever minimisation of truncation error is practically the entire content of the field of numerical analysis!

1.1.4 An example: finite differentiation

Imagine that you have a procedure which computes a function $f(x)$, and now you want to compute its derivative $f'(x)$. Easy, right? The definition of the derivative, the limit as $h \to 0$ of

$$f'(x) \approx \frac{f(x + h) - f(x)}{h}$$

practically suggests the program: Pick a small value $h$; evaluate $f(x + h)$; you probably have $f(x)$ already evaluated, but if not, do it too; finally apply equation (1). What more needs to be said? Quite a lot, actually. Applied uncritically, the above procedure is almost guaranteed to produce inaccurate results. There are two sources of error in equation (1): truncation error and round-off error.

Truncation error. We know that (Taylor expansion)

$$f(x + h) = f(x) + hf'(x) + \frac{1}{2}h^2f''(x) + \ldots$$
therefore
\[ \frac{f(x + h) - f(x)}{h} = f' + \frac{1}{2} h f'' + \ldots \]

Then, when we approximate \( f' \) as in equation (1), we make a truncation error:
\[ \varepsilon_t = \frac{1}{2} h f'' + \ldots \]

In this case, the truncation error is linearly proportional to \( h \),
\[ \varepsilon_t = O(h). \]

The round-off error. The round-off error has various contributions:

- Neither \( x \) nor \( x + h \) is a number with an exact binary representation; each is therefore represented with some fractional error characteristic of the machine’s floating-point format, \( \varepsilon_m \). Since they are represented in the machine as rounded to the machine precision, the difference between \( x + h \) and \( x \) is \( x \varepsilon_m \). A good strategy is to choose \( h \) so that \( x + h \) and \( x \) differ by an exact number, for example using the following construct:
\[
\text{temp} = x + h \\
h = \text{temp} - x
\]

Beware that the compiler could “optimize-out” these lines. Depending on the compiler and on the language, dedicated keywords can be used to prevent this.

- The fractional accuracy with which \( f \) is computed is at least
\[ \varepsilon_r = \varepsilon_m \left| \frac{f(x)}{h} \right| \]

(but for a complicated calculation with additional sources of inaccuracy it might be larger)

So one has a total error
\[ \varepsilon_{\text{total}} = \varepsilon_t + \varepsilon_r = \frac{1}{2} h f'' + \varepsilon_m \left| \frac{f(x)}{h} \right| \]

Equation (1.1.4) allows one to determine the optimal choice of \( h \) which minimizes \( \varepsilon_{\text{total}} \):
\[ h \sim \sqrt{\frac{\varepsilon_m f}{f''}} \]

1.1.4.1 Remark.
Notice that, if one takes a central difference,
\[ f'(x) \approx \frac{f(x + h) - f(x - h)}{2h}, \]
rather than a right difference like in equation (1), the calculation is more accurate, as the truncation error becomes
\[ \varepsilon_t = O(h^2). \]

The demonstration is left as an exercise to the reader.
1.1.5 Underflow and overflow errors, cancellation error

The underflow is a condition in a computer program where the result of a calculation is a number of smaller absolute value than the computer can actually represent in memory. Overflow is condition occurring when an operation attempts to create a numeric value that is outside of the range that can be represented with a given number of digits – either higher than the maximum or lower than the minimum representable value.

Try for instance:

\[ 1 \times 10^100 + 1 - 1 \times 10^100 \]

What do you obtain? Most likely, 0. Additions and subtractions between numbers that differ in magnitude by a factor that is larger than the machine precision are likely to incur in underflow or overflow errors. This is also called cancellation error (that sometimes can even be catastrophic as in the example above).

Expressions like the following,

\[ x^2 - y^2, \]

can incur in underflow errors if \( y^2 \ll x^2 \) (in particular, and more precisely, when \( y^2 < x^2 \varepsilon_m \)). Such an expression is more accurately evaluated as

\[ (x + y)(x - y). \]

Some cancellation could still occur, however, avoiding the squared power operation, it will be less catastrophic.

1.1.6 Numerical stability

Let’s take the function “sin cardinal” as an example. Sin cardinal, sinc(\( x \)), also called the “sampling function”, is a function that arises frequently in signal processing and the theory of Fourier transforms, and it is defined as

\[
\text{sinc}(x) = \begin{cases} 1 & \text{for } x = 0 \\ \sin x \div x & \text{otherwise.} \end{cases}
\]

The implementation of this function requires special attention because, when \( x \rightarrow 0 \), numerical instabilities appear due to the division between two nearly-zero numbers. A robust implementation comes from a careful consideration of this function. Let’s take the Taylor expansion sinc(\( x \)) to first order,

\[
\frac{\sin x}{x} \approx 1 - \frac{x^2}{6} + \ldots
\]

If we look at the right-hand side, we can appreciate the fact that in this form, when \( x \) is small, the numerical instability simply disappears. The final result will differ from zero if and only if

\[
\left| \frac{x^2}{6} \right| < \varepsilon_m.
\]

or, if \( x \) is made explicit, if

\[
|x| < \sqrt{6} \varepsilon_m.
\]

This leads to a robust implementation of the function sinc:

1. function sinc(x)
2.     taylor_limit = sqrt(6*epsilon);
3.     if abs(X) < taylor_limit
4.         return 1;
5.     endif
6.     return sin(x)/x;
7. endfunction
1.2 Exact numbers
In cases where double-, extended- or even quadruple-precision are not enough, there exist a couple of solutions to achieve higher precision and in some cases even exact results. One case use more bits of precision, from a few hundred up to thousands or even millions of bits, or symbolic calculation.

1.2.0.1 Symbolic calculation
Symbolic calculation is the “holy grail” of exact calculations. Programs such as Maxima know the rules of math and represents data as symbols rather rounded numbers. For example: 1/3 is actually a fraction “one divided by three”. Even transcendental numbers like $e$ and $\pi$ are known, together with their properties, so that $e^{i\pi}$ is exactly equal to $-1$. Symbolic math systems are invaluable tools, but they are complex, slow, and generally not appropriate for computationally intensive simulations. They should rather be used as helper tools to develop faster and more accurate algorithms. For example, to simplify expressions before they are coded in faster languages, with the aim to limit the number of operations affected by truncation and round-off errors. In the section 3 we will expand more this topic.

1.2.0.2 Arbitrary precision arithmetic
When numerical calculations with precision higher than double-precision floating points, one can resolve to use dedicated libraries that can handle arbitrary, user-defined precision such as GMP, the GNU Multiple Precision Arithmetic Library for the C and C++ programming languages.

GMP is a free library for arbitrary precision arithmetic, operating on signed integers, rational numbers, and floating-point numbers. There is no practical limit to the precision except the ones implied by the available memory in the machine GMP runs on. GMP has a rich set of functions, and the functions have a regular interface.

In case one needs to perform sporadically operations in arbitrary precision, one can use the shell command bc. See the dedicated paragraph in the following pages.

2 Scientific programming languages
Scientific programming languages allow one to perform numerical computations easily, using high-level concepts such as matrices, complex numbers, data processing, statistical analysis, fitting procedures. Their focus more on easiness of use and richness of the numerical toolboxes available, than on rapidity. Octave [1] and Scilab [2] are excellent examples of scientific languages (and they are modeled to resemble Matlab). Also Python [3] gained some popularity in this field, as explained below.

2.1 Octave
GNU Octave is a high-level language primarily intended for numerical computations. It is typically used for such problems as solving linear and nonlinear equations, numerical linear algebra, statistical analysis, and for performing other numerical experiments. It may also be used as a batch-oriented language for automated data processing.

The current version of Octave executes in a graphical user interface (GUI). The GUI hosts an Integrated Development Environment (IDE) which includes a code editor with syntax highlighting, built-in debugger, documentation browser, as well as the interpreter for the language itself. A command-line interface for Octave is also available.

2.2 Python
Python is a general-purpose programming language. Through modules such as numpy, scipy, and matplotlib, the Python instruction set grows significantly to include a rich set of functions very close to those provided by Octave and Matlab.
– Row Major Order: When matrix is accessed row by row (Python and C language)
– Column Major Order: When matrix is accessed column by column (Mathematics)

2.3 Gnuplot

Gnuplot is a portable command-line driven graphing utility for Linux, OS/2, MS Windows, OSX, VMS, and many other platforms. The source code is copyrighted but freely distributed (i.e., you don’t have to pay for it). It was originally created to allow scientists and students to visualize mathematical functions and data interactively, but has grown to support many non-interactive uses such as web scripting. It is also used as a plotting engine by third-party applications like Octave. Gnuplot has been supported and under active development since 1986.

Not many people know that Gnuplot can also be used for fitting functions. Gnuplot uses an excellent implementation of the nonlinear least-squares (NLLS) Marquardt-Levenberg algorithm to a set of data points \((x, y)\) or \((x, y, z)\). Any user-defined variable occurring in the function body may serve as a fit parameter, but the return type of the function must be real. Here it follows an example:

```plaintext
f(x) = a*x**2 + b*x + c
g(x,y) = a*x**2 + b*y**2 + c*x*y
FIT_LIMIT = 1e-6
fit f(x) 'measured.dat' via 'start.par'
fit f(x) 'measured.dat' using 3:($7-5) via 'start.par'
fit f(x) './data/trash.dat' using 1:2:3 via a, b, c
fit g(x,y) 'surface.dat' using 1:2:3:(1) via a, b, c
```

3 Symbolic computation

Computer algebra, also called symbolic computation or algebraic computation, is a scientific area that refers to algorithms and procedures for manipulating mathematical expressions and other mathematical objects. Computer algebra is generally considered as a distinct field of scientific computing, because scientific computing is usually based on numerical computation with approximate floating point numbers (truncation error!), while symbolic computation emphasizes exact computation with expressions containing variables that have no given value and are manipulated as symbols.

Computer algebra and symbolic calculation can also be used to optimize expressions prior to their implementation in other languages like Octave, Python, C, and C++, in order to minimize truncation and round-off errors.

3.1 Maxima

Maxima is a computer algebra system with a long history. It is based on a 1982 version of Macsyma, it is written in Common Lisp and runs on all POSIX platforms such as macOS, Unix, BSD, and Linux, as well as under Microsoft Windows and Android. It is free software released under the terms of the GNU General Public License (GPL). An excellent front end for Maxima is wxMaxima [4], see figure 4.

Maxima is a system for the manipulation of symbolic and numerical expressions, including differentiation, integration, Taylor series, Laplace transforms, ordinary differential equations, systems of linear equations, polynomials, sets, lists, vectors, matrices and tensors. Maxima yields high precision numerical results by using exact fractions, arbitrary-precision integers and variable-precision floating-point numbers. Maxima can plot functions and data in two and three dimensions.

Besides the fact of being free software and open-source, a great advantage of Maxima versus its more popular competitors (e.g., Mathematica), is that its syntax and outputs are compatible with languages like Octave, Python, C, and C++. This makes it possible to develop a complex calculation (like the harmonic oscillator, below), then copy it & paste it directly into your simulation code.
3.1.1 Maxima: 1-D harmonic oscillator

As an example of Maxima usage, we resolve the 2nd-order differential equation of a 1-D harmonic oscillator, where we express as $\beta$ the restraining focusing force. Of course, there is a connection between $\beta$, and the Twiss parameter $\beta$, known as $\beta$ function. We start with declaring $\beta$ as a positive constant.

(% i1) declare(beta, constant)$
(% i2) assume(beta > 0); [0 < $\beta$]

Follows the equation of motion. Notice the symbol ‘ ’ : it tells Maxima to defer the evaluation of the derivative to a later moment.

(% i3) eqn_1: 'diff(x(s), s, 2) + x(s)/beta**2 = 0;

\[ \frac{d^2}{ds^2}x(s) + \frac{x(s)}{\beta^2} = 0 \]

Let’s set the initial conditions

(% i4) atvalue(x(s), s=0, x0)$
(% i5) atvalue('diff(x(s),s,1), s=0, xp0)$

Solution

(% i6) desolve(eqn_1, x(s));

\[ x(s) = \frac{\beta^3 \sin \left( \frac{s}{\beta} \right) x0 + \beta^2 \cos \left( \frac{s}{\beta} \right) x0}{\beta^2} \]
\[ x(s) = \beta \sin \left( \frac{s}{\beta} \right) x p0 + \cos \left( \frac{s}{\beta} \right) x \theta \]

\[ \frac{d}{ds} x(s) = \cos \left( \frac{s}{\beta} \right) x p0 - \frac{\sin \left( \frac{s}{\beta} \right) x \theta}{\beta} \]

3.2 Symbolic computation in Python and Octave

Symbolic computations can also be performed within Octave and Python. This adds the possibility to perform basic symbolic computations, including common Computer Algebra System tools such as algebraic operations, calculus, equation solving, Fourier and Laplace transforms, variable precision arithmetic and other features, in scripts.

3.2.1 Octave “symbolic” package

Here follows an example of Octave symbolic:

```octave
% Load the symbolic package
pkg load symbolic

% This is just a formula to start with, have fun and change it if you want to.
f = @(x) x.^2 + 3*x - 1 + 5*x.*sin(x);

% These next lines take the Anonymous function into a symbolic formula
syms x;
ff = f(x);

% Now we can calculate the derivative of the function
ffd = diff(ff, x);

% and convert it back to an Anonymous function
df = function_handle(ffd)
```

3.2.2 Sympy

The Python package Sympy offers similar capabilities.

```python
>>> from sympy import *
>>> x = symbols('x')
>>> simplify(sin(x)**2 + cos(x)**2)
1
```

4 High-performance computing: C and C++

For intensive computations, no scripting language can beat the speed of compiled languages such as C and C++. The C++ version of a simulation can be hundreds or even thousands of times faster than the equivalent script written in Python or Octave.

The price to pay for such high speed is the limited expressiveness of the C and C++ languages and their lower level. Programming in C and C++ is closer to programming in the assembly language understood by the CPU than any of the aforementioned languages. Writing code in C is nearly equivalent to writing in assembly directly.

In other words, C and C++ are two low-level languages. Which is the reason why, on the one hand, they are regarded as “difficult” languages, but on the other hand it is their greatest power, and the reason for their unbeatable speed. Through pointers, for example, one can directly access memory areas...
and process data at light speed. Which unfortunately is the same speed at which one can heavily mess up things, for example by mistakenly accessing memory belonging to other processes.

Being the C language well established and standardized, despite low-level, a number of libraries exist to provide advanced numerical tools. One such library is the GNU Scientific Library (GSL), which implements a large variety of high-quality routines spanning nearly all aspects of numerical computing. The library provides a wide range of mathematical routines such as random number generators, special functions and least-squares fitting. There are over 1000 functions in total. GSL is free software under the GNU General Public License.

The object-oriented features of C++ come to help with the possibility to write and utilize complex “objects” (or classes) that can accomplish complex tasks in a safe manner, without sacrificing performance. This effectively means that, in C++, one can “customize” the language adding high-level objects and types to serve virtually any purpose. Of course, mathematical objects and physics laws find a perfect realisation in C++ classes, which makes C++ an ideal language for high-performance computing.

4.1 A word about C matrices

The C and C++ language specifications (as well as Python) state that arrays are laid out in memory in a row-major order: the elements of the first row are laid out consecutively in memory, followed by the elements of the second row, and so on. The way we access a matrix impacts the performance of our code. See this example:

```c
// C program showing time difference
// in row major and column major access
#include <stdio.h>
#include <time.h>

// taking MAX 10000 so that time difference
// can be shown
#define MAX 10000

int arr[MAX][MAX] = {0};

void rowMajor() {
    int i, j;
    // accessing element row wise
    for (i = 0; i < MAX; i++) {
        for (j = 0; j < MAX; j++) {
            arr[i][j]++;
        }
    }
}

void colMajor() {
    int i, j;
    // accessing element column wise
    for (i = 0; i < MAX; i++) {
        for (j = 0; j < MAX; j++) {
            arr[j][i]++;
        }
    }
}

// driver code
int main() {
    int i, j;
    // Time taken by row major order
    clock_t t = clock();
    rowMajor();
    t = clock() - t;
    printf("row major: %f\n", (float)t/1000.0);
    return 0;
}
```

The output is:

1 Row major access time: 0.492000 s
2 Column major access time: 1.621000 s

Notice that Octave use Matlab column-major order, which might lead to some confusion sometimes.

### 4.2 Linear algebra

#### 4.2.1 BLAS and LAPACK

LAPACK [5] is a library of Fortran 77 subroutines for solving the most commonly occurring problems in numerical linear algebra. It has been designed to be efficient on a wide range of modern high-performance computers. The name LAPACK is an acronym for Linear Algebra PACKage.

LAPACK routines are written so that as much as possible of the computation is performed by calls to the Basic Linear Algebra Subprograms (BLAS) [6]. BLAS defines a set of fundamental operations on vectors and matrices which can be used to create optimized higher-level linear algebra functionality. Highly efficient machine-specific implementations of the BLAS are available for many modern high-performance computers. The BLAS enable LAPACK routines to achieve high performance with portable code. The BLAS are not strictly speaking part of LAPACK, but Fortran 77 code for the BLAS is distributed with LAPACK, or can be obtained separately from netlib.

The library provides a low-level layer which corresponds directly to the C-language BLAS standard, referred to here as “CBLAS”, and a higher-level interface for operations on GSL vectors and matrices. Users who are interested in simple operations on GSL vector and matrix objects should use the high-level layer described in this chapter.

#### 4.2.2 Sparse matrices

Matrices which are populated primarily with zeros and contain only a few non-zero elements are called “sparse matrices”. Sparse matrices often appear in the solution of partial differential equations, like Poisson solvers. It is beneficial to use specialized data structures and algorithms for storing and working with sparse matrices, since dense matrix algorithms and structures can be prohibitively slow and use huge amounts of memory when applied to sparse matrices. Good implementations of sparse matrices come with GSL (see below), and SuiteSparse [link].

### 4.3 C++ template libraries

Templates are one of the most interesting and precious constructs of C++. operate with generic types. Templates allow the creation of functions and classes whose functionality can be adapted to more than one type or class without repeating the entire code for each type. Due to the implementation of templates in modern compilers, which enables significant optimisation at compilation time, they produce extremely efficient and fast code. For instance, one can imagine a C++ class dedicated to $2 \times 2$ matrices,
class Matrix22 {
  T r11, r12, r21, r22;
public:
  friend Matrix22 operator(const Matrix22 &a, const Matrix22 &b )
  {
    Matrix22 r;
    r.r11 = b.r21*a.r12+b.r11*a.r11;
    r.r12 = b.r22*a.r12+b.r12*a.r11;
    r.r21 = b.r21*a.r22+b.r11*a.r21;
    r.r22 = b.r22*a.r22+b.r12*a.r21;
    return r;
  }
  T determinant() const { return r11*r22 - r21*r12; }
};

To be used to allocate and operate on matrices of different types:

```cpp
// Declare three 2x2 matrices of different type
Matrix22<int> int_matrix;
Matrix22<double> dbl_matrix;
Matrix22<std::complex<double>> cmp_matrix;

// Print out the determinants
std::cout << int_matrix.determinant() << std::endl;
std::cout << dbl_matrix.determinant() << std::endl;
std::cout << cmp_matrix.determinant() << std::endl;
```

In short, templates are used to teach the compiler the archetype of a function, regardless of its actual type. In our example, our template class Matrix22 described what a $2 \times 2$ matrix is, regardless of the actual “content” that will constitute the matrix itself. Oftentimes, templates let our code do more than we even imagine or planned them for. For instance, one could imagine to define a Matrix22<Matrix22> which well represents a $4 \times 4$ matrix. The definition of determinant would still apply.

Template classes are extremely useful to define generic containers, such as lists, queues, vectors of elements. Their abstraction is their strength, together with the high level of optimisation that the C++ compiler can apply to the code. Template classes can also be used to define abstract functions or algorithms, as well as service classes like smart pointers.

### 4.3.0.1 C++ Standard Library

The C++ language comes with its “Standard Library” which consists of a large collection of generic containers, functions to use and to manipulate such containers, generic strings and streams (including interactive and file I/O), etc. It is based upon conventions introduced by the Standard Template Library (STL), and has been influenced by research in generic programming and developers of the STL. All features of the C++ Standard Library are declared within the std namespace. Useful container classes are for example std::array<T>, std::vector<T>, std::list<T>, std::deque<T>. To create dictionaries and associations between types, there exist associative containers like std::set<K>, std::map<K,T>.

The C++ standard library also offers efficient implementation of several algorithms, e.g., sorting algorithms, see for instance std::qsort, which provides a very fast implementation of the quicksort algorithm.

### 4.3.0.2 BOOST - [link]

Boost is a set of C++ libraries that provides support for tasks and structures such as linear algebra, pseudorandom number generation, multithreading, image processing, regular expressions, and unit testing. It contains 164 individual libraries (as of version 1.76). Boost is meant to build on top of STL and many of the libraries are slated to become part of the standard library eventually. Boost libraries are generally
less mature and less standard than STL.

4.3.0.3 Armadillo - [link]
Armadillo is a high quality linear algebra library for the C++ language, aiming towards a good balance between speed and ease of use. It provides high-level syntax and functionality deliberately similar to Matlab and Octave. Useful for algorithm development directly in C++, or quick conversion of research code into production environments. Armadillo provides efficient classes for vectors, matrices and cubes; dense and sparse matrices are supported; integer, floating point and complex numbers are supported. A sophisticated expression evaluator (based on template meta-programming) automatically combines several operations to increase speed and efficiency. Various matrix decompositions (eigen, SVD, QR, etc) are provided through integration with LAPACK, or one of its high performance drop-in replacements (e.g. MKL or OpenBLAS). Armadillo can automatically use OpenMP multi-threading (parallelisation) to speed up computationally expensive operations. Distributed under the permissive Apache 2.0 license, useful for both open-source and proprietary (closed-source) software. Armadillo can be used for machine learning, pattern recognition, computer vision, signal processing, bioinformatics, statistics, finance, etc.

4.3.0.4 Eigen - [link]
Eigen is a(n)other C++ template library for linear algebra: matrices, vectors, numerical solvers, and related algorithms. Eigen is implemented using the expression templates metaprogramming technique, meaning it builds expression trees at compile time and generates custom code to evaluate these. Using expression templates and a cost model of floating point operations, the library performs its own loop unrolling and vectorisation. This guarantees excellent execution speed at run time. Eigen itself can provide BLAS and a subset of LAPACK interfaces.

4.4 Random number generation
Random number generators are widely using in numerical physics. They are at the base of each Monte Carlo technique, and are often the only practical way to evaluate difficult integrals or to sample random variables governed by complicated probability density functions.

4.4.1 Pseudorandom numbers
It might seem impossible to produce random numbers through deterministic algorithms. Nevertheless, computer “random number generators” are in common use. Oftentimes, computer-generated sequences are in fact called pseudo-random, while the word random is reserved for the output of an intrinsically random physical process, like the elapsed time between clicks of a Geiger counter placed next to a sample of some radioactive element. Entire books have been dedicated to this topic, most notably [7].

A working, though imprecise, definition of randomness in the context of computer-generated sequences, is to say that the deterministic program that produces a random sequence should be different from, and –in all measurable respects– statistically uncorrelated with, the computer program that uses its output. In other words, any two different random number generators ought to produce statistically the same results when coupled to your particular applications program. If they don’t, then at least one of them is not (from your point of view) a good generator.

All of the algorithms produce a periodic sequence of numbers, and to obtain effectively random values, one must not use more than a small subset of a single period. The quality of a random number generator is defined on the randomness of each sequence, and on the length of the period. There exist tens, if not hundreds, of random number generators.

The performance of the generators can be investigated with tests such as DIEHARD [8] or TestU01 [9]. Good random number generators ought to pass these tests; or at least the user had better be aware of any
that they fail, so that he or she will be able to judge whether they are relevant to the case at hand. Good quality uniformly distributed random numbers, lying within a specified range (typically 0 to 1), are an essential building block for any sort of stochastic modeling or Monte Carlo computer work as they can be used to generate any other distribution.

A short description of the most common generators can be found in the Gnu Scientific Library documentation [10], also on-line.

4.4.2 Quasirandom numbers

Quasi-random sequences are sequences that progressively cover a $N$-dimensional space with a set of points that are uniformly distributed. Quasi-random sequences are also known as low-discrepancy sequences. The quasi-random sequence generators use an interface that is similar to the interface for random number generators, except that seeding is not required—each generator produces a single sequence.

Unlike the pseudo-random sequences, quasi-random sequences fail many statistical tests for randomness. Approximating true randomness, however, is not their goal. Quasi-random sequences seek to fill space uniformly, and to do so in such a way that initial segments approximate this behavior up to a specified density.

4.5 Numerical libraries

4.5.1 GSL - The GNU Scientific Library

The GNU Scientific Library (GSL) is an excellent numerical library for C and C++ programmers. It provides a wide range of mathematical routines such as random number generators, special functions and least-squares fitting. There are over 1000 functions in total with an extensive test suite. The GSL is written in C, and can be called from C, C++, and even Fortran programs. The GSL it’s free software under the GNU General Public License.

4.5.2 The origins: the NAG library

The NAG Numerical Library is the mainstream, commercial, cumbersome precursor of GSL. It’s a library of numerical analysis routines, containing nearly 2,000 mathematical and statistical algorithms. Areas covered by the library include linear algebra, optimisation, quadrature, the solution of ordinary and partial differential equations, regression analysis, and time series analysis. This library is developed and sold by The Numerical Algorithms Group.

The original version of the NAG Library was written in Algol 60 and Fortran. It contained 98 user-callable routines, and was released for the ICL 1906A and 1906S machines on October 1, 1971. The first partially vectorized implementation of the NAG Fortran Library for the Cray-1 was released in 1983, while the first release of the NAG Parallel Library (which is specially designed for distributed memory parallel computer architectures) was in the early 1990s. Mark 1 of the NAG C Library was released in 1990. In 1992, the Library incorporated LAPACK routines for the first time; NAG had been a collaborator in the LAPACK project since 1987.

4.6 Parallelism

Over the last decade, CPUs have progressively switched towards more and more parallel computing. Parallelism, that is splitting a problem in tasks that can be performed simultaneously, can dramatically reduce the computation speed by order of magnitudes. Several models of parallelisation exist, depending on the problem being studied. There is not a general parallelisation strategy, and often the best solution for a given problem does not work with another. It also depends on the hardware we intend to run our software on. For instance, parallel programming technologies such as MPI are used in a distributed
**Fig. 5:** Coverage of the unit square. Left for quasi-random numbers. Right for pseudo-random numbers. From top to bottom. 10, 100, 1000, 10000 points. (Source: Wikipedia)
computing environment (dedicated clusters of computer connected by fast links to be used), while multi-
threads programs are limited to a single computer system: all threads within a process share the same
address space. In some specific cases, like for instance lattice QCD calculations, the physical imple-
mentation of the cluster matches the lattice structure of the problem in study: the cluster’s nodes are
physically connected to their first neighbors to reproduce the lattice structure being studied.

In this section we will give a few examples that should illustrate the variety of solutions in
accelerator-physics problems.

4.6.1 Embarrassingly parallel problems
This is the class of problems that gives more satisfaction and that better benefits from parallelism. Em-
arrassingly parallel problems are those where a large number of tasks need to be performed, with each
single task being completely independent of the others. In this case, all tasks can in principle be per-
formed simultaneously, and the speed up factor is proportional to the number of processes that one can
run in parallel.

The simulation of accelerator imperfections for instance, where hundreds of different random
misalignment configurations are tested in order the sample the response of a system to random installation
ersors, provides a perfect example of embarrassingly parallel problem.

In this case, no modifications are needed to the simulation code -which can even be sequential
and run on a single core-, and all the random configurations can be spawn to hundreds of different
computer and run independently and simultaneously. Practically, this can be achieved using software
solutions called “job schedulers”, like for instance HTCondor (which is in use at CERN), that help
running computations on a pool of multiple computers (or a “farm” of computers).

4.6.2 MPI
For computational problems that can be solved with parallel algorithms, several approaches exist. One
such approaches, typical of massively parallel problems, consists in designing and writing to run on
clusters of computers. In this case, the software must be conceived, designed, and written to function
on parallel computing architectures (e.g., a cluster of computers, called nodes, connected between each
other using high-speed low-latency connections).

A standardized and portable solution to handle message-passing (and data sharing) between nodes
of a cluster is the “Message-Passing Interface” (MPI). MPI is not a library in itself. The MPI is a standard
that defines the syntax and semantics of a set of routines useful to a wide range of users writing portable
message-passing programs in C, C++, and Fortran. There exist several open-source implementations of
MPI, which fostered the development of a parallel software industry, and encouraged development of
portable and scalable large-scale parallel applications. Two well-established MPI implementations are
“Open MPI” and “MPICH”.

In spite of the unequivocal advantage of controlling from a single code a cluster of computers,
writing MPI code requires a considerable effort. A code must be conceived and designed to run on MPI
from the beginning. Adapting an existing code to run on MPI is a nearly impossible task. To give an
example, follows an example of “Hello world” written in C using MPI:

```c
#include <mpi.h>
#include <stdio.h>

int main()
{
    // Initialize the MPI environment
    MPI_Init(NULL, NULL);
    // Get the number of processes
    // ...more code...
}
```
```c
int world_size;
MPI_Comm_size(MPI_COMM_WORLD, &world_size);

// Get the rank of the process
int world_rank;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

// Get the name of the processor
char processor_name[MPI_MAX_PROCESSOR_NAME];
int name_len;
MPI_Get_processor_name(processor_name, &name_len);

// Print off a hello world message
printf("Hello world from processor %s, rank %d out of %d processors\n", processor_name, world_rank, world_size);

// Finalize the MPI environment.
MPI_Finalize();
```

### 4.6.3 OpenMP

Hacking an existing code to make it parallel, it’s best done with OpenMP. OpenMP is a programming interface that supports multi-platform shared-memory multiprocessing programming in C, C++, and Fortran. In simpler words, it makes programs run in parallel on multi-cores computers, exploiting the multi-threaded architecture of modern CPUs. OpenMP is best understood by looking at an example:

```c
int main()
{
    int a[100000];

    #pragma omp parallel for
    for (int i = 0; i < 100000; i++) {
        a[i] = 2 * i;
    }

    return 0;
}
```

In this code, the `#pragma omp parallel for` is used to fork additional threads and carry out the work enclosed in the construct in parallel. This specific pragma applies to loops that are “embarrassingly parallel” (each iteration is independent of the others), however more sophisticated schemes exist to handle more complex cases.

### 4.6.4 C++ threads

When designing and writing a code from scratch, with a multi-core / multi-threaded architecture in mind (like basically any computer today, and even a smart phone), the best solution to handle parallelism is to use functions and constructs offered by the language of choice. This ensures portability across systems and better integration within the language. Modern C++, since the standard version C++11, offers a set of classes to handle parallelism, synchronisation, and data exchange between threads. This revolves around the class `thread`, defined in the include file `<thread>`.

For instance, in C++, one can easily implement a generic “parallel for” through the use of lambda functions and “functors”. Here is the header file that implements it:

```c
#ifndef parallel_for.hh
#define parallel_for.hh

#pragma omp parallel for
for (int i = 0; i < 100000; i++) {
    a[i] = 2 * i;
}

return 0;
```
#include <algorithm>
#include <cstdlib>
#include <vector>
#include <thread>

template <typename Function>
size_t parallel_for(size_t Nthreads, int begin, int end, Function func)
{
    const int size = end - begin;
    // func must be in the form: func(thread, start, end)
    if (size < Nthreads)
        Nthreads = size;
    if (Nthreads) {
        std::vector<std::thread> threads(Nthreads - 1);
        for (size_t thread = 1; thread < Nthreads; thread++) {
            int from = begin + thread * size / Nthreads;
            int to = begin + (thread + 1) * size / Nthreads;
            threads[thread - 1] = std::thread(func, thread, from, to);
        }
        func(0, begin, begin + size / Nthreads);
        for (auto &t: threads)
            t.join();
    }
    return Nthreads; // returns the effective number of threads used
}

#include "parallel_for.hh"

int main()
{
    std::vector<int> a(100000);
    auto do_something = [&] (size_t thread, int start, int end) {
        for (int i = start; i < end; i++) {
            a[i] = 2 * i;
        }
    };
    size_t max_number_of_threads = std::thread::hardware_concurrency();
    parallel_for(max_number_of_threads, 0, a.size(), do_something);
    return 0;
}

4.6.5 C threads

The C language does not support multi-threading natively. To achieve parallelism in C one can use the Portable Operating System Interface (POSIX) set of libraries. POSIX is a family of standards specified by the IEEE Computer Society for maintaining compatibility between operating systems.

POSIX defines the both system- and user-level application programming interfaces. The POSIX thread libraries are a standards based thread API for C/C++. It allows one to spawn a new concurrent process flow. It is most effective on multi-processor or multi-core systems where the process flow can be
scheduled to run on another processor thus gaining speed through parallel or distributed processing.

The use of POSIX is quite complicated, and a detailed explanation goes beyond the scope of these lectures. Easier solutions exist using Octave, Python, C++, or even just including a rationalisation of the simulation procedure and some useful shell commands (See for example to use of FIFOs, below.)

4.6.6 GPU Computing: OpenCL / CUDA

The advent of powerful graphics cards has opened a new chapter in the history of high-performance computing. GPU computing is the use of a GPU (graphics processing unit) as a co-processor to accelerate general-purpose scientific and engineering computing.

The GPU accelerates applications running on the CPU by offloading some of the compute-intensive and time consuming portions of the code. The rest of the application still runs on the CPU. From a user’s perspective, the application runs faster because it’s using the massively parallel processing power of the GPU to boost performance. This is known as "heterogeneous" or "hybrid" computing.

The most striking difference between CPUs and GPUs is the number of available cores. High-end CPUs like Intel Xeon processors feature a number of cores up to 30s. In nVidia’s and AMD’s current generation of high end GPUs has nearly 4,000 cores. Two solutions exist to access the enormous computational power of GPUs, OpenCL and CUDA.

OpenCL OpenCL (Open Computing Language) is a framework for writing programs that execute across heterogeneous platforms consisting of central processing units (CPUs), graphics processing units (GPUs), digital signal processors (DSPs), field-programmable gate arrays (FPGAs) and other processors or hardware accelerators. OpenCL specifies directives for programming these devices and APIs (application programming interfaces) to control the platform and execute programs on the compute devices. OpenCL provides a standard interface for parallel computing using task- and data-based parallelism.

OpenCL is an open standard maintained by the non-profit technology consortium Khronos Group. Conformant implementations are available from Altera, AMD, Apple, ARM, Creative, IBM, Imagination, Intel, Nvidia, Qualcomm, Samsung, Vivante, Xilinx, and ZiiLABS.

CUDA (an acronym for Compute Unified Device Architecture) is a proprietary model created by Nvidia to program Nvidia GPUs for general purpose processing. The CUDA platform is a software layer that gives direct access to the GPU’s virtual instruction set and parallel computational elements, for the execution of compute kernels.

The challenge is that GPU computing requires the use of graphics programming languages like OpenGL and CUDA to program the GPU. And one has to rewrite from scratch scientific applications.

4.6.6.1 Parallel Octave and Python

Octave and Python can provide limited parallelism through a dedicated package and a module. In Octave, an easy solution is to use the parallel package, which is available from the Octave Sourceforge website. We refer to the online documentation of this package for more details.

In Python, the multiprocessing module is used to run independent parallel processes by using subprocesses (instead of threads). It allows one to leverage multiple processors on a machine, which means, the processes can be run in completely separate memory locations.

4.6.7 Parallelism in the shell, using FIFOs

A certain degree of parallelism can also be obtained without even using parallel codes, just using the Linux command line wisely.

Imagine we are simulating the two arms of electron-positron linear collider, and intend to simulate the collision once the two bunches arrive at the interaction point (IP). It is clear that the tracking of the
two bunches along their respective linacs can be performed simultaneously (just like in the real world!). Only once the two bunches reach the IP they can be passed to the beam-beam simulation code.

A convenient and elegant solution to perform these operations and synchronize them, without modifying a single line of code, is to make use of the powerful “named pipes”, of FIFO. The term “FIFO” refers to its first-in, first-out functioning mode, and it’s a mechanism, in Unix and Linux, used to enable inter-process communication within the same machine. FIFOs allow two-way communication, using a special “file” as a “meeting point” between two processes.

So, here’s an example of creating a named pipe.

```bash
$ mkfifo mypipe
$ ls -l mypipe
prw-r-----. 1 myself staff 0 Jan 31 13:59 mypipe
```

Notice the special file type designation of "p" and the file length of zero. You can write to a named pipe by redirecting output to it and the length will still be zero.

```bash
$ echo "Can you read this?" > mypipe
$ ls -l mypipe
prw-r-----. 1 myself staff 0 Jan 31 13:59 mypipe
```

So far, so good, but hit return and nothing much happens.

```bash
$ echo "Can you read this?" > mypipe
```

While it might not be obvious, your text has entered into the pipe, but you’re still peeking into the input end of it. You or someone else may be sitting at the output end and be ready to read the data that’s being poured into the pipe, now waiting for it to be read.

```bash
$ cat mypipe
Can you read this?
```

Once read, the contents of the pipe are gone.

Now, the tracking along the linacs has, as ultimate result, the task to leave two files on disk, say electrons.dat and positrons.dat, containing the two bunches at the IP. The beam-beam code will read these files as inputs and compute the luminosity.

If we create two FIFOs named electrons.dat and positrons.dat our simulation codes with automatically use these FIFOs to send the electron and the positron bunches from the linacs to the beam-beam at the IP, without writing a single byte on disk.

Let’s see how to do it. First, let’s create the FIFOs:

```bash
$ mkfifo electrons.dat
$ mkfifo positrons.dat
```

Then, we run our simulation (notice the ‘&’, to run the two tracking simultaneously):

```bash
$ track electron_linac.m > electrons.dat &
$ track positron_linac.m > positrons.dat &
$ beam_beam electrons.dat positrons.dat > luminosity.dat
```

The beam-beam simulation will diligently wait for the bunches to traverse the linacs before it starts computing the luminosity.

### 4.7 Advanced programming

#### 4.7.1 Intrinsics

For those who try to squeeze CPU cycle The key for speed, in modern CPUs, is vectorisation. By explicit vectorisation one can access specific instruction sets like MMX, SSE, SSE2, SSE3, AVX, AVX2. For those interested, we suggest to browse the Intel Intrinsics Guide page [link].
4.7.2 Programmable user interfaces: SWIG

SWIG is a software development tool that connects programs written in C and C++ with a variety of high-level programming languages. SWIG is used with different types of target languages including common scripting languages such as Javascript, Perl, PHP, Python, Tcl and Ruby. The list of supported languages also includes non-scripting languages such as C#, D, Go language, Java including Android, Lua, OCaml, Octave, Scilab and R. Also several interpreted and compiled Scheme implementations (Guile, MzScheme/Racket) are supported. SWIG is most commonly used to create high-level interpreted or compiled programming environments, user interfaces, and as a tool for testing and prototyping C/C++ software. SWIG is typically used to parse C/C++ interfaces and generate the “glue code” required for the above target languages to call into the C/C++ code. SWIG can also export its parse tree in the form of XML. SWIG is free software and the code that SWIG generates is compatible with both commercial and non-commercial projects.

5 Helper tools

Besides, Octave, Python, and Maxima, the Linux/Unix environments offer a number of helper tools that can greatly help a scientist in his/her daily job. Here we list a few of them, but we are open to suggestions if your favorite tool is not in this list.

5.1 Shell commands

These tools are all “command-line” based, that is, they can simply be run in the command line terminal.

5.1.1 units - conversion program

Units is a great tool: it knows the value of the most important scientific constants, it performs units conversions, and most of all it’s a calculator with units. Let’s consider an example: let’s computer the average power of a 50 Hz, 300 pC single-bunch charge, 15-GeV beam. One inputs in units the following quantities:

\[
\begin{align*}
\text{You have: } & 300 \text{ pC } \times 15 \text{ GV } \times 50 \text{ Hz} \\
\text{You want: } & \text{W}
\end{align*}
\]

and units returns

\[
\begin{align*}
300 \text{ pC } \times 15 \text{ GV } \times 50 \text{ Hz} &= 225 \text{ W} \\
300 \text{ pC } \times 15 \text{ GV } \times 50 \text{ Hz} &= (1 / 0.004444444444444444) \text{ W}
\end{align*}
\]

The option -v makes the output of units more verbose and more clear.

Units it’s an excellent tool that is too often undervalued. The consistent use of it strengthen and simplifies the writing of any physics-based code. Let’s compute for example the electric force experienced by two charged particles at a distance, for example. The force can easily be written as:

\[
F = K \frac{Q_1 \cdot Q_2}{d^2} \text{[eV/m]}
\]

where \(Q_1\) and \(Q_2\) are obviously the charges of the two particles involved, and \(d\) is their relative distance; \(K = \frac{1}{4\pi\epsilon_0}\) is the coupling constant. We choose to use eV/m as units of the force, because expressing the force, e.g. in Newton, would certainly lead to very small numbers. As a general rule, a good choice is to pick units that make the quantities at play be small numbers whose integer part is larger than 1. We use \(e\) (the charge of a positron) as the units of charge, and mm as the units of distance. Units helps us compute the numerical value of the coupling constant \(K\), in the desired units:
You have: \( e^2 / 4 \pi \epsilon_0 \text{ mm}^2 \)
You want: eV/m
\[ e^2 / 4 \pi \epsilon_0 \text{ mm}^2 = 0.00143996457846902 \text{ eV/m} \]
\[ e^2 / 4 \pi \epsilon_0 \text{ mm}^2 = (1 / 694.4615417756247) \text{ eV/m} \]

Therefore our code will look like:

```plaintext
Q1 = -1; % the charge of an electron [e]
Q2 = +1; % the charge of a proton [e]
d = 1; % the relative distance [mm]
K = 0.00143996457846902; % the coupling constant [e^2 / 4 \pi \epsilon_0 \text{ mm}^2]
```

The result being:

```plaintext
F = K * Q1 * Q2 / (d*d) % the force [eV/m]
```

### 5.1.2 bc - An arbitrary precision calculator language

`bc` is a shell calculator that supports arbitrary precision numbers with interactive execution of statements. There are some similarities in the syntax to the C programming language. A standard math library is available by command line option. If requested, the math library is defined before processing any files.

In `bc`, the variable `scale` allows one to select the total number of decimal digits after the decimal point to be used:

```plaintext
$ bc
bc 1.06
Copyright 1991-1994, 1997, 1998, 2000 Free Software Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type `warranty'.
scale=1
sqrt(2)
1.4
scale=40
sqrt(2)
1.4142135623730950488016887242096980785696
```

### 6 Suggested literature

There are a number of classic books that every scientist dealing with numerical calculations should know. Here we list some of our favourites:

- Donald Knuth, “The Art of Computing programming”, is a comprehensive monograph written by computer scientist Donald Knuth that covers many kinds of programming algorithms and their analysis. Knuth began the project, originally conceived as a single book with twelve chapters, in 1962. The four volumes are:
  - Volume 1 – Fundamental Algorithms: Basic concepts, Information structures
  - Volume 2 - Seminumerical Algorithms: Random numbers, Arithmetic
  - Volume 3 – Sorting and searching: Sorting, Searching
  - Volume 4 - Combinatorial searching: Combinatorial searching.

*En passant*, Donald Knuth is the creator of \( \text{TeX} \), the typesetting system at the base of \( \text{LaTeX} \).

- W. Press, S. Teukolsky, W. Vetterling, and B. Flannery, “Numerical Recipes: The Art of Scientific Computing”, is a complete text and reference book on scientific computing. In a self-contained manner it proceeds from mathematical and theoretical considerations to actual practical computer routines. Even though its routines are nowadays available in libraries such as GSL.
or NAG, this book remains the most practical, comprehensive handbook of scientific computing available today. Cambridge University Press.

– Abramowitz and Stegun, “Handbook of Mathematical Functions with Formulas”. Since it was first published in 1964, the 1046-page Handbook has been one of the most comprehensive sources of information on special functions, containing definitions, identities, approximations, plots, and tables of values of numerous functions used in virtually all fields of applied mathematics. At the time of its publication, the Handbook was an essential resource for practitioners. Nowadays, computer algebra systems have replaced the function tables, but the Handbook remains an important reference source for finite difference methods, numerical integration, etc. A high quality scan of the book is available at the University of Birmingham, UK [link].

– Olver, F., Lozier, D., Boisvert, R. and Clark, C. (2010), “The NIST Handbook of Mathematical Functions”. This is a modern version of the Abramowitz-Stegun, and is comprehensive collection of mathematical functions, from elementary trigonometric functions to the multitude of special functions.

– Zyla, P. A., et al., “Review of Particle Physics”, Oxford University Press. s. A huse summary of particle physics, enriched with extremely useful reviews of topics such as particle-matter interaction, probability, Monte Carlo techniques, and statistics.

– George B. Arfken, “Mathematical Methods for Physicists”. This is a thorough handbook about mathematics that is useful in physics. It is a venerable book that goes back to 1966; this seventh edition (2012) adds a new co-author, Frank E. Harris. It is pitched as a textbook but because of its size and breadth probably works better as a reference. Very Good Feature: all the examples are real problems from physics. Note that it is not a mathematical physics book, though: it quotes the models but does not develop them.

– Hockney, “Computer Simulation Using Particles”. This is another venerable reference in scientific research, including the simulation of systems through the motion and the interaction of particles. This book provides an introduction to simulation using particles based on the PIC (Particle-in-cell), CIC (Cloud-in-cell), and other algorithms and the programming principles that assist with the preparations of large simulation programs.

References
[1] J. W. Eaton et al. GNU Octave: a high-level interactive language for numerical computations. https://octave.org.
[2] François Delebecque and Claude Gomez. Scilab. https://www.scilab.org.
[3] Guido Van Rossum and Fred L. Drake Jr. Python reference manual. Centrum voor Wiskunde en Informatica Amsterdam, 1995.
[4] wxmaxima. http://wxmaxima-developers.github.io/wxmaxima.
[5] Lapack – linear algebra package. http://www.netlib.org/lapack.
[6] Blas – basic linear algebra subprograms. http://www.netlib.org/blas.
[7] Donald E. Knuth. Volume 2: Seminumerical Algorithms. The Art of Computer Programming, page 216, 1998.
[8] G. Marsiglia. A current view of random numbers. Computer Science and Statistics The Interface, pages 3–10, 1985.
[9] P. A. Zyla et al. Oxford University Press : Review of Particle Physics, 2020-2021. PTEP, 2020(8):083C01, 2020.
[10] Mark. Galassi. GNU scientific library: reference manual. Network Theory, 2009.