Computational Physics I
(PHY 4140/5340)

Instructor: Mykyta Chubynsky

chubynsky@gmail.com

Thanks for responding to the survey!

TA: Tyler Shendruk

tshendruk@gmail.com

A bit about myself:

Research associate in Prof. Gary Slater's group in physics department.

Use theoretical and computational methods to study soft matter and polymer systems with emphasis on biological applications.

Electrophoresis of biomolecules, e.g., DNA, as a separation method.

Done molecular dynamics simulations of electrophoresis of rigid rods in an array of obstacles. Also, studied a model of free-solution electrophoresis, in which the problem was essentially reduced to inverting a matrix.

Also, done some Monte Carlo simulations and worked on optimizing the Lattice Monte Carlo approach for simulating diffusion. Can be recast in terms of a finite-difference scheme for solving the diffusion equation.

Worked as a postdoc with Normand Mousseau at Université de Montréal on a hybrid molecular dynamics – Monte Carlo method for simulating complex systems (like disordered solids).

Did my Ph.D. with Michael Thorpe at Michigan State using a graph-theoretical algorithm to study rigidity of elastic networks with application to glasses.

chubynsky.info

My first time teaching this (or any!) course. Any feedback is very welcome!

Use of computers in physics

1. Slides for presentations.

2. Writing papers.

3. Searching and retrieving literature.

4. Controlling experimental equipment and processing experimental results.

   Can be as simple as data fitting (covered in Comp. Phys. II), more complex like tracking colloidal particles under the microscope, or extremely complex, like finding the Higgs boson

5. Doing analytical calculations (e.g., integration) using, e.g., Maple (symbolic computation)

6. Using the computer to solve theoretical problems that cannot be solved analytically

Computational physics

Problems unsolvable analytically can sometimes be solved by very simple means

Consider an equation $x = \cos x$

Choose x, calculate its cosine. If we are lucky and cos x = x, we are done

x = 1, cos (1) = 0.5403 ... No luck. Let's try another number. Why not the one we ended up with? cos(0.5403...) = 0.8575... Keep doing anyway. Neighbouring numbers in the sequence get closer and closer to each other.

After clicking the cos button many times, get 0.7390851332 ...

A bad method, because need the equation to be in the form x = f(x), but more importantly, such iterations don't always converge. Try x = tan x, with one obvious solution x = 0. Turns out we need |f '(x)| < 1.

How can we do this more reliably? Solving algebraic equations (root finding) will be one of the topics considered in the course.

Other topics:

Errors

Both due to finite precision of representation of numbers on a computer and due to approximations of the algorithms

Numerical differentiation

Sure, if the function is given, we can usually do it analytically, but what if the function is itself obtained numerically? Also, useful when constructing algorithms for solving differential equations

Interpolation

Given the values of a function in a few points, what are the values in between? For some reason, mentioned in 5340 description only, but anyway, useful for ...

Numerical integration

Includes Fast Fourier Transform

Other topics:

Ordinary differential equations with initial conditions

E.g., Newton's equations of motion. Molecular dynamics algorithm (intro, mostly in Comp. Phys. II). Nonlinear dynamics.

ODEs with boundary conditions

E.g., stationary Schrödinger eq. in 1D. To solve them, it is useful to know about ...

Linear algebra algorithms

Solving systems of linear equations, inverting matrices, eigenvalue problems. A huge topic, actually.

Partial differential equations

Schrödinger eq., Poisson eq. for the electrostatic potential, wave eq.

Finding maxima and minima

Simply equate the derivatives to 0; but there are other approaches

Computational Physics II next semester (probably Lora Ramunno)

Mostly stochastic methods, i.e., those that use (pseudo)random number generation.

These can be used to study random or quasi-random systems (e.g., Brownian motion or a disordered material), but they can also be used to study deterministic problems. E.g., Monte Carlo integration is more efficient than the methods we will consider for high-dimensional integrals (above ~4D); to find a global minimum of a high-dimensional function, it is useful to explore the space randomly (simulated annealing or genetic algorithms).

Some physical phenomena can be studied by both deterministic and stochastic methods. For instance, diffusion is a random process and can be simulated by moving particles at random. But the density of diffusing particles is described by a deterministic PDE and can be computed by deterministic methods that we will consider in this course. Simulation vs. computation.

This sounds like an applied math course, so where is physics?

In the labs and projects! You will implement these methods in code and use it to solve physics problems.

This way, the course should achieve 3 goals:

1. Give some knowledge about numerical methods, their strengths and weaknesses and how to analyze them;

2. Help gain some experience doing scientific programming;

3. Learn some physics by solving some interesting problems that could not be part of your other classes, because they are not solvable analytically.

There will be 5-6 1-week homeworks (aka “labs”) and 3-4 2-week homeworks (aka “projects”). Graduate students will have more tasks and will need to do one more project. You can do the homeworks on your own, but if you need some help, this is what the “lab” on Tuesdays is for. In principle, you don't need to come to the lab if you know how to do the problems, but who knows, some interesting discussions may arise.

Evaluation

1-week labs – 30%

2-week projects – 40%

Take-home final exam - 30%

The final is, basically, yet another homework, which is why the weight is the lowest legally possible.

The details on the homework requirements, how to submit them, etc. are posted at http://chubynsky.info/teaching/CompPhys13/ . If there are any questions, suggestions, objections, etc., please let me know next time we meet on Monday. This is when the first homework will be given.

Textbooks and other materials

No required textbook. A lot of free material online. Some textbooks and lecture notes (including some free ones) are mentioned in the information posted online. Some textbooks are accessible on campus or via proxy.

One particularly useful reference is

Press, Teukolsky, Vetterling, Flannery, Numerical Recipes in (C, Fortran etc.)

http://www.nr.com/

Older editions are available on the website for free. There are code examples, but to actually use them, you need to pay for a license (even if you bought the book), and you are not allowed to use any code not written by you for the course anyway. But the explanations of the methods are very good.

There are several computational physics journals, but one particularly recommended is Computing in Science and Engineering (CiSE) published jointly by IEEE Computer Society and American Institute of Physics

http://cise.aip.org/

Even though computational physics is done today using computers, many numerical methods were invented and used to solve physics problems long before computers.

Euler (1707-1783) and Runge (1856-1927) – Kutta (1867-1944) methods for ODEs

Newton (1643-1727) – Raphson (~1648–~1715) method of solving algebraic equations

A famous early example was a calculation of the time of return of Halley's Comet in 1758 by Clairaut, Lalande and Lepaute – also an early example of parallel computing! Took into account interactions with Jupiter and Saturn. Took them 5 months.

Artillery table computations during the 1st and 2nd World Wars. Bomb calculations at Los Alamos.

Fermi, Pasta, Ulam (1955) – studied a chain of 32 beads connected by nonlinear springs. Quasiperiodic instead of expected chaotic behaviour. Relation to solitons and integrable equations.

Alder and Wainwright (1957) – liquid-solid transition in hard-sphere system (108 particles)

For some systems, people can do more than a billion particles now.

Contributions to nearly all areas of physics now, but also applications outside physics: engineering, weather forecasting, video games ...

Some people say now that computational physics is the third way of doing physics, distinct from both theory and experiment. Perhaps a matter of taste and background of a particular researcher. But there are commonalities and differences with both experiment and theory:

Like in theory, ultimately, a theoretical model is considered and one can include and exclude different effects at will, something often impossible experimentally.

On the other hand, the way results are obtained and processed has much in common with experiment: rather than getting an analytical formula, computational physicists obtain sets of data by tweaking various parameters and then process them in much the same way as experimentalists, by fitting, etc. Often, especially in stochastic simulations, there is random noise, and one needs to get enough statistics to see the signal.

Despite all the successes of computational physics, analytical calculations are, of course, still necessary. Ultimately, numerical solutions are always approximate and, while it is often possible to estimate the error, it is usually impossible to be completely sure that this estimate is correct. So theory can serve as a benchmark. In your problems, you will be asked to compare to theory and in general it is a good idea to do this as much as possible.

Interestingly, in many cases theory and computation are complementary, because theory usually works in some limit, when something is large or small, but computation is the easiest when all parameters are of the same order of magnitude. In particular, finite-size effects are very often quite important in computation.

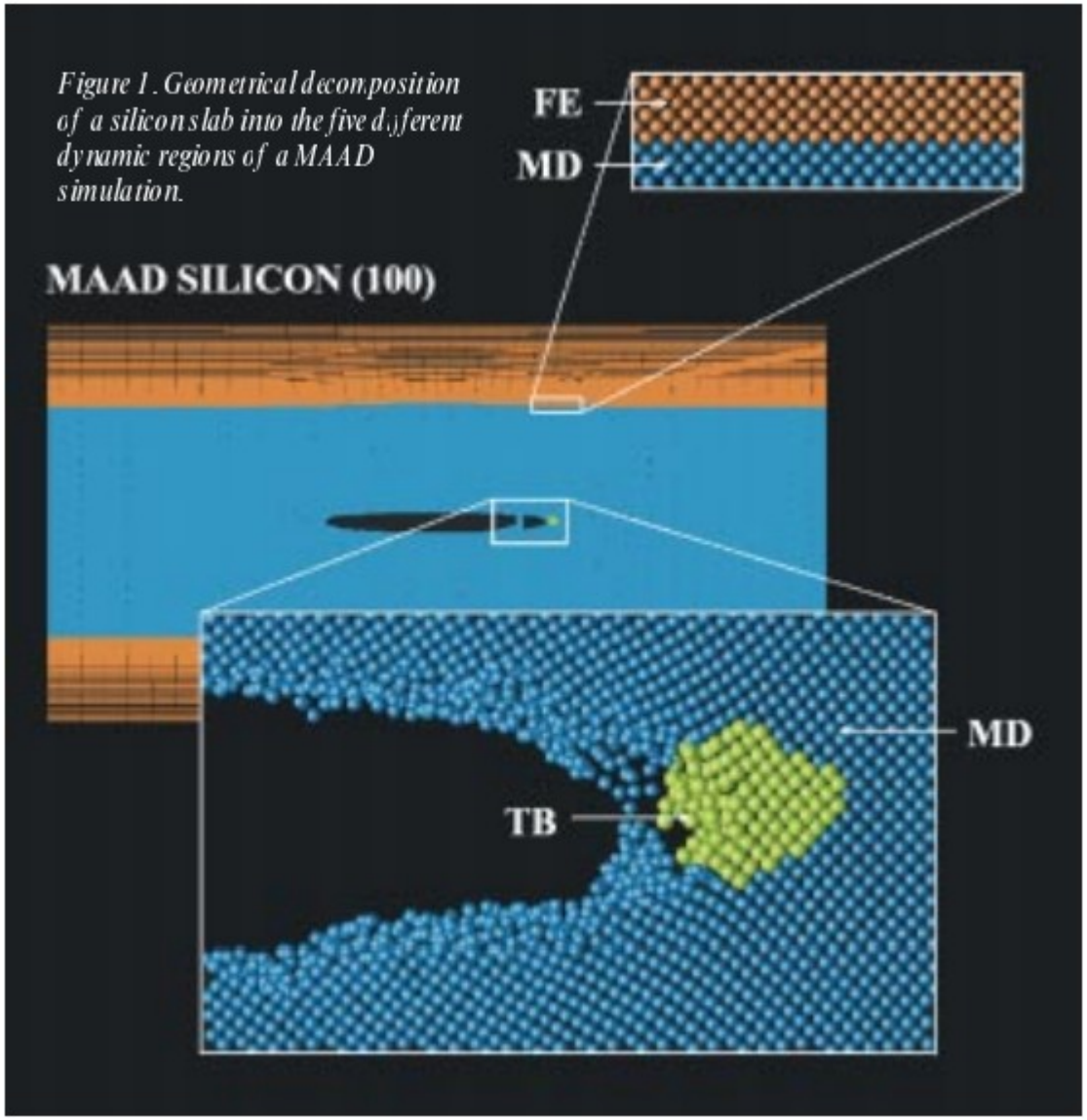
Also, an analytical solution (when it is possible) is just nicer to deal with than some data set!

Even though it may seem based on the outline of this course that all computational methods are just methods of solving mathematical problems developed by mathematicians and physicists just use them, this is certainly not the case. A lot of problems are too hard even for computers and physical intuition is needed to simplify them.

Quantum systems: for a system of N particles one needs to solve a 3N-dimensional Schrödinger equation. If this is done straghtforwardly, computational effort scales exponentially with the number of dimensions. Physicists came up with different approximations: Hartree-Fock, density-functional theory (DFT), quantum Monte Carlo methods.

J. Thijssen, Computational Physics

Multiscale simulations



Abraham et al., Computers in Physics 12 (1998) 538

Programming languages

Computer processors understand machine code instructions, but it is very hard to program in machine code (or even in assembly language, which has a more or less one-to-one correspondence to machine code, but more human-readable).

Therefore, people use programming languages. They need to be translated into machine code. There are 2 basic ways: interpretation and compilation. Simplistically, interpreters read programming language instructions one by one and translate into machine code directly on the fly executing the code. Compilers look at the whole code and translate it into an executable file, which is then executed. The second way in general produces more efficient code, because the compiler can optimize by rearranging different parts of the code, etc.

While in principle any language can be interpreted or compiled, it simply does not make sense to compile dynamic languages, because they cannot be optimized very much. In this sense, one can talk about interpreted and compiled languages.

http://benchmarksgame.alioth.debian.org/u32/benchmark.php?test=all&lang=ifc&lang2=gcc&data=u32

In scientific programming efficiency is usually very important and therefore compiled languages are generally preferred for computationally intensive code.

So there is a good reason why Fortran, C and C++ are preferred in scientific programming. Fortran used to be much faster, and there is a lot of older code in Fortran, but probably most new code these days is in C or C++. Java is a bit slower and less often used for scientific programming, but is probably OK.

Does this mean that Python is useless for scientific programming? No, because it has modules for numerical computation that are written in C. If most of the computation occurs in these modules, then the code will be fast. But if you actually try to implement the numerical algorithms in Python, the result will likely be slow.

Since much of this course is about implementing numerical algorithms, use of Python is discouraged.

On the other hand, projects we will be dealing with are sufficiently simple that even Python code should run fast. So it's really up to you and depends on whether you want to gain experience in a language used for more “serious” projects. But please, don't use features that implement numerical algorithms, like integration routines in SciPy.

And, of course, there is nothing wrong with using Python for plotting.

To be able to do the homeworks, you will need a compiler of your preferred language installed on your computer.

If you work under Unix (including Linux), then at least C and Fortran compilers should be installed.

For Windows, one possibility is Cygwin (cygwin.com), which is a Unix-like environment for Windows and includes many standard Linux tools, including the gcc compiler collection (Fortran, C, C++, Java and a few other languages). When installing Cygwin, you are given a choice what to install, so make sure you check the necessary boxes.

For Macs, supposedly, you can install Xcode and Xcode command tools.

http://www.mkyong.com/mac/how-to-install-gcc-compiler-on-mac-os-x/

You will also need to do plotting and data fitting. Under Linux and Cygwin, gnuplot and xmgrace can be used.

Representation of numbers

All numbers are represented as sets of 0's and 1's (bits).

Integer numbers (type `int` in C/C++, `integer` in Fortran): usually 32 bits (4 bytes). The most common representation is called two's complement. The last (leftmost) bit is the sign (0 for “+”, 1 for “-”). Positive numbers coincide with their standard binary representation:

1 is 00000 ... 001
2 is 00000 ... 010
3 is 00000 ... 011
. . . . . .
2^{31}-1 = 2147483647 is 01111 ... 111

For negative numbers, it is trickier:

-2^{31} = -2147483648 is 10000 ... 000
-2147483647 is 10000 ... 001
. . . . . .
-1 is 11111 ... 111

How much is 2147483647+1 ?

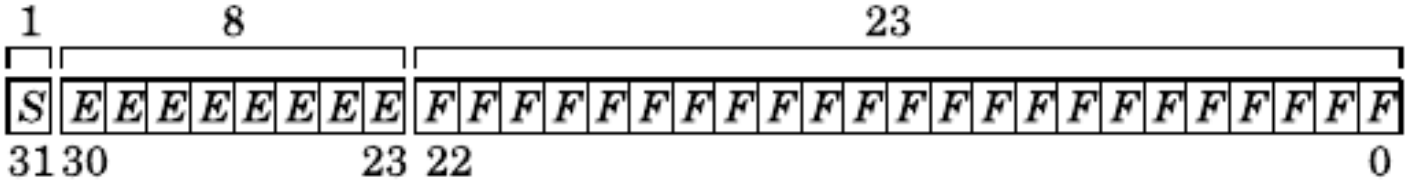
Representation of numbers

Fractional numbers are represented as floating-point numbers.

These representations are specified by a standard, IEEE 754-2008.

The most important types are 32-bit single precision (`float` in C, `real` in Fortran) and 64-bit double precision (`double` in C, `double precision` in Fortran).

Single precision:



S. Širca, M. Horvat, *Computational Methods for Physicists*

Still 1 sign bit, but the rest are divided into the exponent and the fraction (mantissa)

If E=255 and F ≠ 0, then "Not a Number" (NaN). If E=255 and F=0, then ±∞ (Inf). If 0<E<255, then $(1+F_{22}\times 2^{-1}+F_{21}\times 2^{-2}+\ldots)\times 2^{E-127}$; if E=0, then $(F_{22}\times 2^{-1}+F_{21}\times 2^{-2}+\ldots)\times 2^{-126}$

Max possible value is $2^{127}(2-2^{-23})\approx 3.4\times 10^{38}$. Min nonzero value is $2^{-149}\approx 1.4\times 10^{-45}$.

For double precision, 11 E bits and 52 F bits. Max is $2^{1023}(2-2^{-52})\approx 1.8\times 10^{308}$. Min is $2^{-1074}\approx 4.9\times 10^{-324}$.

Machine precision and round-off error

For single precision, the number >1 that is closest to 1 is $1+2^{-23}$. If we assume that the result of any operation is rounded to the nearest representable number, then the largest number ε such that $1+\varepsilon$ is still represented as 1 is $\varepsilon \approx 2^{-24} \approx 6.0\times 10^{-8}$. This is called machine precision or machine epsilon. (Note that sometimes it is defined as the difference between 1 and the next larger number, in which case it will be 2^{-23}.) For double precision, $\varepsilon \approx 2^{-53} \approx 1.1\times 10^{-16}$.

Any number can be represented with relative precision higher than ε. That is, number a will be represented as

$$a(1+\epsilon_a), \quad \text{where} \quad |\epsilon_a| \le \epsilon .$$

(round-off error)

But, of course, after some operations the round-off error may increase.

$$\frac{a(1+\epsilon_a)-b(1+\epsilon_b)-(a-b)}{a-b} = \frac{a\,\epsilon_a - b\,\epsilon_b}{a-b}$$

This can be as large as $\frac{a+b}{a-b}\epsilon$, i.e., the error may increase by a factor (a+b)/(a-b).

Can be large, if a-b<<a,b.

Avoid subtracting close numbers, if possible.

Suppose we have a product of *N* numbers.

$$\prod_{i=1}^{N} a_i(1+\epsilon_i) \approx \left(1+\sum_{i=1}^{N} \epsilon_i\right) \prod_{i=1}^{N} a_i$$

So the relative error will be the sum of the relative errors of the individual factors. If all numbers are different, it is likely that some errors are positive and some are negative. This is like a displacement of a random walk, which is $\propto \sqrt{N}$

Will be similar for a sum, if all terms are of roughly the same magnitude.

Of course, if the factors in the product or the terms in the sum are the same, then the error will go as *N*.

Let's now consider a real (but simple) numerical algorithm.

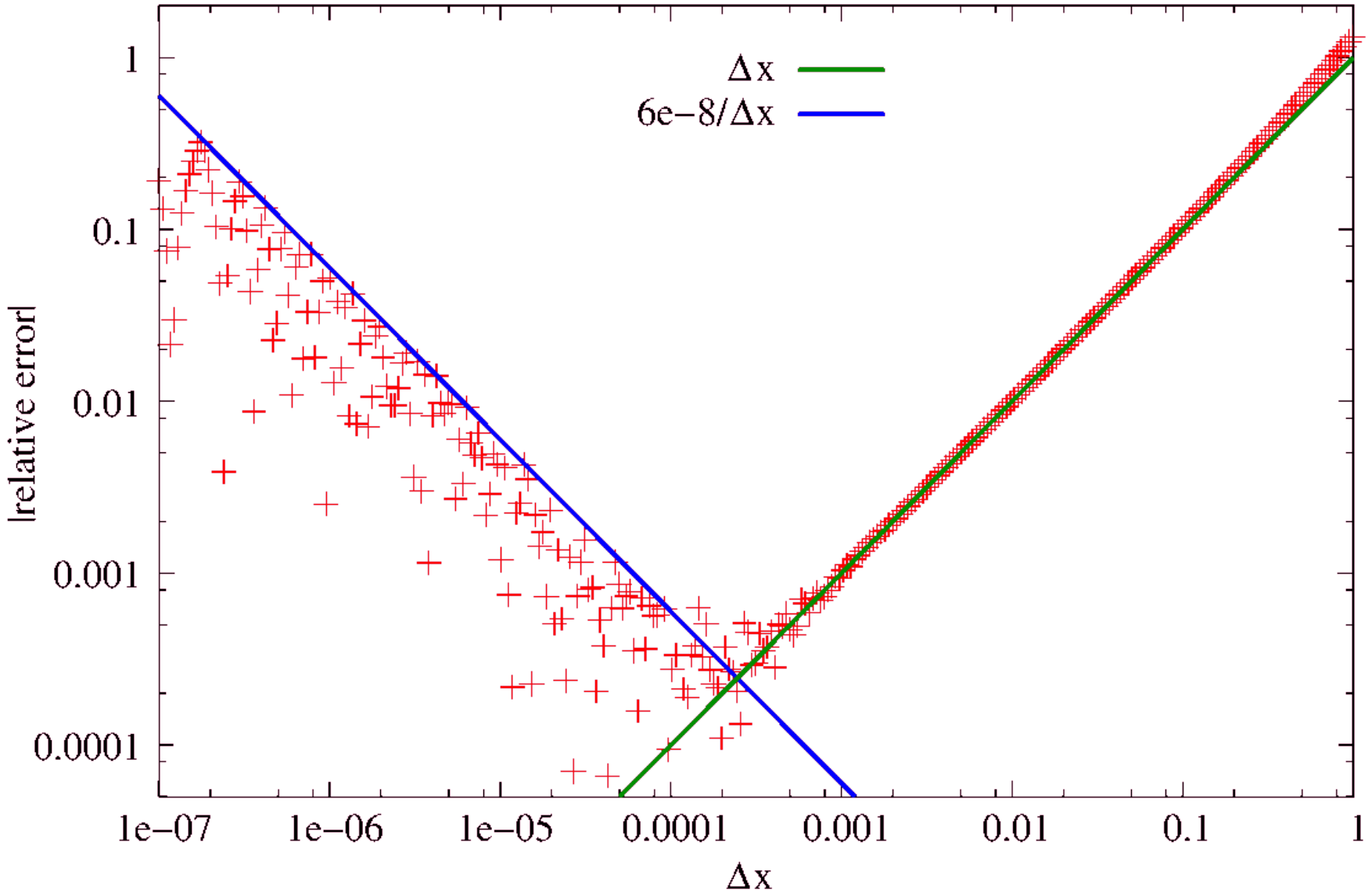
$$f'(x)=\lim_{\Delta x\to 0}\frac{f(x+\Delta x)-f(x)}{\Delta x}$$

Consider finite (not infinitely small) Δx. In principle, the result should approach the exact value as Δx decreases.

Consider $f(x)=x^3;\quad x=1.$ Know the exact result $f'(1)=3$.

Use single precision.

```
#include <stdio.h> /* standard input-output library */

#define X 1.0f /* the value of x where the derivative is taken */
#define V 3.0f /* the theoretical value of the derivative */

/* note that suffix "f" after a number means it's a float, rather than
   double */

float f(float); /* declare the function - the actual code is at the
                   bottom */

void main() {
  FILE *fp2;
  float dx,deriv;

  fp2=fopen("result.dat","w"); /* the file where the result is written*/

  dx=100.0f; /* the initial value of delta_x */

  while (dx>1.0e-7f) { /* loop until delta_x is small */
    deriv=(f(X+dx)-f(X))/dx; /* calculation of the derivative */
    fprintf(fp2,"%e %e\n",dx,deriv-V); /* output delta_x and error */
    dx*=0.95f; /* reduce delta_x by 5% for the next iteration */
  }

}

float f(float x) { /* the cubic function */
  return x*x*x;
}
```



As Δx decreases, the error first decreases, but then starts increasing again $\Rightarrow$ there is an optimal Δx.

The best achievable relative error is >> ε (actually, ~ $\sqrt{\varepsilon}$ in this case).

Two contributions to the error:

1. The algorithmic error.

$$f(x+\Delta x)=f(x)+f'(x)\Delta x+\frac{1}{2}f''(x)(\Delta x)^2+O((\Delta x)^3)$$

Absolute error $$\frac{f(x+\Delta x)-f(x)}{\Delta x}-f'(x)=\frac{1}{2}f''(x)(\Delta x)+O((\Delta x)^2)$$

Relative error $$\left[\frac{1}{2}f''(x)(\Delta x)+O((\Delta x)^2)\right]/f'(x).$$ In our case $$\Delta x+O((\Delta x)^2)$$

Also called the approximation error, or, somewhat confusingly, truncation error.

2. The round-off error.

$$\left|\frac{f(x+\Delta x)[1+\epsilon_{f(x+\Delta x)}]-f(x)[1+\epsilon_{f(x)}]}{\Delta x}-\frac{f(x+\Delta x)-f(x)}{\Delta x}\right|<\sim\frac{f(x)}{\Delta x}\epsilon$$

The relative error $$<\sim\frac{f(x)}{f'(x)\Delta x}\epsilon$$, which in our case is $$\frac{1}{3\Delta x}\epsilon$$

More careful consideration shows there is an extra factor of 3, thus $$\frac{1}{\Delta x}\epsilon$$

How to improve the accuracy

1. Obviously, double instead of single precision.

Reduces the round-off error; the algorithmic error stays the same.

2. A more accurate algorithm: centred difference instead of forward difference

$$f'(x) \approx \frac{f(x+\Delta x/2) - f(x-\Delta x/2)}{\Delta x}$$

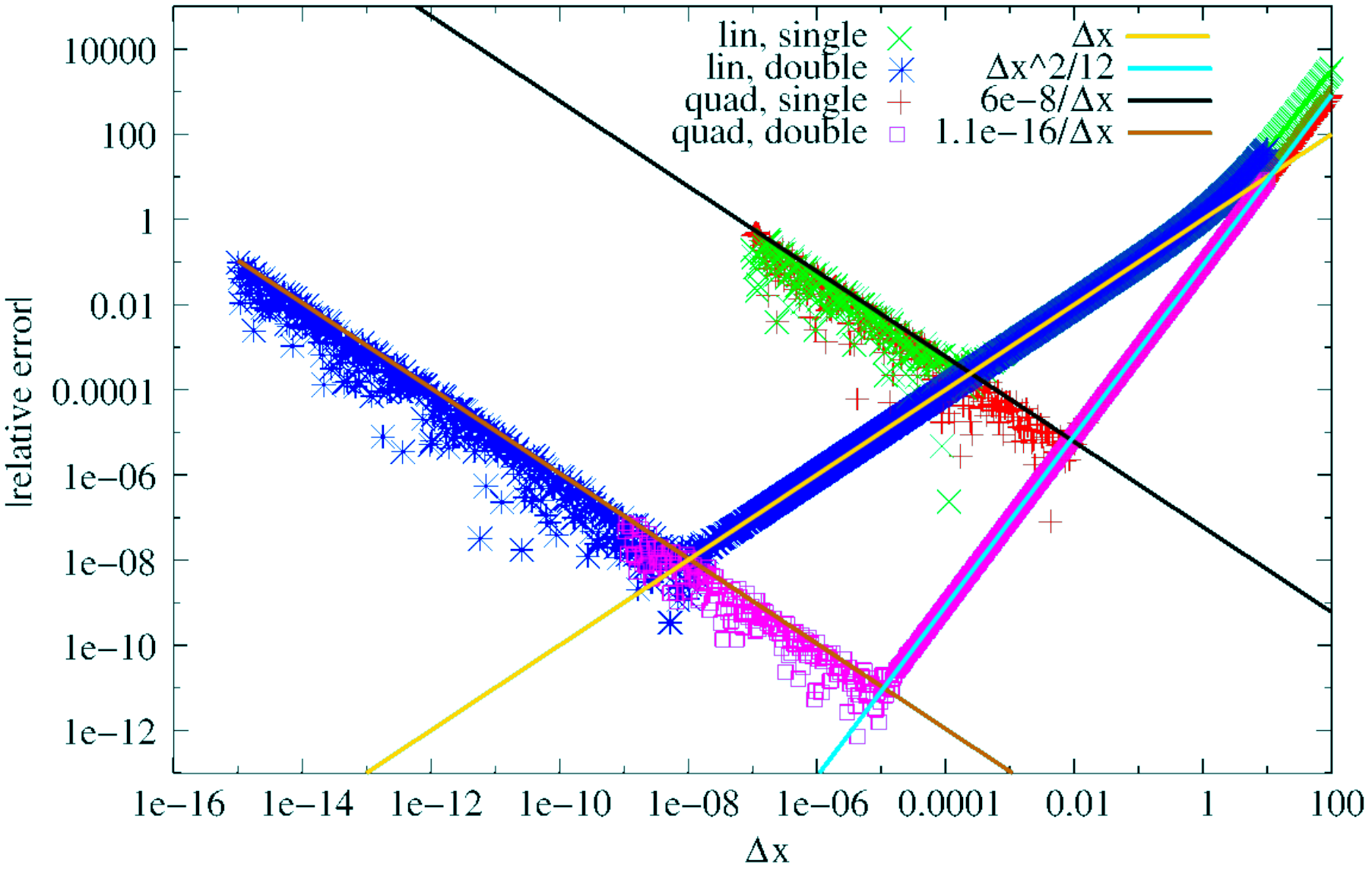
$$f(x \pm \Delta x/2) = f(x) \pm f'(x)(\Delta x/2) + (1/2) f''(x)(\Delta x/2)^2 \pm (1/6) f'''(x)(\Delta x/2)^3$$
$$+ (1/24) f^{(4)}(x)(\Delta x/2)^4 + O((\Delta x)^5)$$

$$\frac{f(x+\Delta x/2) - f(x-\Delta x/2)}{\Delta x} = f'(x) + (1/24) f'''(x)(\Delta x)^2 + O(\Delta x^4)$$

Quadratic instead of linear accuracy. Reduces the algorithmic error:

relative error is $\frac{f'''(x)}{24 f'(x)}(\Delta x)^2$, which in our case is $(\Delta x)^2/12$.

The round-off error stays the same.


lin, single
lin, double
quad, single
quad, double
Δx
Δx^2/12
6e−8/Δx
1.1e−16/Δx
|relative error|
10000
100
1
0.01
0.0001
1e−06
1e−08
1e−10
1e−12
1e−16
1e−14
1e−12
1e−10
1e−08
1e−06
0.0001
0.01
1
100
Δx