Fundamentals of Simulation Methods

Practice Exam

Janosh Riebesell

January 21st, 2016

Lecturer: Volker Springel

1 Short questions

State whether the following statements are right or wrong, and discuss in 1-2 sentences why this is the case. Note: There might be any number of correct statements.

(a) Molecular dynamics simulation: You are solving a molecular dynamics problem using a short-range potential and an algorithm that loops, for each particle individually, over all other simulation particles. You realise that this algorithm scales unfavourably with particle number N. How can you improve the scaling with N?

(1) Parallelising the algorithm such that several CPUs can compute the problem simultaneously.

(2) Using a search-mesh to preselect the neighbouring particles and only loop over them.

(3) Introducing a large enough softening to avoid close encounters between the particles.

(4) Using a tree algorithm to find interacting particles.

(a) Molecular dynamics

(1) False. Calculating dynamics form particle interactions is not an easily parallelizable problem since dynamics calculated by one node influence the calculations another node should be doing. The problem is that without loads of communications (or shared memory with frequent readouts) nodes would frequently be working with outdated particle coordinates.

But even if it were easily parallelizable, reducing computation time by increasing computational power is no way related to improving a method's scaling behavior with N.

(2) Correct. Depending on how large N is (and the efficiency of the mesh search) this can speed up calculations immensely. It does, however, present a potential source of inaccuracy if the short-range interactions aren't fully captured.

(3) False. In a short-range potential close encounters are necessary to produce interactions. Plus calculating interactions does not become computationally more expensive because two interacting particles are close together.

(4) Correct. Tree algorithms are a great way to speed up any interacting many-body simulation, but are particularly well suited for short-range potentials. In general, tree algorithms are based on a hierarchical grouping of particles in a tree-like data structure. For each group, one performs an expansion of the potential into multipole moments. These moments are then used in approximations of the force due to distant groups. The great thing about short-range potentials is that the second step isn't even necessary as the effects of distant groups can simply be discarded. In this case, the tree-like dissection of particles into groups is only used to avoid having to calculate distances between all particles of the system. Far away sections of the tree can be ignored right from the outset.

(b) Suppose you like to sum over a long list of 10^7 floating point numbers containing all natural numbers from 1 to 10^7 in random order as a double precision (64 bit, 53 bit mantissa) floating point number. Which of the following methods will lead to the mathematically correct result?

(1) Summing up all numbers from the first to the last entry of the list using a single precision (32 bit, 23 bit mantissa) summation variable.

(2) Sorting the numbers and then summing them up starting from the largest using a single precision summation variable.

(3) Sorting the numbers and then summing them up starting from the largest using a quad double (128 bit, 112 bit mantissa) precision summation variable.

(4) Mapping all numbers in the list to integer numbers (32 bit) and summing those up using a long long (64 bit) integer summation variable.

(b) Summing floating point numbers

(1) False. A single precision float is not precise enough to store all integers in the interval $[1, 10^7]$ exactly, let alone sum them up without loss of precision.

(2) False. The result will be different but still wrong. The problem explained in part (1) still exists, but rounding errors during summation occur differently now due to the changed order.

(3) Correct. The type `long double` has sufficient precision to store all integers in $[1, 10^7]$ as well as their sum exactly. It will yield the correct result of

$$\frac{n}{2}(n+1) = 50\,000\,005\,000\,000, \qquad \text{where } n = 10^7. \tag{1}$$

(4) Correct. Since the type `double` is also capable of storing all integers in $[1, 10^7]$ exactly, the type conversion from `double` to `int` will be performed without error. Also, the `long long` is large enough to store the correct result. `long long` overflow does not occur until

$$2^{64-1} - 1 = 9\,223\,372\,036\,854\,775\,807 \gg 50\,000\,005\,000\,000. \tag{2}$$

(c) Which of the statements about gravity solvers are true?

(1) Direct summation algorithms have a higher force accuracy than tree algorithms.

(2) The resolution of a particle-mesh algorithm only depends on the gravitational softening length of the particles.

(3) Gauss-Seidel algorithms suffer from slow convergence of large scale modes.

(4) Multigrid V-cycles can be used to speed up a tree algorithm.

(b) Gravity solvers

(1) Correct. Tree algorithms approximate forces acting within a system by bunching groups of far-away particles together and expanding their potential into a truncated series of multipoles.

(2) False. While it is true that the softening length introduces a smallest resolved length-scale into the simulation, the accuracy of the particle-mesh algorithm depends also on the mesh size. That is because once all particles are mapped onto a discrete mass or charge density on the mesh, the system is mapped into Fourier space and the Poisson equation is solved there. The resolution in Fourier space is directly proportional to the grid's size.

(3) Correct. That is because in every sweeping of the grid only neighbouring points communicate. In other words, during each iteration, information travels only by one cell in any direction. High-frequency, short-wavelength errors spanning only a couple of cells are therefore smoothed out rapidly, whereas low-frequency, high-wavelength errors that may span large parts of the mesh disappear only gradually.

(4) False. Multigrid V-cycles can be used to speed up iterative solvers of the Poisson equation. They have nothing to do with tree algorithms.

2 Sampling with nonuniform probability distribution functions

Suppose you have a large set of random numbers x drawn from the interval $[0, 1)$ with a uniform probability distribution.

(a) Calculate the transformation $y(x)$ such that you get random numbers y with the probability density function

$$p(y) = \frac{1}{3}y^2 \qquad \text{for } y \in [0, 3). \tag{3}$$

(b) How many more random numbers with uniform probability distribution would you have to generate to obtain equally many random numbers according to $p(y)$ using instead the rejection method with a constant envelope function?

(a) This can be achieved with the exact inversion method. Due to conservation of probability, we have

$$q(x)\,\mathrm{d}x \stackrel{!}{=} p(y)\,\mathrm{d}y. \tag{4}$$

where q is the uniform distribution we know how to produce, i.e.

$$q(x) = \begin{cases} 1 & \text{for } x \in [0, 1], \\ 0 & \text{otherwise,} \end{cases} \tag{5}$$

and p is the distribution we would like to achieve. To find the mapping the $y(x)$ that tells us how to compute the sample y in p that corresponds to x in the uniform distribution, we equate the cumulative distribution functions (CDF) of p and q

$$Q(x) = \int_{-\infty}^{x} q(x')\,\mathrm{d}x' \stackrel{!}{=} \int_{-\infty}^{y} p(y')\,\mathrm{d}y' = P(y). \tag{6}$$

Inserting eq. (5) for q and eq. (3) for p, we get

$$x = \int_{-\infty}^{y} p(y')\,\mathrm{d}y' = \int_{0}^{y} \frac{1}{3}(y')^2\,\mathrm{d}y' = \frac{1}{9}y^3. \tag{7}$$

Solving this equation for y, we arrive at

$$y(x) = \sqrt[3]{9x}. \tag{8}$$

(b) The cheapest constant envelope function would be

$$f(y) = \sup\big\{p(y) \,\big|\, y \in [0,3)\big\} = 3 \qquad \forall y \in [0,3). \tag{9}$$

The integral I_f of $f(y)$ across our domain $[0,3)$ is three times as large as that of $p(y)$,

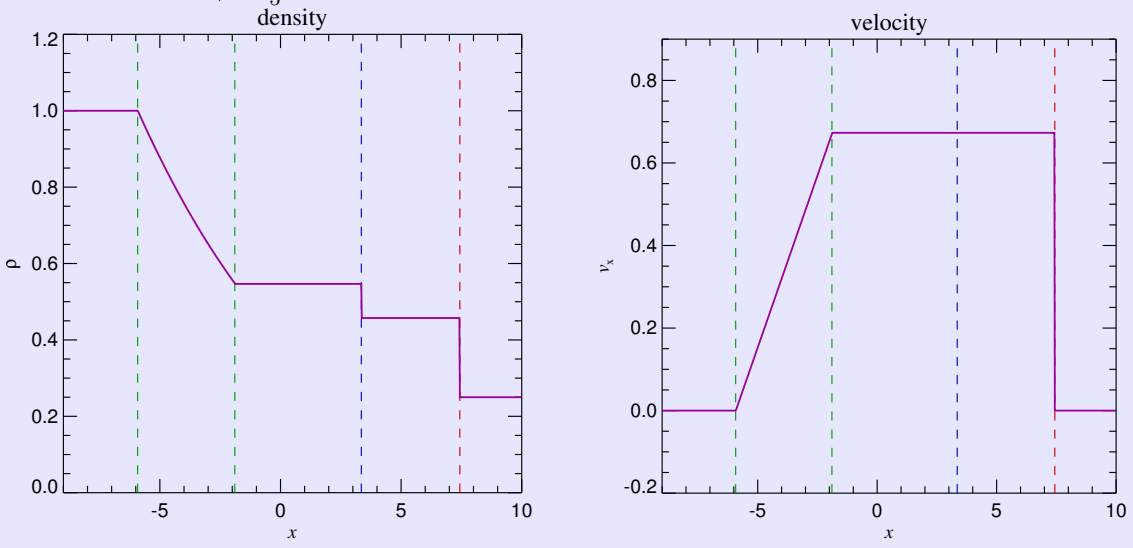
$$I_f = \int_0^3 f(y)\,\mathrm{d}y = 9, \tag{10}$$

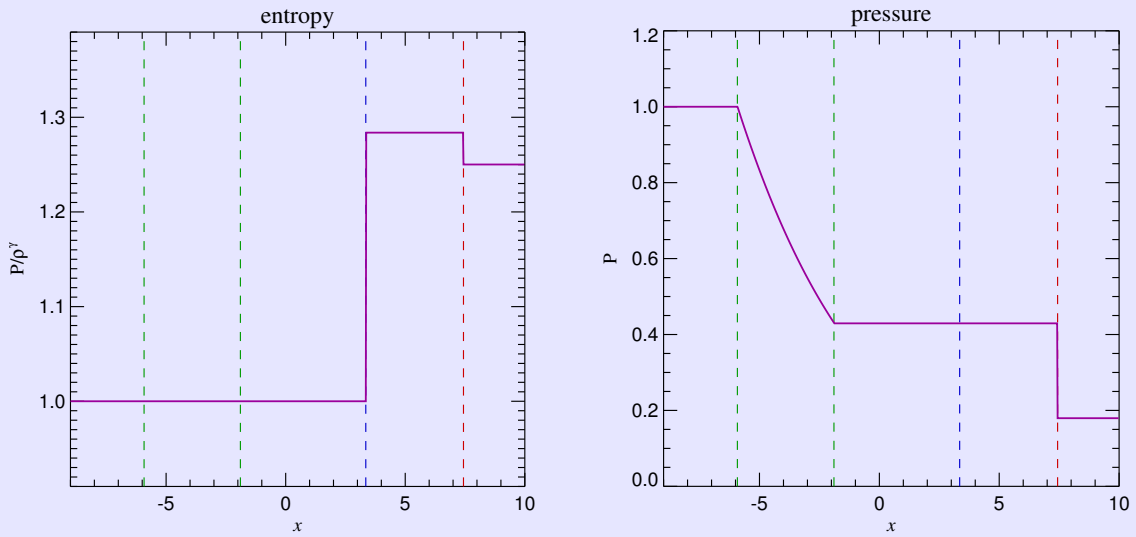
$$I_p = \int_0^3 p(y)\,\mathrm{d}y = \frac{1}{9}\,3^3 = 3. \tag{11}$$

This means that we will have to generate **six times as many numbers** if we use the constant envelope function together with a uniform distribution since two thirds of our randomly generated samples will not lie between $p(y)$ and the y-axis, and we need to generate two random numbers for every sample, one for the y-coordinate, i.e. the potential sample itself, and one for the vertical, say z-coordinate, to check if it is smaller than $p(y)$.

3 Fluid discontinuities

Consider the following evolved state of a Sod shock-tube problem for an ideal gas with adiabatic index $\gamma = \frac{7}{5}$:





The flow is shown at time $t = 5.0$ and has developed a characteristic wave structure. At the initial time $t = 0$, there were two piece-wise constant states separated by an interface at $x = 0$.

(a) Determine the initial state (ρ, P, v) on the left and right side of this Sod shock tube.

(b) Identify the location of shocks, contact discontinuities and rarefaction waves and mark them in the panels.

(c) What is the Mach number of the shock present in this flow?

(a) Looking at the left and right edges of the diagrams for the density, pressure and velocity, we read off the initial state of the system as characterized by the values

$$\begin{pmatrix} \rho_L \\ P_L \\ v_L \end{pmatrix} = \begin{pmatrix} 1.0 \\ 1.0 \\ 0.0 \end{pmatrix}, \qquad \begin{pmatrix} \rho_R \\ P_R \\ v_R \end{pmatrix} = \begin{pmatrix} 0.25 \\ 0.18 \\ 0.0 \end{pmatrix}, \qquad \text{at } t = 0. \tag{12}$$

(b) The positions of both the shock and contact waves as well as the edges of the rarefaction wave are already marked in the diagrams. The rarefaction wave is enclosed in green dashed lines, contact of the two states occurs at the blue dashed line, and the shock's propagation is indicated by the red dashed line.

(c) A shock's Mach number is given by $\mathcal{M} = |v_s/c_s|$, where v_s is the velocity with which the gas (or fluid) flows into the shock, and c_s is the pre-shock speed of sound. For a shock to be present requires $v_s > c_s$, i.e. a Mach number is always greater than one.

In an ideal gas, the pre-shock speed of sound can be calculated from the pre-shock pressure and density, here P_R and ρ_R, respectively, via

$$c_s = \sqrt{\frac{\gamma P_R}{\rho_R}} = \sqrt{\frac{1.4 \cdot 0.18}{0.25}} \approx 1. \tag{13}$$

Looking at any of the four diagrams, we can see that the shock has traveled a distance of approximately $\Delta x = 7.4$ during the simulation time $\Delta t = 5.0$. The shock velocity is therefore $v_s = \frac{\Delta x}{\Delta t} \approx 1.48$. This yields a Mach number of

$$\mathcal{M} = \left| \frac{v_s}{c_s} \right| \approx 1.495. \tag{14}$$

4 Integration with an adaptive stepsize

You want to test an integration algorithm with adaptive stepsize by integrating the equation

$$\frac{\mathrm{d}}{\mathrm{d}t}x = f(t) = t^2, \tag{15}$$

with the initial value $x(t = 0) = 1$ up to $t = 1$. Your algorithm should have a maximum relative error of 10^{-5}, estimated by evaluating the difference of the integration with two consecutive half timesteps and with the full timestep. You are using the following code:

```
double f(double t)
{
    return t*t;
}

double euler(double x, double t, double dt)
{
    return x + dt * f(t);
}

int main()
{
    double t = 0.0, x = 1.0;
    double dt = 0.5, tmax = 1.0;
    double max_error = 1.0e-5;
    double min_error = 0.1*max_error;
    double abserr, xlarge, xsmall;
    while(t < tmax)
    {
        xlarge = euler(x,t,dt);
        xsmall = euler(x,t,0.5*dt);
        xsmall = euler(xsmall,t+0.5*dt,0.5*dt);
        t += dt;
        x = xsmall;
        abserr = fabs(xlarge-xsmall);
        if(abserr/fabs(xsmall) < min_error) dt *= 2.0;
        if(abserr/fabs(xsmall) > max_error) dt *= 0.5;
        printf("t = %.3g, x = %.3g timestep = %.3g, error = %04.3g \n
            ", t, x, dt, fabs((xlarge-xsmall)/xsmall) );
    }
}
```

This is the output of your algorithm:

```
t = 0.5, x = 1.02 timestep = 0.25, error = 0.0154
t = 0.75, x = 1.1 timestep = 0.125, error = 0.016
t = 0.875, x = 1.17 timestep = 0.0625, error = 0.00521
t = 0.938, x = 1.22 timestep = 0.0312, error = 0.00142
t = 0.969, x = 1.25 timestep = 0.0156, error = 0.000369
t = 0.984, x = 1.26 timestep = 0.00781, error = 9.39e-05
t = 0.992, x = 1.27 timestep = 0.00391, error = 2.37e-05
t = 0.996, x = 1.28 timestep = 0.00391, error = 5.94e-06
t = 1, x = 1.28 timestep = 0.00391, error = 5.94e-06
```

(a) Compute the analytical result at $t = 1$.

(b) Interpret the above output and explain why the errors are unexpectedly large.

(c) How can you change this? You can either modify the code or explain in detail, which lines to edit and how.

(a) Integrating the differential equation eq. (15), we get

$$x(t) = \int t^2 \, \mathrm{d}t = \frac{1}{3} t^3 + c. \tag{16}$$

The initial conidition $x(t = 0) = 1$ sets c to 1. Thus the analytical result at $t = 1$ is

$$x(t = 1) = \frac{1}{3} 1^3 + 1 = \frac{4}{3}. \tag{17}$$

(b) The errors are not unexpectedly large. The Euler method is a very simple but inaccurate integrator and the initial stepsize was chosen much too large.

In its current implementation, the code proceeds with a time step even if the relative error is larger than 10^{-5}. To ensure smaller errors right from the outset, we would have to discard steps with a relative error larger than our desired upper bound.

(c) The code can be modified as follows to discard trial steps that return an oversized error:

```
int main()
{
    double t = 0.0, x = 1.0, dt = 0.5, tmax = 1.0;
    double max_error = 1.0e-5, min_error = 0.1*max_error;
    double abserr, xlarge, xsmall;

    while(t < tmax)
    {
        xlarge = euler(x,t,dt);
        xsmall = euler(x,t,0.5*dt);
        xsmall = euler(xsmall,t+0.5*dt,0.5*dt);
        abserr = fabs((xlarge - xsmall)/xsmall);

        if(abserr < min_error) dt *= 2.0;
        else if(abserr > max_error) dt *= 0.5;
        else {
            t += dt; x = xsmall;
            printf("t = %-10.3gx = %-10.3gdt = %-10.3gerr = %-10.3g
                \n", t, x, dt, abserr);
        }
    }
}
```

The final output lines of this implementation reads:

```
  ⋮
t = 0.992      x = 1.32       dt = 0.00391    err = 5.7e-06
t = 0.996      x = 1.33       dt = 0.00391    err = 5.7e-06
t = 1          x = 1.33       dt = 0.00391    err = 5.71e-06
```