Tutorial

1: Equations of motion and integrators

Alexander Schlaich*, Stefan Kesselheim†, Florian Rühle‡

November 2, 2011
ICP, Uni Stuttgart

Contents

1 A simple first order integrator: Cannonball

1.1 Introduction

The simplest possible set of equations of motion is one particle with a constant external force. To get some first impressions on C programming, we will solve this one-particle problem by implementing the easiest possible straight forward integrator.

The idea is simply to formulate the Newton equations for finite time steps:

$$v(t + \Delta t) = v(t) + \left(\frac{F(t)}{m}\right) \Delta t \tag{1}$$

$$x(t + \Delta t) = x(t) + v(t) \cdot \Delta t \tag{2}$$

*schlaich@icp.uni-stuttgart.de
†kessel@icp.uni-stuttgart.de
‡floh@icp.uni-stuttgart.de

Of course, we still need x_0 and v_0 as initial conditions. As you can easily see, these two equations are nothing but a second order Taylor expansion around the time coordinate:

$$v(t + \Delta t) = v(t) + \frac{\partial v(t)}{\partial t}\Delta t + \mathcal{O}(\Delta t^2) \tag{3}$$

$$x(t + \Delta t) = x(t) + \frac{\partial x(t)}{\partial t}\Delta t + \mathcal{O}(\Delta t^2) \tag{4}$$

Thus the order of numerical steps for our integrator is given: We first have to calculate the force on the particles for their current positions. From this, we can get the new positions and the new velocities. This method will allow us to implement an easy code.
Note: Is this algorithm time-reversible? (...if you do a Taylor expansion to $(t - \Delta t)$...?)

1.2 Implementing the simple integrator

Task: (0 points)
Implement our simple integrator in the function `integrate()`

In the tutorial's directory, there is the subdirectory "integrator", which contains the prepared code. First, you only need to take a look at the `integrate` function in the `integrate.c` file. It already contains all needed variables to implement your integrator. The function is prepared for gravitational and wind force. Implement the gravitational force first (the particle's mass is 1). Compile the program with

```
./compile_integrator
```

If you have trouble, ask your tutors for help. Run the compiled code for the first time with

```
./integrator
```

You will now see a file called `trajectory.dat` that contains the trajectory of a thrown particle. To view this trajectory, we will need the very useful plotting tool `gnuplot`. In your terminal, type in

```
gnuplot
```

Plot the trajectory with the gnuplot command

```
plot "trajectory.dat"
```

The result should look like figure 1. If it doesn't, take another look at your integrator. If necessary, get help from your tutors.

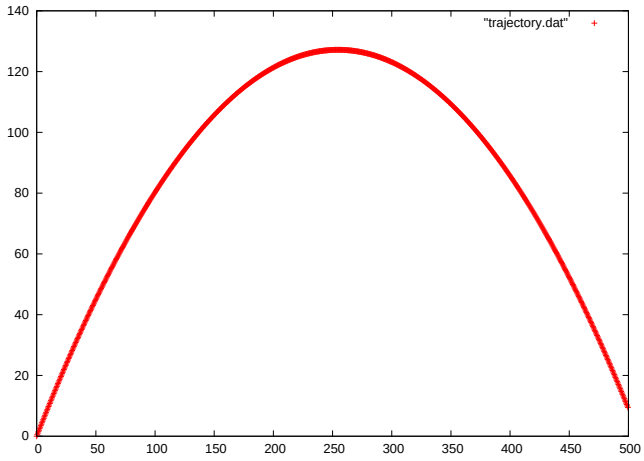

Figure 1: The trajectory as printed with `gnuplot` .

If you change something in your code, be sure to compile it again! This sounds clear but it is often forgotten.

1.3 Study the trajectory

Influence of wind

With the programm, you have the option to set starting values for your parameters from the console. Take a look at the file `integrator.c`, which contains the information about the possible parameters.

Task: (1 points)
Add constant wind force to the integrator. The implementation for wind force is analogous to the gravitational force, just in another direction.
Run the program several times with the same starting velocity of $(50, 50)$ *and vary the wind force. Does the wind force change the trajectory? Are the changes in a realistic range? If not, take a closer look at your integrator!*

Note for presenting the homework: Whenever you are asked to produce/compare some results you need to include in your report a plot showing the result and some discussion of it.

Influence of starting parameters

Task: (1 points)
Vary the starting speed of your particle. At an angle of 45 degrees ($v_x = v_y$), try to hit the legend produced by gnuplot in the top right corner of your plot. Write down the used starting speed and include the plot in the report.

The graphical user interface – play the game

As you can tell from the name of this section, the integrator you have just programmed can be used for a very old computer game called "Cannonball". After all, the trajectory you just plotted may show the flight of a cannonball! :o) We have prepared a graphical user interface that can be used with your own integrator.

Go to the directory "cannonball" and copy the integration algorithm you just programmed:

```
cp ../integrate.* .
```

Now you can compile the code with the command `./compile-it.sh` Start the game typing `./cannonball` If everything worked fine, you can now play the game with random wind and adjustable angle and initial velocitiy. And all with your integrator.

2 Advanced integrators: Solar system

2.1 Introduction

One of the basic concepts of MD simulations is the integration of classical equations of motion. Therefore a suitable algorithm is required. Since the equations of motion are **time-reversible**, we have to use an appropiate integration scheme. A simple approach to do this is the so called Verlet algorithm:

$$x(t + \Delta t) = 2\,x(t) - x(t - \Delta t) + a(t) \cdot \Delta t^2 + \mathcal{O}\left(\Delta t^4\right) \tag{5}$$

Positions, velocities and the acceleration at time *t* are denoted by *x(t)*, *v(t)*, and *a(t)*, respectively.

However this algorithm comes with a difficulty during the first time step (think about it!). To overcome this problem, a wide range of methods exists. A widely used approach is named Velocity-Verlet algorithm, that includes the contributions of the velocities directly in every integration step:

$$x(t + \Delta t) = x(t) + v(t) \cdot \Delta t + \frac{a(t)}{2}\Delta t^2 + \mathcal{O}\left(\Delta t^3\right) \tag{6}$$

$$v(t + \Delta t) = v(t) + \frac{a(t + \Delta t) + a(t)}{2}\Delta t + \mathcal{O}\left(\Delta t^3\right). \tag{7}$$

Of course, to implement the algorithm like this, we would have to access the old and new forces in one timestep. To make the code less complicated, the following scheme is mostly used:

1. Update positions by simple Taylor expansion:

$$x(t + \Delta t) = x(t) + v(t) \cdot \Delta t + \frac{a(t)}{2}\Delta t^2$$

2. Save half time step temporary in velocity variable:

$$v(t + \frac{\Delta t}{2}) = v(t) + a(t)\frac{\Delta t}{2}$$

3. Update accelerations $a(t + \Delta t)$ (call forces routine)

4. Update velocities to full time step:

$$v(t + \Delta t) = v(t + \frac{\Delta t}{2}) + a(t + \Delta t)\frac{\Delta t}{2}$$

This simply means shifting the algorithm's order in a way that we don't need two values for the force but a temporary velocity instead. As we have seen, the error for our Verlet update is $\mathcal{O}(\Delta t^4)$. What is the error order of our first calculation step here?

Another way to deal with this problem is the Leapfrog Verlet algorithm, whose name comes from the leapfrog-looking integration scheme as seen in figure 2.

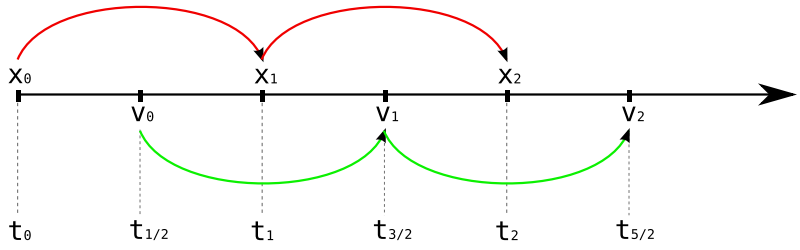

Figure 2: Illustration of the Leapfrog integration method

The basic idea is to give up the synchronous modification of position and velocity data. We first calculate the velocity at $t + \frac{1}{2}\Delta t$

$$v\left(t + \frac{1}{2}\Delta t\right) = v(t) + \left(\frac{F(t)}{m}\right)\frac{\Delta t}{2} + \mathcal{O}(\Delta t^2)$$

and then we update the velocities asynchronously, but by whole time steps:

$$x(t + \Delta t) = x(t) + v\left(t + \frac{1}{2}\Delta t\right)\Delta t + \mathcal{O}(\Delta t^2) \tag{8}$$

$$v\left(t + \frac{3}{2}\Delta t\right) = v\left(t + \frac{1}{2}\Delta t\right) + \left(\frac{F\left(t + \Delta t\right)}{m}\right)\Delta t + \mathcal{O}(\Delta t^2) \tag{9}$$

This algorithm has the advantage of being totally symmetric (time-reversibility!) and also avoids the initialization problem.

As you can see, we now have first, third and second order integration algorithms. Each has its advantages when it comes to speed, numerical precision and error calculation. We will examine this later.

Task: (2 points)
Derive the Velocity Verlet algorithm. Do it in both ways listed below.

1. *Using a Taylor expansion and show the order of precision of the velocity Verlet integrator.*
   Recall, how the Verlet algorithm was derived in the lecture of prof. Holm. To derive the update algorithm for positions, use a Taylor expansion of $x(t + \Delta t)$ truncated after third order. To derive the velocity update, Taylor-expand $v(t + \Delta t)$ up to the third order. To obtain an expression for $\partial^2 v(t)/\partial t^2$, use a Taylor expansion for $\partial v(t + \Delta t)/\partial t$ truncated after the second order.

2. *Rearranging the equations of the Verlet algorithm. Express $x(t + \Delta t)$ using Equation 6 and rearrange Equation 6 to express $x(t)$. Add the two equations and then group velocity terms together. Put all velocity terms on one side of Equation 7 and use them to substitute in your last equation.*

Which differences in the results would you expect when using our non time-reversible simple integrator and a time reversible integration scheme like the Verlet algorithm? Think about the influence on energy and momentum of the system!

Task: (2 points)
Implement the Velocity Verlet and Leapfrog Integrator

- *Implement the Velocity Verlet algorithm in the function* `integrate\_verlet` *in the code file* `smd.c`. *All required variables are already declared, just fill in the correct lines. If unsure how to use the variables, you can take a quick look at the function* `integrate`, *which is nothing but the first simple integrator you just implemented.*

- *Implement the Leapfrog algorithm in the function* `integrate\_leapfrog` *in the code file* `smd.c`. *All required variables are already declared, just fill in the correct lines.*

Optional task: (2 points)
Implement an advanced integrator

- *If you are already an advanced programmer, you can implement an integration algorithm of even higher order (≥ 4) and later compare it with the other integration schemes.*

2.2 Compilation and visualization

The prepared code you need for this exercise is placed in the directory `simple\_md`. The file you want to edit is `smd.c`. The `Makefile` in the directory lets you compile the code by simply typing `make`.

When you compile and run the code (`smd`), it produces two files `traj.pdb` and `energy.dat`. The second one contains the total energy of your system and can be plotted with `gnuplot`. The first one is a PDB-file (particle database) and can be viewed with the tool VMD (Visual Molecular Dynamics). An interesting feature will certainly be the possibility to change the appearance of your particles via menu "Graphics" →"Representations". Try e.g. a Van-Der-Waals Drawing method (VDW) with a smaller sphere scale, and set the coloring method to "SegName", which has proven to look nicely. :o) Attention: If your particles spread too far because you are using a less accurate integrator over a very long simulation time, you won't see your very small particles anymore because VMD zooms out automatically!

You need to take a look at the loop within the `main` function, where you can decide which function to call for your integration. When compiling and running the code for several different integrators, be sure to move the resulting files to suitable names in between the runs, before you overwrite them! In this function, you can also choose values for the main variables to influence the simulation later.

2.3 Properties of the integrators

You now have a simple code with which you can apply integrators to your problem, each having its advantages and disadvantages! Your homework is to find out the advantages in speed, reliability and long-term stability of each integrator. You will want to adapt the values for the variables `timestep`, `duration` and `update\_interval` in your function `main` later.

Long-term stability

You can choose which integrator to use by, in the `main` routine, calling one of the functions `integrate`, `integrate\_leapfrog` or `integrate\_verlet`. Open the written pdb-files with VMD. You can see some parts of our solar system: The sun, Venus, Earth, Mars and Jupiter. We also implemented the earth's moon as the first thing to fall apart with a less exact integrator.

Task: (1 points)

- *For our initial value of* 0.012 *for the timestep, after how many earth years does the moon leave the earth's orbit for the three different integrators?*
- *What happens when you decrease or increase the timestep of your integration?*

Energy conservation

To find out more about your system and the time evolution, it is usually helpful to take a look at some other observables, not just the particle trajectories. As you know, our program already calculates the system's total energy (potential and kinetic) and prints it into a file.

When you have produced energy files for all three integrators, take a look at them with gnuplot. You can squeeze several data files into one plot by seperating them with commas:

```
plot ``data1'', ``data2'', ``data3''
```

Take a look at the energy files of the simple and the leapfrog integrator and compare each in one plot together with the Velocity Verlet integrator.

Task: (2 points)

- *Why do the plots look like this? Explain! Hand in your explanation and the graphs. Think about conservation laws. To check what happens in the interesting moments, you can use the visualization program VMD. To visualize you simulation using VMD, use the command* `vmd -f traj.pdb`
- *How does each integrator perform when it comes to speed? You can measure the runtime on your computer by using the command* `time ./smd`.
  Hint: *Writing to the hard drive takes a lot of time. So be sure to make the "update interval" as long as your whole simulation time. If your simulation runs too fast and there is hardly any difference between the integrators, what are the two main things you can do to increase the runtime?*
- *How much smaller do you have to make your timestep to reach the stability of the Velocity Verlet algorithm with the Leap Frog integrator?*
- *Which integrator would you use for this particular system when you want to simulate the solar system for a million years? For ten years without the moon? Include the reasons for your decision.*