http://demonstrations.wolfram.com/TwoStepAndFourStepAdamsPredictorCorrectorMethod/HTMLImages/index.en/11.gif(predictor step: two-step Adams–Bashforth)

http://demonstrations.wolfram.com/TwoStepAndFourStepAdamsPredictorCorrectorMethod/HTMLImages/index.en/12.gif(corrector step: two-step Adams–Moulton)

The four-step Adams predictor-corrector method uses the four-step Adams–Bashforth and Adams-Moulton methods together:

http://demonstrations.wolfram.com/TwoStepAndFourStepAdamsPredictorCorrectorMethod/HTMLImages/index.en/13.gif(predictor step)   
http://demonstrations.wolfram.com/TwoStepAndFourStepAdamsPredictorCorrectorMethod/HTMLImages/index.en/14.gif(corrector step)

Several years ago I thought of writing a solar system simulator.

Newton’s law of gravity is simple to integrate, so this should be a simple program.

Just set an initial position for all the planets, then run the integrate loop.

The main issue I encountered was that I could not find the initial positions of the planets :)

I thought a little about this problem, and realized that theoretically it should be possible to recover the 3d position from the aparent positions of the planets on the earth sky (which I could find in some astronomic calendars). But this was not an easy task. Basically to integrate the solar system I needed the precise location of the planets in some coordinate reference system, and their speed. To get this from the 2D projection on the sky should be possible: make some estimation for the 3d position and speed, then apply corrections to get the same 2d timeseries from the astronomical calendar. An interesting problem, but not quite the one I wanted to solve.

So in the end I did nothing more, and the idea was shelved.

Recently I googled for a solar system simulator, and after a couple of false starts I found <http://www.moshier.net/ssystem.html> which already does what I intended. From my point of view, the gem was that it contains a header file with exactly the information I was missing (the 3d positions and speed for all planets at a specific moment in time). As it turns out, there were a lot more things that I was missing, but more on this later.

From there I recovered the following story.

JPL needed to simulate the solar system for their missions and to provide that data to other projects.

So they made a detailed simulation (I was barely born around that time), and then published the resulted efemerides tables for all planets for a couple of kyears.

The Moshier guy wanted to reproduce the same data, 20 years later.

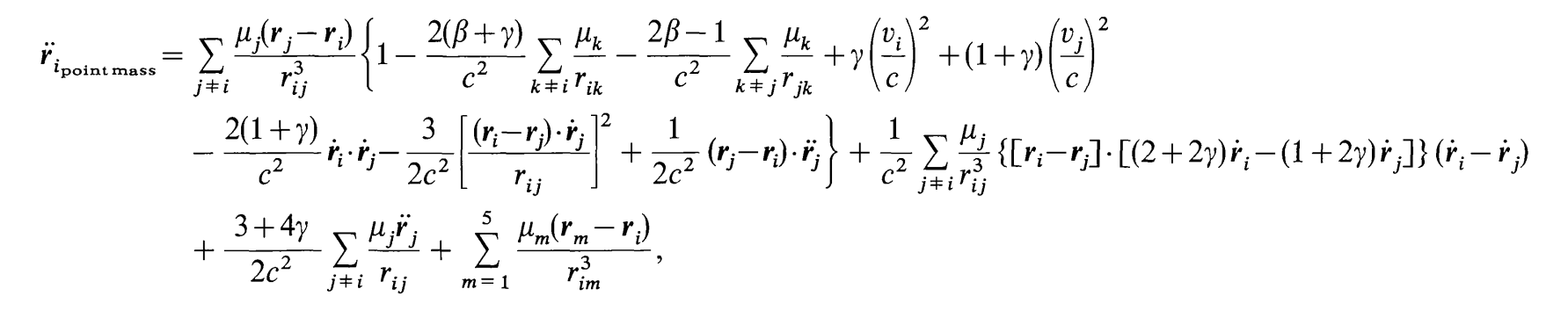
So I took a look at his code, which contained some stuff I did not expect.

There were 2 things that stand out:

1. The newton law of gravitation is not enough for a realistic simulation; relativistic corrections need to be applied.
2. The integrator used needs to be quite precise. A basic Runge-Kutta is not enough.

# 2-Body gravitation law

The equation used by the Jpl simulation has the following form:



I tried to find out what it means, but so far the results have been inconclusive.

It seems to be the PPN (post newton) approximation, with beta and gamma set to 1 (for general relativity).

# Integration

We have the equation of motion, and we know the initial state. So the obvious next step is to use an integration method to calculate the time evolution of the system.

The jpl simulation uses a predictor-corrector integrator or order up to 19 (Adams-Bashfort-Moulton) with separate adaptive time steps. I think that the separate time steps is just a minor optimization in the solar system case because there are only 10 bodies, and because it influences only one part of the total computation. Best case scenario it probably provide a 2-4 times speed up. So I think I will skip the separate time steps. (But it is a major speed up for the N-Body cases where N is big).

I found a paper about an easy way to generate the coefficients for A-B-M (solving an linear eq system), but for normal uses I can just take precomputed coefficients.

Still not sure what is the advantage of a 4 order predictor-corrector (8 terms total) versus RK of order 8.

The plot thickens when reading about N-body simulations I found that the modern implementations use a Hermite implementation, which apparently is faster and easier to implement. To check the stability of the integration the total energy of the system is used (i.e it must remain constant).

I found some details about the 4th order Hermite, and about 6 and 8th order which should be more precise.

Because some formulations of the Hermite interpolation are time symetric, it is possible to apply the corrector step several time to obtain better results. This is called the P(EC)n method.

In retrospect, I do not understand how this sophisticated integration methods work. I am currently not able to deduce them on my own.

I found a Hermite implementation on Github: <https://github.com/nitadori/Hermite> . This contains even a GPU implementation and AVX optimizations. The guy has published some papers about GPU NBody simulations.

I finally got lucky with my internet searches, and found <http://www.artcompsci.org/kali/vol/two_body_problem_2/ch09.html>

In a dialog style it tries to develop software for NBody simulations.

It uses ruby for code examples.

In 2.4 it describes a bug, quite ugly, caused by object programming and the ruby function call convention; basically function calls look the same as variable names; in the bug a function method was also accessing class members (not visible in any way in the place of the call – for all practical reasons it’s like using globals in a C program). Also must take care about side effects! Now when I read the code samples, I need to remember that mentions of “acc” are actually functions calls to acc() that are using the latest updates to “pos”.

Like Casey said, it is good to take a look into the disassembly of the code. Things I learned doing this: by default I was using fp:precise, so for sqrt it was calling a sqrt function. The operators for v3 are not inlined. The /2 is not replaced by the compiler with multiplication. Also this happens for v3... I was generating 32 bit code so later I switched to 64 bit. Tried to inline the vector operators, but it does not work.. need to add some high resolution timers to measure what happens.

The leapfrog method takes 160 ticks. These happen when calling the Vector3 operators instead of inline and optimize them.

I rewrote the leapfrog method manually by unrolling the Vector3 operators to simple double operations. The code is bigger, but also it takes only 90 ticks. The generated SSE code looks dense enough.

Implemented hermite4 as described in the book.

Hermite Initial: 417 ticks

Hermite Optimized: 180 ticks

Realized that I was not enabling avx compilation; after enable:

Hermite Initial: 350 ticks

Hermite Optimized: 170 ticks

So far I tested the 2-body integrators from the Kali book.

Next step is to implement the same with NBody.

DE 102: a numerically integrated ephemeris of the Moon and planets spanning forty-four centuries - 1982 – Newhall JPL

<http://www.artcompsci.org/kali/vol/two_body_problem_2/ch10.html> - hermite interpolation explained