The Art of Computational Science

The Kali Code

vol. 2

The 2-Body Problem:

Higher-Order Integrators

Piet Hut and Jun Makino

September 14, 2007

Contents

Preface

In the current volume, we continue the dialogue that we started in the previous volume, where Alice and Bob developed a gravitational 2-body code, based on the simplest of all integration algorithms, the forward Euler method. They are now ready to implement a variety of other algorithms, such as the leapfrog and Runge-Kutta schemes, as well as multi-step methods. They will even succeed in implementing two different eigth-order schemes.

0.1 Acknowledgments

Besides thanking our home institutes, the Institute for Advanced Study in Princeton and the University of Tokyo, we want to convey our special gratitude to the Yukawa Institute of Theoretical Physics in Kyoto, where we have produced a substantial part of our ACS material, including its basic infrastructure, during extended visits made possible by the kind invitations to both of us by Professor Masao Ninomiya. In addition, we thank the Observatory of Strasbourg, where the kind invitation of Professor Christian Boily allowed us to make a rapid beginning with the current volume.

It is our pleasure to thank Steve Giess, Douglas Heggie, Stephan Kolassa, Ernest Mamikonyan and Michele Trenti for their comments on the manuscript.

Piet Hut and Jun Makino

Kyoto, July 2004

Chapter 1

Leapfrog

1.1 The Basic Equations

Alice: Hi, Bob! Any luck in getting a second order integrator to work?

Bob: No problem, it was easy! Actually, I got two different ones to work, and a fourth order integrator as well.

Alice: Wow, that was more than I expected!

Bob: Let's start with the second order leapfrog integrator.

Alice: Wait, I know what a leapfrog is, but we'd better make some notes about how to present the idea to your students. How can we do that quickly?

Bob: Let me give it a try. The name *leapfrog* comes from one of the ways to write this algorithm, where positions and velocities 'leap over' each other. Positions are defined at times $t_i, t_{i+1}, t_{i+2}, \ldots$, spaced at constant intervals Δt, while the velocities are defined at times halfway in between, indicated by $t_{i-1/2}, t_{i+1/2}, t_{i+3/2}, \ldots$, where $t_{i+1} - t_{i+1/2} = t_{i+1/2} - t_i = \Delta t/2$. The leapfrog integration scheme then reads:

$$
\begin{aligned}
\mathbf{r}_i &= \mathbf{r}_{i-1} + \mathbf{v}_{i-1/2}\Delta t \\
\mathbf{v}_{i+1/2} &= \mathbf{v}_{i-1/2} + \mathbf{a}_i\Delta t
\end{aligned}
$$

Note that the accelerations $\mathbf{a}$ are defined only on integer times, just like the positions, while the velocities are defined only on half-integer times. This makes sense, given that $\mathbf{a}(\mathbf{r},\mathbf{v}) = \mathbf{a}(\mathbf{r})$: the acceleration on one particle depends only on its position with respect to all other particles, and not on its or their velocities. Only at the beginning of the integration do we have to set up the velocity at its first half-integer time step. Starting with initial conditions $\mathbf{r}_0$ and $\mathbf{v}_0$, we take the first term in the Taylor series expansion to compute the first leap value for $\mathbf{v}$:

$$\mathbf{v}_{i+1/2} = \mathbf{v}_i + \mathbf{a}_i \Delta t/2. \tag{1.1}$$

We are then ready to apply the first equation above to compute the new position $\mathbf{r}_1$, using the first leap value for $\mathbf{v}_{1/2}$. Next we compute the acceleration $\mathbf{a}_1$, which enables us to compute the second leap value, $\mathbf{v}_{3/2}$, using the second equation above, and then we just march on.

A second way to write the leapfrog looks quite different at first sight. Defining all quantities only at integer times, we can write:

$$\begin{aligned} \mathbf{r}_{i+1} &= \mathbf{r}_i + \mathbf{v}_i \Delta t + \mathbf{a}_i (\Delta t)^2/2 \\ \mathbf{v}_{i+1} &= \mathbf{v}_i + (\mathbf{a}_i + \mathbf{a}_{i+1}) \Delta t/2 \end{aligned} \tag{1.2}$$

This is still the same leapfrog scheme, although represented in a different way. Notice that the increment in $\mathbf{r}$ is given by the time step multiplied by $\mathbf{v}_i + \mathbf{a}_i \Delta t/2$, effectively equal to $\mathbf{v}_{i+1/2}$. Similarly, the increment in $\mathbf{v}$ is given by the time step multiplied by $(\mathbf{a}_i + \mathbf{a}_{i+1})/2$, effectively equal to the intermediate value $\mathbf{a}_{i+1/2}$. In conclusion, although both positions and velocities are defined at integer times, their increments are governed by quantities approximately defined at half-integer values of time.

1.2 Time reversibility

Alice: A great summary! I can see that you have taught numerical integration classes before. At this point an attentive student might be surprised by the difference between the two descriptions, which you claim to describe the same algorithm. They may doubt that they really address the same scheme.

Bob: How would you convince them?

Alice: An interesting way to see the equivalence of the two descriptions is to note the fact that the first two equations are explicitly time-reversible, while it is not at all obvious whether the last two equations are time-reversible. For the two systems to be equivalent, they'd better share this property. Let us inspect.

Starting with the first set of equations, even though it may be obvious, let us write out the time reversibility. We will take one step forward, taking a time step $+\Delta t$, to evolve $\{\mathbf{r}_i, \mathbf{v}_{i-1/2}\}$ to $\{\mathbf{r}_{i+1}, \mathbf{v}_{i+1/2}\}$, and then we will take one step backward, using the same scheme, taking a time step $-\Delta t$. Clearly, the time will return to the same value since $+\Delta t - \Delta t = 0$, but we have to inspect where the final positions and velocities $\{\mathbf{r}_f(t=i), \mathbf{v}_f(t=i-1/2)\}$ are indeed equal to their initial values $\{\mathbf{r}_i, \mathbf{v}_{i-1/2}\}$. Here is the calculation, resulting from applying the first set of equations twice:

$$\begin{aligned}
\mathbf{r}_f &= \mathbf{r}_{i+1} - \mathbf{v}_{i+1/2}\Delta t \\
&= \left[\mathbf{r}_i + \mathbf{v}_{i+1/2}\Delta t\right] - \mathbf{v}_{i+1/2}\Delta t \\
&= \mathbf{r}_i
\end{aligned}$$

$$\begin{aligned}
\mathbf{v}_f &= \mathbf{v}_{i+1/2} - \mathbf{a}_i\Delta t \\
&= \left[\mathbf{v}_{i-1/2} + \mathbf{a}_i\Delta t\right] - \mathbf{a}_i\Delta t \\
&= \mathbf{v}_{i-1/2}
\end{aligned}$$

In an almost trivial way, we can see clearly that time reversal causes both positions and velocities to return to their old values, not only in an approximate way, but exactly. In a computer application, this means that we can evolve forward a thousand time steps and then evolve backward for the same length of time. Although we will make integration errors (remember, leapfrog is only second-order, and thus not very precise), those errors will exactly cancel each other, apart from possible round-off effects, due to limited machine accuracy.

Now the real fun comes in, when we inspect the equal-time version, the second set of equations you presented:

$$\begin{aligned}
\mathbf{r}_f &= \mathbf{r}_{i+1} - \mathbf{v}_{i+1}\Delta t + \mathbf{a}_{i+1}(\Delta t)^2/2 \\
&= \left[\mathbf{r}_i + \mathbf{v}_i\Delta t + \mathbf{a}_i(\Delta t)^2/2\right] - \left[\mathbf{v}_i + (\mathbf{a}_i + \mathbf{a}_{i+1})\Delta t/2\right]\Delta t + \mathbf{a}_{i+1}(\Delta t)^2/2 \\
&= \mathbf{r}_i
\end{aligned}$$

$$\begin{aligned}
\mathbf{v}_f &= \mathbf{v}_{i+1} - (\mathbf{a}_{i+1} + \mathbf{a}_i)\Delta t/2 \\
&= \left[\mathbf{v}_i + (\mathbf{a}_i + \mathbf{a}_{i+1})\Delta t/2\right] - (\mathbf{a}_{i+1} + \mathbf{a}_i)\Delta t/2 \\
&= \mathbf{v}_i
\end{aligned}$$

In this case, too, we have exact time reversibility. Even though not immediately obvious from an inspection of your second set of equations, as soon as we write out the effects of stepping forward and backward, the cancellations become manifest.

Bob: A good exercise to give them. I'll add that to my notes.

1.3 A New Driver

Alice: Now show me your leapfrog, I'm curious.

Bob: I wrote two new drivers, each with its own extended `Body` class. Let me show you the simplest one first. Here is the leapfrog driver `integrator_driver1.rb`:

```
require "lbody.rb"

include Math

dt = 0.001           # time step
dt_dia = 10          # diagnostics printing interval
dt_out = 10          # output interval
dt_end = 10          # duration of the integration
##method = "forward"    # integration method
method = "leapfrog"   # integration method

STDERR.print "dt = ", dt, "\n",
      "dt_dia = ", dt_dia, "\n",
      "dt_out = ", dt_out, "\n",
      "dt_end = ", dt_end, "\n",
      "method = ", method, "\n"

b = Body.new
b.simple_read
b.evolve(method, dt, dt_dia, dt_out, dt_end)
```

Same as before, except that now you can choose your integrator. The method `evolve`, at the end, now has an extra parameter, namely the integrator.

Alice: And you can supply that parameter as a string, either "forward" for our old forward Euler method, or "leapfrog" for your new leapfrog integrator. That is very nice, that you can treat that choice on the same level as the other choices you have to make when integrating, such as time step size, and so on.

Bob: And it makes it really easy to change integration method: one moment you calculate with one method, the next moment with another. You don't even have to type in the name of the method: I have written it so that you can switch from leapfrog back to forward Euler with two key strokes: you uncomment the line

```
##method = "forward"    # integration method
```

and comment out the line

```
method = "leapfrog"   # integration method
```

Alice: It is easy to switch lines in the driver, but I'm really curious to see how you let Ruby actually make that switch in executing the code differently, replacing one integrator by another!

1.4 An Extended Body Class

Bob: Here is the new version of our `Body` class, in a file called `lbody.rb` with `l` for leapfrog. It is not much longer than the previous file `body.rb`, so let me show it again in full:

```
require "vector.rb"

class Body

  attr_accessor :mass, :pos, :vel

  def initialize(mass = 0, pos = Vector[0,0,0], vel = Vector[0,0,0])
    @mass, @pos, @vel = mass, pos, vel
  end

  def evolve(integration_method, dt, dt_dia, dt_out, dt_end)
    time = 0
    nsteps = 0
    e_init
    write_diagnostics(nsteps, time)

    t_dia = dt_dia - 0.5*dt
    t_out = dt_out - 0.5*dt
    t_end = dt_end - 0.5*dt

    while time < t_end
      send(integration_method,dt)
      time += dt
      nsteps += 1
      if time >= t_dia
        write_diagnostics(nsteps, time)
        t_dia += dt_dia
      end
      if time >= t_out
        simple_print
        t_out += dt_out
      end
    end
  end

  def acc
    r2 = @pos*@pos
    r3 = r2*sqrt(r2)
    @pos*(-@mass/r3)
```

```
  end

  def forward(dt)
    old_acc = acc
    @pos += @vel*dt
    @vel += old_acc*dt
  end

  def leapfrog(dt)
    @vel += acc*0.5*dt
    @pos += @vel*dt
    @vel += acc*0.5*dt
  end

  def ekin                        # kinetic energy
    0.5*(@vel*@vel)               # per unit of reduced mass
  end

  def epot                        # potential energy
    -@mass/sqrt(@pos*@pos)        # per unit of reduced mass
  end

  def e_init                      # initial total energy
    @e0 = ekin + epot             # per unit of reduced mass
  end

  def write_diagnostics(nsteps, time)
    etot = ekin + epot
    STDERR.print <<END
at time t = #{sprintf("%g", time)}, after #{nsteps} steps :
  E_kin = #{sprintf("%.3g", ekin)} ,\
 E_pot =  #{sprintf("%.3g", epot)} ,\
 E_tot = #{sprintf("%.3g", etot)}
             E_tot - E_init = #{sprintf("%.3g", etot-@e0)}
  (E_tot - E_init) / E_init =#{sprintf("%.3g", (etot - @e0) / @e0 )}
END
  end

  def to_s
    "  mass = " + @mass.to_s + "\n" +
    "   pos = " + @pos.join(", ") + "\n" +
    "   vel = " + @vel.join(", ") + "\n"
  end

  def pp               # pretty print
    print to_s
```

```
  end

  def simple_print
    printf("%24.16e\n", @mass)
    @pos.each{|x| printf("%24.16e", x)}; print "\n"
    @vel.each{|x| printf("%24.16e", x)}; print "\n"
  end

  def simple_read
    @mass = gets.to_f
    @pos = gets.split.map{|x| x.to_f}.to_v
    @vel = gets.split.map{|x| x.to_f}.to_v
  end

end
```

1.5 Minimal Changes

Alice: Before you explain to me the details, remember that I challenged you to write a new code while changing or adding at most a dozen lines? How did you fare?

Bob: I forgot all about that. It seemed so unrealistic at the time. But let us check. I first corrected this mistake about the mass factor that I had left out in the file `body.rb`. Then I wrote this new file `lbody.rb`. Let's do a diff:

```
|gravity> diff body.rb lbody.rb
11c11
<   def evolve(dt, dt_dia, dt_out, dt_end)
- -
>   def evolve(integration_method, dt, dt_dia, dt_out, dt_end)
22c22
<       evolve_step(dt)
- -
>       send(integration_method,dt)
36c36
<   def evolve_step(dt)
- -
>   def acc
39c39,49
<     acc = @pos*(-@mass/r3)
- -
>     @pos*(-@mass/r3)
>   end
```

```
>
>   def forward(dt)
>     old_acc = acc
>     @pos += @vel*dt
>     @vel += old_acc*dt
>   end
>
>   def leapfrog(dt)
>     @vel += acc*0.5*dt
41c51
<     @vel += acc*dt
- -
>     @vel += acc*0.5*dt
```

To wit: four lines from the old code have been left out, and twelve new lines appeared.

Alice: Only twelve! You did it, Bob, exactly one dozen, indeed.

Bob: I had not realized that the changes were so minimal. While I was playing around, I first added a whole lot more, but when I started to clean up the code, after it worked, I realized that most of my changes could be expressed in a more compact way.

Alice: Clearly Ruby *is* very compact.

Bob: Let me step through the code.

Chapter 2

Integrators on the Menu

2.1 The Leapfrog Method

Bob: It will be easy to show you what I have done, in order to implement the leapfrog. My first attempt was to code it up in the same way as we did for the forward Euler scheme. Remember how we did that? In order to take one forward Euler step, we used the following method:

```
def evolve_step(dt)
  r2 = @pos*@pos
  r3 = r2*sqrt(r2)
  acc = @pos*(-@mass/r3)
  @pos += @vel*dt
  @vel += acc*dt
end
```

Switching to the leapfrog, I wrote:

```
def evolve_step(dt)
  r2 = @pos*@pos
  r3 = r2*sqrt(r2)
  acc = @pos*(-@mass/r3)
  @vel += acc*0.5*dt
  @pos += @vel*dt
  r2 = @pos*@pos
  r3 = r2*sqrt(r2)
  acc = @pos*(-@mass/r3)
  @vel += acc*0.5*dt
end
```

Alice: I presume you used the second set of equations which you wrote before:

$$\begin{aligned}\mathbf{r}_{i+1} &= \mathbf{r}_i + \mathbf{v}_i\Delta t + \mathbf{a}_i(\Delta t)^2/2 \\ \mathbf{v}_{i+1} &= \mathbf{v}_i + (\mathbf{a}_i + \mathbf{a}_{i+1})\Delta t/2\end{aligned}$$

Bob: Yes, they are much more convenient; the first set defined the velocities at half-step times, but here everything is defined at the same time: positions, velocities, and accelerations.

Alice: Why did you not start by implementing the first equation first?

Bob: If I would step forward the position first, I would not be able to obtain the acceleration $\mathbf{a}_i$ that is needed for the second equation, since the acceleration at time i can only be obtained from the position at time i. Therefore, the first thing to do is to obtain $\mathbf{a}_i$. Only then can I step forward the position, in order to calculate $\mathbf{a}_{i+1}$. However, before doing that, I'd better use the old acceleration $\mathbf{a}_i$, since another acceleration call that would calculate $\mathbf{a}_{i+1}$ would overwrite $\mathbf{a}_i$.

2.2 Two Different Versions

Alice: Can you summarize that in a list of steps?

Bob: Here is my recipe:

- calculate $\mathbf{a}_i$
- use it to step the velocity forward by half a time step
- step the position forward
- this allows you to calculate $\mathbf{a}_{i+1}$
- use that to step the velocity forward by another half step.

So the 'leap' part of the frog shows itself in this half step business for the velocity.

Alice: Ah, I see now. You really had to add $\mathbf{v}_i\Delta t + \mathbf{a}_i(\Delta t)^2/2$ to the position in the third of your five steps. But since you had already updated the velocity from $\mathbf{v}_i$ to effectively $\mathbf{v}_{i+1/2} = \mathbf{v}_i + \mathbf{a}_i\Delta t/2$, you could abbreviate the position update. You just added $\mathbf{v}_{i+1/2}\Delta t$ to the position instead. Clever!

The only drawback of writing it in such a compact and smart way is that the correspondence between the equations is not immediately clear, when you look at the code.

Bob: You could call it FORMula TRANSlating with an optimizer.

Alice: You may want to write a version where you switch off your clever optimizer. Let's see whether we can translate your equations more directly:

```
def evolve_step(dt)
  r2 = @pos*@pos
  r3 = r2*sqrt(r2)
  old_acc = @pos*(-@mass/r3)
  @pos += @vel*dt + old_acc*0.5*dt*dt
  r2 = @pos*@pos
  r3 = r2*sqrt(r2)
  new_acc = @pos*(-@mass/r3)
  @vel += (old_acc + new_acc)*0.5*dt
end
```

Hey, it is even one line shorter than what you wrote!

Bob: But it introduces an extra variable, so there is a trade off. Yes, that would be fine as well. It depends on what you find more elegant.

In fact, note that none of these methods are very efficient. At the beginning of the next step, we will calculate the old acceleration, which is the same as what was called the new acceleration in the previous step. So while we are stepping along, we do every acceleration calculation twice, in your version as well as in mine. I was thinking about avoiding that redundancy, but I realized that that would make the code less clear. And as long as we are integrating only two bodies, there is really no need to optimize at this point.

Alice: That reminds me of Alan Perlis' saying: "premature optimization is the root of all evil".

Bob: I thought that quote was from Hoare, the inventer of qsort; I came across it in some of the writings of Knuth.

Alice: It is probably a sign of a a good idea that it gets attributed to different people.

Bob: But every good idea has to be used sparingly. Postponing optimization too long will never grow fruit.

Alice: Did you just make that up?

Bob: I just thought about balancing roots and fruits, as a warning against too much abstraction, modularity, cleanliness, and all that stands in the way of optimized code. But for now, as long as we are still working on a toy project, I really don't mind.

2.3 Tidying Up

Alice: You have told us how you wrote your first leapfrog version of `evolve_step`; and now I can see what you did in the much more condensed version in your code:

```
def leapfrog(dt)
  @vel += acc*0.5*dt
  @pos += @vel*dt
  @vel += acc*0.5*dt
end
```

Instead of repeating the three line calculation of the acceleration twice you put in into a method instead:

```
def acc
  r2 = @pos*@pos
  r3 = r2*sqrt(r2)
  @pos*(-@mass/r3)
end
```

Another nice example of a method without parameters, like the `ekin` and `epot` that we saw before.

Bob: I know you would like that. I'm curious, how would your version of the leapfrog look, in this notation? You start by introducing variables `old_acc` and `new_acc`, so that must then become:

```
def leapfrog(dt)
  old_acc = acc
  @pos += @vel*dt + old_acc*0.5*dt*dt
  new_acc = acc
  @vel += (old_acc + new_acc)*0.5*dt
end
```

So there: one line longer than my version, and with longer lines to boot.

Alice: That is interesting, isn't it? At first I really preferred my version, both because it was shorter and because it remained closer to the mathematical formulae that you had written down. But I must admit that I like the way you encapsulated – dare I say modularized – the acceleration into a separate method. And now suddenly my version looks a bit clumsy and lengthy.

Bob: But I'm glad we're going over the code so carefully together. I wouldn't have thought about your way of writing the leapfrog, and it is nice to try different variations, in order to see which one you like best.

2.4 A Subtle Bug

Alice: So I agree: we'll stick with yours. And your forward Euler now also looks very short.

```
def forward(dt)
  old_acc = acc
  @pos += @vel*dt
  @vel += old_acc*dt
end
```

Bob: In fact, my first try was even one line shorter:

```
def forward(dt)
  @pos += vel*dt
  @vel += acc*dt
end
```

Alice: That looks beautifully symmetric. I see that you left out the `@` sign at the right in front of `vel`; it doesn't make a difference, since `vel` is a method returning `@vel` anyway, and this way the symmetry between the two lines comes out better.

I like this form! What is wrong with it? You're not the type of person who likes to add an extra line without a good reason!

Bob: It is totally wrong. But it took me quite a while to figure that out. Can you guess what is wrong?

Alice: It looks right to me. In fact, it is classic forward Euler. The position gets increased by the velocity times time step, and the velocity gets increased by the acceleration times time step. What can be wrong with that?

Bob: Let me give you a hint. You praised the symmetry of the two lines. But are they *really* symmetric?

Alice: Is that a trick question? In both lines literally all characters are the same, except that `pos` above is replaced by its derivative `vel` below, and `vel` above is also replaced by its derivative `acc` below.

Bob: And in both cases the left-hand side of the equation contains an instance variable, and the right-hand side contains a method . . .

Alice: Yes, I had already commented on that, praising you! The left hand side is symmetric, because the variable holding `@vel` is exactly of the same sort of thing as the variable holding `@pos`. How much further do you want to push me to spell everything out? Okay. The right hand side is symmetric, because the method `acc` . . .

Bob: You see! I can see from your face that you got it.

Alice: Well, now that *is* a subtle bug. What a beauty!

Bob: Yeah, I agree, I first felt rather stupid, but now that I see your reaction, I think that I may have stumble upon an interesting variation of stupidity.

Alice: I can't wait to show this to the students, and let them figure it out.

Bob: In a class full of them, there may be a few that get it quicker than we did.

Alice: So much the better, more power to them! What a tricky situation. So yes, thanks to your prodding I saw it all as in a flash. They two lines are *not* symmetric. In the first line, `vel` just reads out a stored value, while in the second line, `acc` computes a value on the fly. And it does so using the `pos` value at that time, a value that has just been changed prematurely in the previous line. So the velocity increment will be wrong! And what is worse, you can't repair the damage by changing the order of the two lines:

```
def forward(dt)
  @vel += acc*dt
  @pos += vel*dt
end
```

In that case, the velocity gets updated correctly, with the acceleration based on the old `@pos` value, but now the position increment gets wrongly calculated, since it uses the new value for `vel`, while I should have used the old. A lose-lose situation, whatever order you give these two lines.

Bob: So I had to add a third line, introducing a temporary variable, `old_acc`, the acceleration at the beginning of the step.

Alice: I'm impressed that you found this bug at all.

Bob: Well, if you stare at something long enough, chances are that you stumble upon a moment of clarity.

Alice: But no guarantee. And even testing would not have revealed that bug, since the bug introduces a second-order error, and the error in a first-order method is second-order anyway. To be explicit, in the last version, the position increment would use a velocity value `vel` that would be off by an amount proportional to `dt`, so the product `vel*dt` would introduce an error in `@pos` that would be proportional to `dt*dt`. So the method would still be first-order accurate!

Bob: But it would no longer be a forward Euler method. Yes, it is tricky. It makes you wonder how many hidden bugs there are still lurking in even well-tested codes, let alone the many codes that are not-so-well tested . . .

2.5 Ordering an Integrator

Alice: Back to your two integrator methods. I very much like the idea of separating out the acceleration as a calculation that gets done in a separate method. But where is the magic that enables you to call the shots, when you call `evolve`? How does `evolve` know that a first argument of the string `"forward"` directs it to execute the method `forward`, and similarly that a first argument of the string `"leapfrog"` directs it to execute the method `leapfrog`?

Bob: That happens through the `send` method.

Alice: Another piece of Ruby magic, I take it. But how does it work? It seems too easy, though. You send the integration method on its way?

Bob: `send` is a method that comes with each object in Ruby. It is one of the general methods that each object inherits as soon as it is created. What `send` does is to call another method. `send` first reads the name of the other method from the string in its first argument, and then it passes its remaining arguments to the other method. So in our case `send("leapfrog", dt)`, for example amounts to the same as giving the command `leapfrog(dt)` directly.

Alice: That's really nice. Clearly, the designer of Ruby had a very good sense of what is needed to built a truly flexible language. What is his name?

Bob: Matsumoto. I don't know much about him, but yes, he clearly knows how to create a clean yet powerful language.

Alice: I'd like to meet him someday. Does he live in Japan?

Bob: He does.

Alice: Well, there are many astronomy conferences in Japan, so I hope I'll get a chance, some day. And if we really get to build a nice toy model in Ruby, he may enjoy playing with stars.

Bob: Now that you see how I have extended our `Body` class with a menu of integrators, would you like to see it all in action?

Alice: Sure. Let's try the hard case again, for ten time units. That seemed to take forever with the forward Euler method. Let's try it for a few time steps, both with forward Euler and with leapfrog.

2.6 A Comparison test

Bob: Okay. Let me first repeat forward Euler, for a couple choices. This was the largest time step for which we did not get an explosion:

```
dt = 0.001           # time step
dt_dia = 10          # diagnostics printing interval
dt_out = 10          # output interval
dt_end = 10          # duration of the integration
method = "forward"   # integration method
##method = "leapfrog"   # integration method
```

```
|gravity> ruby integrator_driver1a.rb < euler.in
dt = 0.001
dt_dia = 10
dt_out = 10
dt_end = 10
method = forward
at time t = 0, after 0 steps :
  E_kin = 0.125 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 0
  (E_tot - E_init) / E_init =-0
at time t = 10, after 10000 steps :
  E_kin = 0.0451 , E_pot =  -0.495 , E_tot = -0.45
             E_tot - E_init = 0.425
  (E_tot - E_init) / E_init =-0.486
  1.0000000000000000e+00
  2.0143551288236803e+00  1.6256533638564666e-01
 -1.5287552868811088e-01  2.5869644289548283e-01
```

Alice: That is a horrible error in the energy. But we have seen before that the positions get more accurate with a smaller time step, in such a way that the error in the position goes down linearly with the time step size. We can expect the energy error to scale in a similar way.

Bob: Here is the same run with a ten times smaller time step.

```
|gravity> ruby integrator_driver1b.rb < euler.in
dt = 0.0001
dt_dia = 10
dt_out = 10
dt_end = 10
method = forward
at time t = 0, after 0 steps :
  E_kin = 0.125 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 0
  (E_tot - E_init) / E_init =-0
at time t = 10, after 100000 steps :
  E_kin = 1.27 , E_pot =  -2.07 , E_tot = -0.8
             E_tot - E_init = 0.0749
  (E_tot - E_init) / E_init =-0.0856
  1.0000000000000000e+00
  2.9271673782679269e-01  3.8290774857970239e-01
 -1.5655189697698089e+00 -3.1395706386716327e-01
```

In fact, the energy went down by less than a factor of ten, but that may well be because we have not yet reached the linear regime.

Alice: That is probably the reason. With errors this large, you cannot expect an asymptotic behavior.

Bob: We'll just have to take an even smaller time step. Here goes:

```
|gravity> ruby integrator_driver1c.rb < euler.in
dt = 1.0e-05
dt_dia = 10
dt_out = 10
dt_end = 10
method = forward
at time t = 0, after 0 steps :
  E_kin = 0.125 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 0
  (E_tot - E_init) / E_init =-0
at time t = 10, after 1000000 steps :
  E_kin = 0.693 , E_pot =  -1.56 , E_tot = -0.865
             E_tot - E_init = 0.0103
  (E_tot - E_init) / E_init =-0.0117
  1.0000000000000000e+00
  5.1997064263476844e-01 -3.7681799204170163e-01
  1.1712678714357090e+00  1.1470087974069308e-01
```

Alice: That is going in the right direction at least, creeping closer to a factor ten in decrease of the energy errors.

Bob: But creeping very slowly. Before I'll be convinced, I want to see the time step shrinking by another factor of ten:

```
|gravity> ruby integrator_driver1d.rb < euler.in
dt = 1.0e-06
dt_dia = 10
dt_out = 10
dt_end = 10
method = forward
at time t = 0, after 0 steps :
  E_kin = 0.125 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 0
  (E_tot - E_init) / E_init =-0
at time t = 10, after 10000000 steps :
  E_kin = 0.566 , E_pot =  -1.44 , E_tot = -0.874
             E_tot - E_init = 0.00103
  (E_tot - E_init) / E_init =-0.00118
  1.0000000000000000e+00
  5.9216816556748519e-01 -3.6259219766731704e-01
  1.0441831294511545e+00  2.0515662908862703e-01
```

Alice: Ah, finally! Now we're in the linear regime.

Bob: Meanwhile, I am really curious to see what the leapfrog method will give us. Let's have a look:

```
dt = 0.001           # time step
dt_dia = 10          # diagnostics printing interval
dt_out = 10          # output interval
dt_end = 10          # duration of the integration
##method = "forward"   # integration method
method = "leapfrog"  # integration method
```

```
|gravity> ruby integrator_driver1e.rb < euler.in
dt = 0.001
dt_dia = 10
dt_out = 10
dt_end = 10
method = leapfrog
at time t = 0, after 0 steps :
  E_kin = 0.125 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 0
  (E_tot - E_init) / E_init =-0
at time t = 10, after 10000 steps :
  E_kin = 0.554 , E_pot =  -1.43 , E_tot = -0.875
             E_tot - E_init = 3.2e-07
  (E_tot - E_init) / E_init =-3.65e-07
  1.0000000000000000e+00
  5.9946121055215340e-01 -3.6090779482156415e-01
  1.0308896785838775e+00  2.1343145669114691e-01
```

Alice: Ah, *much* better. A thousand times longer time step, and yet an enormously better accuracy already! What will happen with a ten times smaller time step?

2.7 Scaling

```
|gravity> ruby integrator_driver1f.rb < euler.in
dt = 0.0001
dt_dia = 10
```

```
dt_out = 10
dt_end = 10
method = leapfrog
at time t = 0, after 0 steps :
  E_kin = 0.125 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 0
  (E_tot - E_init) / E_init =-0
at time t = 10, after 100000 steps :
  E_kin = 0.554 , E_pot =  -1.43 , E_tot = -0.875
             E_tot - E_init = 3.2e-09
  (E_tot - E_init) / E_init =-3.65e-09
  1.0000000000000000e+00
  5.9961599191051762e-01 -3.6063731614990768e-01
  1.0308077390676098e+00  2.1389066543649665e-01
```

Bob: A hundred times smaller error, as is appropriate for a second order method. What a relief, after the earlier slooooow convergence of forward Euler! Clearly leapfrog is a much much better integrator.

Alice: Yes, to get such a high accuracy would have taken forever with a forward Euler integrator. Congratulations, Bob!

Bob: Thanks, but don't get too excited yet: note that the error in the position is quite a bit larger than the error in the energy. The difference between the positions in the last two runs is of the order of 10^{-4}, while the next-to-last run already had an energy accuracy of a few time 10^{-7}.

Alice: This must be an example of what we talked about, when we first discussed the limits of using energy conservation as a check. The leapfrog is an example of a symplectic integration scheme, one which has better built-in energy conservation. Clearly it conserves energy better than position.

Bob: But since it is a second order scheme, also the accuracy of the position should increase by a factor hundred when we decrease the time step by a factor ten. Let's try and see.

```
|gravity> ruby integrator_driver1g.rb < euler.in
dt = 1.0e-05
dt_dia = 10
dt_out = 10
dt_end = 10
method = leapfrog
at time t = 0, after 0 steps :
  E_kin = 0.125 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 0
  (E_tot - E_init) / E_init =-0
at time t = 10, after 1000000 steps :
```

```
E_kin = 0.554 , E_pot =  -1.43 , E_tot = -0.875
           E_tot - E_init = 3.21e-11
(E_tot - E_init) / E_init =-3.66e-11
1.0000000000000000e+00
5.9961753925629424e-01 -3.6063461077225456e-01
1.0308069185587891e+00  2.1389525780602489e-01
```

Alice: You are right. The difference between the x component of the position in the last two runs is 2.10^{-6}, while the difference between the previous run and the run before that was 2.10^{-4}. So the scheme really is second-order accurate. Still, it would be nice to try another second order integrator, one that does not have built-in energy conservation.

Bob: I'll try to see what I can do, this evening. And who knows, I might even try my hand at a fourth-order integrator. But that will be it; we can't go on testing the two-body problem forever. We have other work to do: finding a reasonable graphics package, for one thing.

Alice: Definitely. Okay, see you tomorrow!

Chapter 3

Runge Kutta

3.1 Two New Integrators

Alice: Good morning, Bob! Any luck with extending your menu of integrators?

Bob: Yes, I added two Runge-Kutta integrators, one second-order and one fourth-order in accuracy. The changes are again minimal.

For the driver program there is no change at all. The only difference is that you can now call `evolve` with `method = "rk2"` to invoke the second-order Runge-Kutta integrator, or `"rk4"` for the fourth-order integrator.

Here are the additions to the `Body` class:

```
def rk2(dt)
  old_pos = pos
  half_vel = vel + acc*0.5*dt
  @pos += vel*0.5*dt
  @vel += acc*dt
  @pos = old_pos + half_vel*dt
end
```

```
def rk4(dt)
  old_pos = pos
  a0 = acc
  @pos = old_pos + vel*0.5*dt + a0*0.125*dt*dt
  a1 = acc
  @pos = old_pos + vel*dt + a1*0.5*dt*dt
  a2 = acc
```

```
    @pos = old_pos + vel*dt + (a0+a1*2)*(1/6.0)*dt*dt
    @vel = vel + (a0+a1*4+a2)*(1/6.0)*dt
  end
```

Alice: Even the fourth-order method is quite short. But let me first have a look at your second-order Runge-Kutta. I see you used an `old_pos` just like I used an `old_acc` in my version of the leapfrog.

Bob: Yes, I saw no other way but to remember some variables before they were overwritten.

Alice: My numerical methods knowledge is quite rusty. Can you remind me of the idea behind the Runge-Kutta algorithms?

3.2 A Leapcat Method

Bob: The Runge-Kutta method does not leap, as the leapfrog does. It is more careful. It feels its way forward by a half-step, sniffs out the situation, returns again, and only then it makes a full step, using the information gained in the half-step.

Alice: Like a cat?

Bob: Do cats move that way?

Alice: Sometimes, when they are trying to catch a mouse, they behave as if they haven't seen their prey at all. They just meander a bit, and then return, so that the poor animal thinks that it is safe now. But then, when the mouse comes out of hiding, the cat pounces.

Bob: So we should call the Runge-Kutta method the leapcat method?

Alice: Why not?

Bob: I'm not sure. For one thing, my code doesn't meander. You can tell your students what you like, I think I'll be conservative and stick to Runge Kutta.

Here are the equations that incorporate the idea of making a half-step first as a feeler:

$$\begin{aligned}
\mathbf{r}_{i+1/2} &= \mathbf{r}_i + \mathbf{v}_i \Delta t/2 \\
\mathbf{v}_{i+1/2} &= \mathbf{v}_i + \mathbf{a}_i \Delta t/2 \\
\mathbf{r}_{i+1} &= \mathbf{r}_i + \mathbf{v}_{i+1/2} \Delta t \\
\mathbf{v}_{i+1} &= \mathbf{v}_i + \mathbf{a}_{i+1/2} \Delta t
\end{aligned}$$

Alice: Ah yes, now it is very clear how you use the information from the half-step, in the form of the velocity and acceleration there, to improve the accuracy of the full step you take at the end.

Bob: Wait a minute. Looking again at the equations that I got from a book yesterday, I begin to wonder. Is this really different from the leapfrog method? We have seen that you can write the leapfrog in at least two quite different ways. Perhaps this is a third way to write it?

Alice: I think it is different, otherwise it would not have a different name, but it is a good idea to check for ourselves. Let us write out the effect of the equations above, by writing the half-step values in terms of the original values. First, let's do it for the position:

$$\begin{aligned}\mathbf{r}_{i+1} &= \mathbf{r}_i + \mathbf{v}_{i+1/2}\Delta t \\ &= \mathbf{r}_i + (\mathbf{v}_i + \mathbf{a}_i\Delta t/2)\Delta t \\ &= \mathbf{r}_i + \mathbf{v}_i\Delta t + \mathbf{a}_i(\Delta t)^2/2\end{aligned}$$

Bob: Hey, that is the exact same equation as we had for the leapfrog. Now I'm curious what the velocity equation will tell us.

$$\begin{aligned}\mathbf{v}_{i+1} &= \mathbf{v}_i + \mathbf{a}_{i+1/2}\Delta t \\ &= \mathbf{v}_i + ...\end{aligned}$$

Ah, that doesn't work, we cannot write $\mathbf{a}_{i+1/2}$ in terms of quantities at times i and $i+1$. It really is a new result, computed halfway.

Alice: You are right. So the two methods are really different then. Good to know. Still, let us see *how* different they are. The corresponding equation for the leapfrog is:

$$\mathbf{v}_{i+1} = \mathbf{v}_i + (\mathbf{a}_i + \mathbf{a}_{i+1})\Delta t/2$$

Bob: Ah, that is neat. The Leapfrog velocity jump uses the average acceleration, averaged between the values at the beginning and at the end of the leap. The Runge-Kutta method approximates this average by taking the midpoint value.

Alice: Or putting it the other way around, the leapfrog approximates the midpoint value by taking the average of the two accelerations.

Bob: Yes, you can look at it both ways. In fact, if you take your view, you can see that the leapfrog is twice as efficient as Runge-Kutta, in that it calculates only the accelerations at the integer time points. In contrast, the Runge-Kutta method also calculates the accelerations at the half-integer points in time. So it requires twice as many acceleration calculations.

Of course, in my current code that is not obvious, since I am calculating the acceleration twice in the leapfrog as well. However, it would be easy to rewrite it in such a way that it needs only one acceleration calculation. And in the Runge-Kutta method you cannot do that.

3.3 Translating Formulas Again

Alice: Going back to the original equations you wrote down for your second-order Runge-Kutta code:

$$\begin{aligned}
\mathbf{r}_{i+1/2} &= \mathbf{r}_i + \mathbf{v}_i \Delta t/2 \\
\mathbf{v}_{i+1/2} &= \mathbf{v}_i + \mathbf{a}_i \Delta t/2 \\
\mathbf{r}_{i+1} &= \mathbf{r}_i + \mathbf{v}_{i+1/2} \Delta t \\
\mathbf{v}_{i+1} &= \mathbf{v}_i + \mathbf{a}_{i+1/2} \Delta t
\end{aligned}$$

I must admit that your way of coding them up is not immediately transparent to me. If I had to implement these equations, I would have written something like:

```
def rk2(dt)
  old_pos = pos
  old_vel = vel
  half_pos = pos + vel*0.5*dt
  half_vel = vel + acc*0.5*dt
  @pos = half_pos
  half_acc = acc
  @vel = old_vel + half_acc*dt
  @pos = old_pos + half_vel*dt
end
```

Note that I first move the position variable `pos` from $\mathbf{r}_i$ to $\mathbf{r}_{i+1/2}$ in the fifth line of this method. Or more precisely: I assign the value of $\mathbf{r}_{i+1/2}$ to the variable `pos`. I then invoke the method `acc` on $\mathbf{r}_{i+1/2}$ in order to obtain $\mathbf{a}_{i+1/2}$, the value of which I then assign to the variable `half_acc`.

Bob: Yes, that is how I started as well. But then I realized that I could save three lines by writing them the way I did in the code:

```
def rk2(dt)
  old_pos = pos
  half_vel = vel + acc*0.5*dt
  @pos += vel*0.5*dt
  @vel += acc*dt
  @pos = old_pos + half_vel*dt
end
```

The number of times you calculate the acceleration is the same, but instead of introducing five new variables, I only introduced two.

Alice: it is probably a good idea to show both ways to the students, so that the whole process of coding up an algorithm becomes more transparent to them.

Bob: Yes, I'll make sure to mention that to them.

3.4 Energy Conservation

Alice: Since we know now that the two methods are different, it seems likely that the energy conservation in the Runge-Kutta method is a better indicator for the magnitude of the positional errors than it was in the leapfrog case. Shall we try the same values as we used before, but now for the Runge-Kutta method?

Bob: I'm all for it! Let me call the file `integrator_driver2.rb`:

```
dt = 0.001          # time step
dt_dia = 10         # diagnostics printing interval
dt_out = 10         # output interval
dt_end = 10         # duration of the integration
method = "rk2"      # integration method
```

```
|gravity> ruby integrator_driver2a.rb < euler.in
dt = 0.001
dt_dia = 10
dt_out = 10
dt_end = 10
method = rk2
at time t = 0, after 0 steps :
  E_kin = 0.125 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 0
  (E_tot - E_init) / E_init =-0
at time t = 10, after 10000 steps :
  E_kin = 0.555 , E_pot =  -1.43 , E_tot = -0.875
             E_tot - E_init = 6.02e-05
  (E_tot - E_init) / E_init =-6.88e-05
  1.0000000000000000e+00
  5.9856491479183715e-01 -3.6183772788952318e-01
  1.0319067591346045e+00  2.1153690796461602e-01
```

Alice: An energy error of order several time 10^{-5}, that is more than a hundred times worse than we saw for the leapfrog, where we had an error of a few times 10^{-7}, for the same time step value. Let's try a ten times smaller time step:

```
|gravity> ruby integrator_driver2b.rb < euler.in
dt = 0.0001
dt_dia = 10
dt_out = 10
dt_end = 10
method = rk2
at time t = 0, after 0 steps :
  E_kin = 0.125 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 0
  (E_tot - E_init) / E_init =-0
at time t = 10, after 100000 steps :
  E_kin = 0.554 , E_pot =  -1.43 , E_tot = -0.875
             E_tot - E_init = 6.06e-08
  (E_tot - E_init) / E_init =-6.92e-08
  1.0000000000000000e+00
  5.9961087073768127e-01 -3.6064562545351836e-01
  1.0308109943449486e+00  2.1387625542844693e-01
```

Bob: That is a surprise: the energy error has become a thousand times smaller, instead of a hundred. The Runge-Kutta seems to behave as if it is a third-order method, rather than a second order method.

Alice: That is interesting. I guess you could say that the errors of a second-order integrator are only guaranteed to be smaller than something that scales like the square of the time step, but still, this is a bit mysterious. Shall we shrink the time step by another factor of ten?

3.5 Accuracy in Position

Bob: Sure. But before doing that, note that the position error in our first run is about 10^{-3}, where we had 10^{-4} for the leapfrog. So the Runge-Kutta, for a time step of 0.001, is a hundred times worse in energy conservation and ten times worse in the accuracy of the positions, compared with the leapfrog. Then, at a time step of 0.0001, the energy error lags only a factor ten behind the leapfrog. Okay, let's go to a time step of 0.00001:

```
|gravity> ruby integrator_driver2c.rb < euler.in
dt = 1.0e-05
dt_dia = 10
dt_out = 10
dt_end = 10
method = rk2
at time t = 0, after 0 steps :
  E_kin = 0.125 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 0
  (E_tot - E_init) / E_init =-0
at time t = 10, after 1000000 steps :
  E_kin = 0.554 , E_pot =  -1.43 , E_tot = -0.875
             E_tot - E_init = 6.41e-11
  (E_tot - E_init) / E_init =-7.32e-11
  1.0000000000000000e+00
  5.9961749187967806e-01 -3.6063469285477523e-01
  1.0308069441700245e+00  2.1389511823566043e-01
```

Alice: The third-order style behavior continues! The energy error again shrunk by a factor a thousand. Now the leapfrog and Runge-Kutta are comparable in energy error, to within a factor of two or so.

Bob: The position error of our second run was 10^{-5}, as we can now see by comparison with the third run. That is strange. The positional accuracy increases by a factor 100, yet the energy accuracy increases by a factor 1000.

Alice: Let's shrink the step size by another factor of ten.

```
|gravity> ruby integrator_driver2.rb < euler.in
dt = 1.0e-06
dt_dia = 10
dt_out = 10
dt_end = 10
method = rk2
at time t = 0, after 0 steps :
  E_kin = 0.125 , E_pot =  -1, E_tot = -0.875
             E_tot - E_init = 0
  (E_tot - E_init) / E_init =-0
at time t = 10, after 10000000 steps :
  E_kin = 0.554 , E_pot =  -1.43, E_tot = -0.875
             E_tot - E_init = -1.66e-13
  (E_tot - E_init) / E_init =1.9e-13
  1.0000000000000000e+00
  5.9961755426085783e-01 -3.6063458453782676e-01
  1.0308069106006272e+00  2.1389530234024676e-01
```

Bob: This time the energy error shrunk only by two and a half orders of magnitude, about a factor 300, but still more than the factor hundred than we would have expected. Also, with 10^7 integration steps, I'm surprised we got even that far. At each time, round off errors must occur in the calculations that are of order 10^{-16}. Then statistical noise in so many calculations must be larger than that by at least the square root of the number of time steps, or $10^{3.5}.10^{-16} = 3.10^{-13}$.

Alice: Which is close to what we got. So that would suggest that further shrinking the time step would not give us more accuracy.

Bob: I would expect so much. But these calculations are taking a long time again, so I'll let the computer start the calculation, and we can check later. At least now we are pushing machine accuracy for 64-bit floating point with our second-order integrator; a while ago it took forever to get the errors in position down to one percent. I can't wait to show you my fourth-order integrator. That will go a lot faster. But first let's see what we get here.

. . . . .

3.6 Reaching the Round-Off Barrier

Alice: It has been quite a while now. Surely the code must have run now.

Bob: Indeed, here are the results.

```
|gravity> ruby integrator_driver2.rb < euler.in
dt = 1.0e-07
dt_dia = 10
dt_out = 10
dt_end = 10
method = rk2
at time t = 0, after 0 steps :
  E_kin = 0.125 , E_pot =  -1, E_tot = -0.875
             E_tot - E_init = 0
  (E_tot - E_init) / E_init =-0
at time t = 10, after 100000000 steps :
  E_kin = 0.554 , E_pot =  -1.43, E_tot = -0.875
             E_tot - E_init = -4.77e-13
  (E_tot - E_init) / E_init =5.45e-13
  1.0000000000000000e+00
  5.9961755488235224e-01 -3.6063458345517674e-01
  1.0308069102692998e+00  2.1389530417820268e-01
```

As we expected, the energy error could not shrink further. Instead, it grew larger, because the random accumulation of errors in ten times more time steps gave an increase of error by roughly the square root of ten, or about a factor three – just what we observe here.

Alice: Note that the positions now agree to within a factor of 10^{-9}. Once more a factor a hundred more accurate than the difference between the previous two integrations. Clearly the positional accuracy of the second order Runge-Kutta is second-order accurate, like that of the leapfrog.

Chapter 4

A Fourth-Order Integrator

4.1 Coefficients

Bob: Now it is really time to show off my fourth-order integrator.

Alice: Can you show me again the code?

Bob: Here it is:

```
def rk4(dt)
  old_pos = pos
  a0 = acc
  @pos = old_pos + vel*0.5*dt + a0*0.125*dt*dt
  a1 = acc
  @pos = old_pos + vel*dt + a1*0.5*dt*dt
  a2 = acc
  @pos = old_pos + vel*dt + (a0+a1*2)*(1/6.0)*dt*dt
  @vel = vel + (a0+a1*4+a2)*(1/6.0)*dt
end
```

Alice: That's quite a bit more complicated. I see coefficients of 1/6 and 1/3 and 2/3. How did you determine those?

Bob: I just looked them up in a book. There are several ways to choose those coefficients. Runge-Kutta formulas form a whole family, and the higher the order, the more choices there are for the coefficients. In addition, there are some extra simplifications that can be made in the case of a second order differential equation where there is no dependence on the first derivative.

Alice: As in the case of Newton's equation of motion, where the acceleration is dependent on the position, but not on the velocity.

Bob: Exactly. Here are the equations. They can be found in that famous compendium of equations, tables and graphs: *Handbook of Mathematical Functions*, by M. Abramowitz and I. A. Stegun, eds. [Dover, 1965]. You can even find the pages on the web now. Here is what I copied from a web page:

Second Order: $y'' = f(x,y)$

Runge-Kutta Method

25.5.22 $$y_{n+1} = y_n + h\left(y'_n + \frac{1}{6}(k_1 + 2k_2)\right) + O(h^4)$$

$$y'_{n+1} = y'_n + \frac{1}{6}k_1 + \frac{2}{3}k_2 + \frac{1}{6}k_3$$

$$k_1 = hf(x_n, y_n)$$

$$k_2 = hf\left(x_n + \frac{h}{2}, y_n + \frac{h}{2}y'_n + \frac{h}{8}k_1\right)$$

$$k_3 = hf\left(x_n + h, y_n + hy'_n + \frac{h}{2}k_2\right).$$

Alice: Quite a bit more complex than the second order one.

Bob: Yes, and of course it is written in a rather different form than the way I chose to implement it. I preferred to stay close to the same style in which I had written the leapfrog and second-order Runge Kutta. This means that I first had to rewrite the Abramowitz and Stegun expression.

4.2 Really Fourth Order?

Alice: Let's write it out specifically, just to check what you did. I know from experience how easy it is to make a mistake in these kinds of transformations.

Bob: So do I! Okay, the book starts with the differential equation:

$$y'' = f(x,y) \tag{4.1}$$

whereas we want to apply it to Newton's equation, which in a similar notation would read

$$\frac{d^2}{dt^2}\mathbf{r} = \mathbf{a}(t,\mathbf{r}) \tag{4.2}$$

with the only difference that of course there is no specific time dependence in Newton's equations of gravity, so we only have

$$\frac{d^2}{dt^2}\mathbf{r} = \mathbf{a}(\mathbf{r}) \tag{4.3}$$

To make a connection with Abramowitz and Stegun, their x becomes our time t, their y becomes our position vector $\mathbf{r}$, and their f becomes our acceleration vector $\mathbf{a}$. Also, their y' becomes our velocity vector $\mathbf{v}$. Finally, their h is our time step `dt`.

Alice: Let us rewrite their set of equations in terms of the dictionary you just provided. And instead of `dt` it is better to write Δt, since `dt` is not an infinitesimal quantity here, but a finite time interval. This then gives us:

$$\begin{aligned}
\mathbf{r}(t+\Delta t) &= \mathbf{r}(t) + \Delta t\left(\mathbf{v}(t) + \tfrac{1}{6}(\mathbf{k}_1 + 2\mathbf{k}_2)\right) \\
\mathbf{v}(t+\Delta t) &= \mathbf{v}(t) + \tfrac{1}{6}\mathbf{k}_1 + \tfrac{2}{3}\mathbf{k}_2 + \tfrac{1}{6}\mathbf{k}_3 \\
\mathbf{k}_1 &= \Delta t\mathbf{a}(\mathbf{r}(t)) \\
\mathbf{k}_2 &= \Delta t\mathbf{a}\left(\mathbf{r}(t) + \tfrac{1}{2}\Delta t\mathbf{v} + \tfrac{1}{8}\Delta t\mathbf{k}_1\right) \\
\mathbf{k}_3 &= \Delta t\mathbf{a}\left(\mathbf{r}(t) + \Delta t\mathbf{v} + \tfrac{1}{2}\Delta t\mathbf{k}_2\right)
\end{aligned}$$

In order to make a connection with your code, we can define three variables $\mathbf{a}_0$, $\mathbf{a}_1$, and $\mathbf{a}_2$ as follows, taking the right hand sides of the last three equations above, but without the Δt factor:

$$\begin{aligned}
\mathbf{a}_0 &= \mathbf{a}(\mathbf{r}(t)) \\
\mathbf{a}_1 &= \mathbf{a}\left(\mathbf{r}(t) + \tfrac{1}{2}\Delta t\mathbf{v} + \tfrac{1}{8}\Delta t\mathbf{k}_1\right) \\
\mathbf{a}_2 &= \mathbf{a}\left(\mathbf{r}(t) + \Delta t\mathbf{v} + \tfrac{1}{2}\Delta t\mathbf{k}_2\right)
\end{aligned}$$

Comparing this with the definitions of $\mathbf{k}_1$, $\mathbf{k}_2$, and $\mathbf{k}_3$, we can eliminate all $\mathbf{k}$ values from the right hand side:

$$\begin{aligned}
\mathbf{a}_0 &= \mathbf{a}(\mathbf{r}(t)) \\
\mathbf{a}_1 &= \mathbf{a}\left(\mathbf{r}(t) + \tfrac{1}{2}\Delta t\mathbf{v} + \tfrac{1}{8}(\Delta t)^2\mathbf{a}_0\right) \\
\mathbf{a}_2 &= \mathbf{a}\left(\mathbf{r}(t) + \Delta t\mathbf{v} + \tfrac{1}{2}(\Delta t)^2\mathbf{a}_1\right)
\end{aligned}$$

And indeed, these are exactly the three variables `a0`, `a1`, and `a2`, that are computed in your `rk4` method.

We can now rewrite the first two of Abramowitz and Stegun's equations as:

$$\begin{aligned}
\mathbf{r}(t+\Delta t) &= \mathbf{r}(t) + \Delta t\mathbf{v}(t) + \tfrac{1}{6}(\Delta t)^2(\mathbf{a}_0 + 2\mathbf{a}_1) \\
\mathbf{v}(t+\Delta t) &= \mathbf{v}(t) + \tfrac{1}{6}\Delta t(\mathbf{a}_0 + 4\mathbf{a}_1 + \mathbf{a}_2)
\end{aligned}$$

And these correspond precisely to the last two lines in your `rk4` method.

Bob: Yes, that was how I derived my code, but I must admit, I did not write it down as neatly and convincingly as you just did.

Alice: So yes, you have indeed implemented Abramowitz and Stegun's fourth-order Runge-Kutta scheme. But wait a minute, what you found here, equation 22.5.22, shows a right-hand side with a stated position error of order h^4 – which would suggest that the scheme is only third-order accurate.

Bob: Hey, that is right. If you make h smaller, the number of steps will go up according to $1/h$, and therefore the total error for a given problem will grow proportionally to $h^4(1/h) \propto h^3$. This would indeed imply that this method is third-order. But that would be very unusual. In all text books I've seen, you mostly come across second-order and fourth-order Runge-Kuttas. While you certainly can construct a third-order version, I wouldn't have expected Abramowitz and Stegun to feature one. Besides, look at the last expression for the velocity. Doesn't that resemble Simpson's rule, suggesting fourth-order accuracy. I'm really puzzled now.

Alice: Could it be a misprint?

Bob: Anything is possible, though here that is not very likely. This is such a famous book, that you would expect the many readers to have debugged the book thoroughly.

Alice: Let's decide for ourselves what is true and what is not.

Bob: Yes. And either way, whatever the outcome, it will be a good exercise for the students. Let's first test it numerically.

Alice: Fine with me. But then I want to derive it analytically as well, to see whether we really can understand the behavior of the numerical results from first principles.

And hey, I just noticed that the expression for the velocity in equation 22.5.22 does not have *any* error estimate. For the scheme to be really fourth-order, not only should the error per step in the first line be of order h^5, but the second line should *also* have an error of order h^5. I want to see whether we can derive both facts, if true.

4.3 Comparing the Four Integrators

Bob: Whether those facts are true or not, that will be easy to figure out numerically. Let us start again with a time step of 0.001, for a duration of ten time units.

Alice: Just to compare, why don't you run them for all four schemes.

Bob: I am happy to do so. And while we are not sure yet whether our higher-order Runge-Kutta scheme is 3rd order or 4th order, let me continue to call it

4th order, since I've called it `rk4` in my code. If it turns out that it is 3rd order, I'll go back and rename it to be `rk3`.

Alice: Fair enough.

Bob: Okay: forward Euler, leapfrog, 2nd order R-K, and hopefully-4th order R-K:

```
|gravity> ruby integrator_driver2d.rb < euler.in
dt = 0.001
dt_dia = 10
dt_out = 10
dt_end = 10
method = forward
at time t = 0, after 0 steps :
  E_kin = 0.125 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 0
  (E_tot - E_init) / E_init =-0
at time t = 10, after 10000 steps :
  E_kin = 0.0451 , E_pot =  -0.495 , E_tot = -0.45
             E_tot - E_init = 0.425
  (E_tot - E_init) / E_init =-0.486
  1.0000000000000000e+00
  2.0143551288236803e+00  1.6256533638564666e-01
 -1.5287552868811088e-01  2.5869644289548283e-01
```

Alice: Ah, this is the result from the forward Euler algorithm, because the fifth line of the output announces that this was the integration method used. As we have seen before, energy conservation has larger errors. We have an absolute total energy error of 0.425, and a relative change in total energy of -0.486. I see that you indeed used `dt = 0.001` from what is echoed on the first line in the output. And yes, for these choices of initial conditions and time step size we had seen that things went pretty badly.

Bob: The leapfrog does better:

```
|gravity> ruby integrator_driver2e.rb < euler.in
dt = 0.001
dt_dia = 10
dt_out = 10
dt_end = 10
method = leapfrog
at time t = 0, after 0 steps :
  E_kin = 0.125 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 0
  (E_tot - E_init) / E_init =-0
```

```
at time t = 10, after 10000 steps :
  E_kin = 0.554 , E_pot =  -1.43 , E_tot = -0.875
             E_tot - E_init = 3.2e-07
  (E_tot - E_init) / E_init =-3.65e-07
  1.0000000000000000e+00
  5.9946121055215340e-01 -3.6090779482156415e-01
  1.0308896785838775e+00  2.1343145669114691e-01
```

Alice: Much better indeed, as we had seen before. A relative energy error of -4×10^{-7} is great, an error much less than one in a million.

Bob: Time for second-order Runge-Kutta:

```
|gravity> ruby integrator_driver2f.rb < euler.in
dt = 0.001
dt_dia = 10
dt_out = 10
dt_end = 10
method = rk2
at time t = 0, after 0 steps :
  E_kin = 0.125 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 0
  (E_tot - E_init) / E_init =-0
at time t = 10, after 10000 steps :
  E_kin = 0.555 , E_pot =  -1.43 , E_tot = -0.875
             E_tot - E_init = 6.02e-05
  (E_tot - E_init) / E_init =-6.88e-05
  1.0000000000000000e+00
  5.9856491479183715e-01 -3.6183772788952318e-01
  1.0319067591346045e+00  2.1153690796461602e-01
```

Alice: Not as good as the leapfrog, but good enough. What I'm curious about is how your hopefully-fourth-order Runge-Kutta will behave.

Bob: Here is the answer:

```
|gravity> ruby integrator_driver2g.rb < euler.in
dt = 0.001
dt_dia = 10
dt_out = 10
dt_end = 10
method = rk4
at time t = 0, after 0 steps :
  E_kin = 0.125 , E_pot =  -1 , E_tot = -0.875
```

```
            E_tot - E_init = 0
  (E_tot - E_init) / E_init =-0
at time t = 10, after 10000 steps :
  E_kin = 0.554 , E_pot =  -1.43 , E_tot = -0.875
            E_tot - E_init = -2.46e-09
  (E_tot - E_init) / E_init =2.81e-09
  1.0000000000000000e+00
  5.9961758437074986e-01 -3.6063455639926667e-01
  1.0308068733946525e+00  2.1389536225475009e-01
```

Alice: What a difference! Not only do we have much better energy conservation, on the order of a few times 10^{-9}, also the positional accuracy is already on the level of a few times 10^{-8}, when we compare the output with our most accurate previous runs. And all that with hardly any waiting time.

Bob: Yes, I'm really glad I threw that integrator into the menu. The second-order Runge-Kutta doesn't fare as well as the leapfrog, at least for this problem, even though they are both second-order. But the hopefully-fourth-order Runge-Kutta wins hands down.

4.4 Fourth Order Runge-Kutta

Alice: Let's make the time step ten times smaller:

```
|gravity> ruby integrator_driver2h.rb < euler.in
dt = 0.0001
dt_dia = 10
dt_out = 10
dt_end = 10
method = rk4
at time t = 0, after 0 steps :
  E_kin = 0.125 , E_pot =  -1 , E_tot = -0.875
            E_tot - E_init = 0
  (E_tot - E_init) / E_init =-0
at time t = 10, after 100000 steps :
  E_kin = 0.554 , E_pot =  -1.43 , E_tot = -0.875
            E_tot - E_init = -8.33e-14
  (E_tot - E_init) / E_init =9.52e-14
  1.0000000000000000e+00
  5.9961755488723312e-01 -3.6063458344261029e-01
  1.0308069102701605e+00  2.1389530419780176e-01
```

Bob: What a charm! Essentially machine accuracy. This makes it pretty obvious that fourth-order integrators win out hands down, in problems like this, over second-order integrators. Higher order integrators may be even faster, but that is something we can explore later.

Alice: Well done. And, by the way, this does suggest that the scheme that you copied from that book is indeed fourth-order. It almost seems better than fourth order.

Bob: Let me try a far larger time step, but a much shorter duration, so that we don't have to integrate over a complicated orbit. How about these two choices. First:

```
|gravity> ruby integrator_driver2i.rb < euler.in
dt = 0.1
dt_dia = 0.1
dt_out = 0.1
dt_end = 0.1
method = rk4
at time t = 0, after 0 steps :
  E_kin = 0.125 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 0
  (E_tot - E_init) / E_init =-0
at time t = 0.1, after 1 steps :
  E_kin = 0.129 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 1.75e-08
  (E_tot - E_init) / E_init =-2.01e-08
  1.0000000000000000e+00
  9.9499478923153439e-01  4.9916431937376750e-02
 -1.0020915515250550e-01  4.9748795077019681e-01
```

And then:

```
|gravity> ruby integrator_driver2j.rb < euler.in
dt = 0.01
dt_dia = 0.1
dt_out = 0.1
dt_end = 0.1
method = rk4
at time t = 0, after 0 steps :
  E_kin = 0.125 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 0
  (E_tot - E_init) / E_init =-0
at time t = 0.1, after 10 steps :
  E_kin = 0.129 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 1.79e-12
  (E_tot - E_init) / E_init =-2.04e-12
```

```
  1.0000000000000000e+00
  9.9499478009063858e-01  4.9916426216739009e-02
 -1.0020902861389222e-01  4.9748796005932194e-01
```

Alice: The time step went down by a factor of 10, and the energy conservation got better by a factor of almost exactly 10^4. We really seem to have a fourth order integration scheme.

Bob: This proves it. And I can keep the name `rk4`, since it lives up to its claim. Good!

Alice: At least it comes close to living up to its name, and it sort-of proves it.

4.5 Snap, Crackle, and Pop

Bob: What do you mean with 'comes close'? What more proof would you like to have?

Alice: I would still be happier if I could really prove it, with pen and paper, rather than through the plausibility of numerical experimentation.

Bob: Go ahead, if you like!

Alice: Let us take your implementation

```
def rk4(dt)
  old_pos = pos
  a0 = acc
  @pos = old_pos + vel*0.5*dt + a0*0.125*dt*dt
  a1 = acc
  @pos = old_pos + vel*dt + a1*0.5*dt*dt
  a2 = acc
  @pos = old_pos + vel*dt + (a0+a1*2)*(1/6.0)*dt*dt
  @vel = vel + (a0+a1*4+a2)*(1/6.0)*dt
end
```

and let us determine analytically what the first five terms in a Taylor series in `dt` look like. We can then see directly whether the error term is proportional to h^4, as the book claims, or h^5, as our calculations indicate.

To start with, let us look at your variable `a2`, which is the acceleration after a time step `dt`, starting from an initial acceleration `a0`. A Taylor expansion can approximate `a2` as:

$$\mathbf{a}_2 = \mathbf{a}_0 + \frac{d\mathbf{a}}{dt}\Delta t + \frac{1}{2}\frac{d^2\mathbf{a}}{dt^2}(\Delta t)^2 + \frac{1}{6}\frac{d^3\mathbf{a}}{dt^3}(\Delta t)^3 + \frac{1}{24}\frac{d^4\mathbf{a}}{dt^4}(\Delta t)^4 \qquad (4.4)$$

Figure 4.1: Snap, Crackle, and Pop

where all the derivatives are understood to be evaluated at the same time as $\mathbf{a}_0$, namely at the beginning of our time step.

Bob: The first derivative of the acceleration, $d\mathbf{a}/dt$, is sometimes called the *jerk*. How about the following notation:

$$\mathbf{j} = \frac{d^3}{dt^3}\mathbf{r} = \frac{d\mathbf{a}}{dt} \tag{4.5}$$

Alice: Fine with me. I believe the term 'jerk' has crept into the literature relatively recently, probably originally as a pun. If a car or train changes acceleration relatively quickly you experience not a smoothly accelerating or decelerating motion, but instead a rather 'jerky' one.

Bob: It may be more difficult to come up with terms for the unevenness in the jerk.

Alice: Actually, I saw somewhere that someone had used the words *snap*, *crackle*, and *pop*, to describe the next three terms.

Bob: As in the rice crispies? Now that will confuse astronomers who did not grow up in the United States! If they haven't seen the rice crispy commercials, they will have no clue why we would use those names. And come to think of it, I don't have much of a clue yet either.

Alice: Ah, but the point is that the names of these three cheerful characters lend themselves quite well to describe more-than-jerky behavior. After all, the popping of a rice crispy is a rather sudden change of state.

Bob: Okay, now I get the drift of the argument. A sudden snap comes to mind, as a change in jerk. And what changes its state more suddenly than a snap? Well, perhaps something that crackles, although that is pushing it a bit. But a pop is certainly a good word for something that changes high derivatives of positions in a substantial way!

4.6 An Attempt at an Analytical Proof

Alice: Okay, let's make it official:

$$\begin{aligned} \mathbf{s} &= \frac{d^4}{dt^4}\mathbf{r} &= \frac{d^2}{dt^2}\mathbf{a} \\ \mathbf{c} &= \frac{d^5}{dt^5}\mathbf{r} &= \frac{d^3}{dt^3}\mathbf{a} \\ \mathbf{p} &= \frac{d^6}{dt^6}\mathbf{r} &= \frac{d^4}{dt^4}\mathbf{a} \end{aligned} \tag{4.6}$$

Which turns my earlier equation into:

$$\mathbf{a}_2 = \mathbf{a}_0 + \mathbf{j}_0\Delta t + \tfrac{1}{2}\mathbf{s}_0(\Delta t)^2 + \tfrac{1}{6}\mathbf{c}_0(\Delta t)^3 + \tfrac{1}{24}\mathbf{p}_0(\Delta t)^4 \tag{4.7}$$

Bob: Much more readable.

Alice: And your other variable `a1`, which indicates the acceleration after only half a time step, now becomes:

$$\mathbf{a}_1 = \mathbf{a}_0 + \tfrac{1}{2}\mathbf{j}_0\Delta t + \tfrac{1}{8}\mathbf{s}_0(\Delta t)^2 + \tfrac{1}{48}\mathbf{c}_0(\Delta t)^3 + \tfrac{1}{384}\mathbf{p}_0(\Delta t)^4 \tag{4.8}$$

Bob: Ah, and now we can evaluate the last two lines in my `rk4` method:

```
    @pos = old_pos + vel*dt + (a0+a1*2)*(1/6.0)*dt*dt
    @vel = vel + (a0+a1*4+a2)*(1/6.0)*dt
```

Alice: Yes. These two lines translate to:

$$\begin{aligned} \mathbf{r}_1 &= \mathbf{r}_0 + \mathbf{v}_0\Delta t + \tfrac{1}{6}(\mathbf{a}_0 + 2\mathbf{a}_1)(\Delta t)^2 \\ \mathbf{v}_1 &= \mathbf{v}_0 + \tfrac{1}{6}(\mathbf{a}_0 + 4\mathbf{a}_1 + \mathbf{a}_2)\Delta t \end{aligned} \tag{4.9}$$

Upon substitution, the first line becomes:

$$\begin{aligned} \mathbf{r}_1 &= \mathbf{r}_0 + \mathbf{v}_0\Delta t + \tfrac{1}{6}\mathbf{a}_0(\Delta t)^2 + \tfrac{1}{3}\mathbf{a}_1(\Delta t)^2 \\ &= \mathbf{r}_0 + \mathbf{v}_0\Delta t + \tfrac{1}{2}\mathbf{a}_0(\Delta t)^2 + \tfrac{1}{6}\mathbf{j}_0(\Delta t)^3 + \tfrac{1}{24}\mathbf{s}_0(\Delta t)^4 + \tfrac{1}{144}\mathbf{c}_0(\Delta t)^5 + O((\Delta t)^6) \end{aligned}$$

And the second line becomes:

$$\begin{aligned}\mathbf{v}_1 &= \mathbf{v}_0 + \tfrac{1}{6}\mathbf{a}_0\Delta t + \tfrac{2}{3}\mathbf{a}_1\Delta t + \tfrac{1}{6}\mathbf{a}_2\Delta t \\ &= \mathbf{v}_0 + \mathbf{a}_0\Delta t + \tfrac{1}{2}\mathbf{j}_0(\Delta t)^2 + \tfrac{1}{6}\mathbf{s}_0(\Delta t)^3 + \tfrac{1}{24}\mathbf{c}_0(\Delta t)^4 + \tfrac{5}{576}\mathbf{p}_0(\Delta t)^5 + O((\Delta t)^6)\end{aligned}$$

Bob: Very nice. In both cases the terms up to $(\Delta t)^4$ are perfectly correct, and the error starts in the $(\Delta t)^5$ term, where the Taylor series coefficient would have been $\frac{1}{120}$. So the leading error terms are:

$$\begin{aligned}\delta\mathbf{r}_1 &= -\tfrac{1}{720}\mathbf{c}_0(\Delta t)^5 \\ \delta\mathbf{v}_1 &= \tfrac{1}{2880}\mathbf{p}_0(\Delta t)^5\end{aligned}$$

This proves that the `rk4` Runge-Kutta method is truly fourth-order, and that there was a typo in Abramowitz and Stegun.

Alice: Or so it seems. But I must admit, I'm not completely comfortable.

4.7 Shocked

Bob: Boy, you're not easy to please! I was ready to declare victory when we had our numerical proof, and you convinced me to do the additional work of doing an analytical check. If it were up to me, I would believe numerics over analytics, any day. Seeing is believing, the proof is in the pudding, and all that.

Alice: Well, it *really* is better to have *both* a numerical demonstration *and* an analytical proof. It is just too easy to be fooled by coincidences or special cases, in numerical demonstrations. And at the same time it is also easy to make a mistake in doing analytical checks. So I think it is only wise to do both, especially with something so fundamental as an N-body integrator.

And I'm glad we did. But something doesn't feel quite right. Let me just review what we have done. We first expanded the orbit in a Taylor series, writing everything as a function of time. Then we showed that the orbit obeyed the equations of motion, and . . . .

Bob: . . . and then you looked shocked.

Alice: Not only do I look shocked, I *am* shocked! How could we make such a stupid mistake!?!

Bob: I beg your pardon?

Alice: There is something quite wrong with the Taylor series I wrote down in Eq. (4.4).

Bob: I looks like a perfectly fine Taylor series to me. All the coefficients are correct, and each next term is a higher derivative of the acceleration. What could possibly be wrong with it?

Alice: Formally it looks right, yes, that is exactly what is so misleading about it. But it is dead wrong.

Bob: Give me a hint.

Alice: Here is the hint. This Taylor series is formally correct, alright, for the case where you really know the function on the left hand side, and you really know all the derivatives on the right hand side. In that case, what you leave out is really fifth-order in Δt.

Bob: So?

Alice: The problem is, we *don't* have such idealized information. Look at Eq. (4.7), or no, even better, look at Eq. (4.8).

Bob: I must be missing something. Here is Eq. (4.8) again:

$$\mathbf{a}_1 = \mathbf{a}_0 + \tfrac{1}{2}\mathbf{j}_0\Delta t + \tfrac{1}{8}\mathbf{s}_0(\Delta t)^2 + \tfrac{1}{48}\mathbf{c}_0(\Delta t)^3 + \tfrac{1}{384}\mathbf{p}_0(\Delta t)^4 \tag{4.10}$$

It still looks perfectly right to me.

Alice: Yes, formally, true. But where do we apply it to? That's the problem. Let us recall how we actually *compute* $\mathbf{a}_1$. Here is your implementation of `rk4` again:

```
def rk4(dt)
  old_pos = pos
  a0 = acc
  @pos = old_pos + vel*0.5*dt + a0*0.125*dt*dt
  a1 = acc
  @pos = old_pos + vel*dt + a1*0.5*dt*dt
  a2 = acc
  @pos = old_pos + vel*dt + (a0+a1*2)*(1/6.0)*dt*dt
  @vel = vel + (a0+a1*4+a2)*(1/6.0)*dt
end
```

Now tell me, how accurate *is* this expression for $\mathbf{a}_1$, or in the terms of the code, for `a1`

Bob: It is based on the position calculated in the third line of the body of the code, which is second order accurate in `dt`. So this means that `a1` must be second-order accurate at this point. But that's okay, since `a2` is already higher-order accurate, since it uses the information of `a1` which was used to construct a better approximation for the position, in the fifth line of the code. And then the next line presumably gives an even better approximation. My guess would be that, while `a1` is second-order accurate in `dt`, `a2` is third-order accurate, and the next acceleration evaluation, at the beginning of the next step, will then be fourth-order accurate; that will be `a0` in the following round. This is the whole point of a Runge-Kutta approach: you bootstrap your way up to the required accuracy.

Alice: Yes, I agree with all that, but that is far more than we need right now. Let's just stick with `a1`, and only `a1`. It was second-order accurate in `dt`, you just said. Now how in the world can a second-order accurate formula give any information in a comparison with four subsequent derivatives in Eq. (4.8) ???

Bob: Aaaaah, *now* I see what you are driving at. Well, I suppose it can't.

Alice: And similarly, your guess was that `a2` was only third-order accurate, right? So it does not make sense to require it to fit snugly up to fourth-order in Eq. (4.7), which is the expression we started with, Eq. (4.4).

Bob: Well, if you put it that way, I must admit, no, that would not make sense. How tricky!

Alice: How tricky indeed! I was fully convinced that we were doing the right thing.

Bob: So was I. How dangerous, to rely on formal expressions!

Alice: And now that we have spotted the fallacy in our reasoning, we can see what we *should* have done: in Eq. (4.4), we should have expanded not only the right-hand side, but also the left-hand side, given that there are approximations involved there as well. Only after doing that, can we equate both sides in all honesty!

Bob: Yes, that is the only way, that much is clear now. Okay, you've convinced me. We really should keep track of all the approximations involved.

Alice: Now let's do it!

Chapter 5

A Complete Derivation

5.1 The One-Dimensional Case

Bob: You have a very determined look in your eyes.

Alice: Yes, I feel determined. I want to know for sure that the integrator listed in Abramowitz and Stegun is fourth-order, and I want to prove it.

In the form given in their book, they are dealing with a force that has the most general form: it depends on position, velocity, and time. In our notation, their differential equation would read

$$\frac{d^2}{dt^2}\mathbf{r} = \mathbf{f}(\mathbf{r}, \mathbf{v}, t) = \mathbf{f}(\mathbf{r}(t), \mathbf{v}(t), t) \tag{5.1}$$

Bob: Fortunately, in our case, we only have to deal with a dependence on position, not on velocity or time.

Alice: That is true, but I would like to prove their original formula.

Bob: I don't think that is necessary. We don't have any magnetic force, or anything else that shows forces that depend on velocity.

Alice: Yes, but now that we've spotted what seems like a mistake, I don't want to be too facile. Okay, let's make a compromise. I'll drop the velocity dependence; hopefully our derivation, if successful, should point the way to how you could include that extra dependence. But I would like to keep the time dependence, just to see for myself how that mixes with the space dependence.

And if you want a motivation: imagine that our star cluster is suspended within the background tidal field of the galaxy. Such a field can be treated as a slowly varying, time-dependent perturbation, exactly of this form:

$$\frac{d^2}{dt^2}\mathbf{r} = \mathbf{f}(\mathbf{r}, t) = \mathbf{f}(\mathbf{r}(t), t) \tag{5.2}$$

Bob: Fair enough. How shall we start?

Alice: If we *really* wanted to be formally correct, we could track the four-dimensional variations in space and time, taking into account the three-dimensional spatial gradient of the interaction terms.

Bob: But from the way you just said that, I figure that that would be too much, even for you. I'm glad to hear that even you have your limits in wanting to check things analytically!

Alice: I sure have. And also, I don't think such a formal treatment would be necessary. If we can prove things in one dimension, I'm convinced that we could rewrite our proof to three dimensions without too many problems, albeit with a lot more manipulations.

5.2 A Change in Notation

Bob: You won't hear me arguing against simplification! But I propose that we use a somewhat more intuitive notation than what Abramowitz and Stegun gave us. To use x to indicate time and y to indicate position is confusing, to say the least. Let us look at the text again:

Second Order: $y''=f(x,y)$

Runge-Kutta Method

25.5.22 $$y_{n+1}=y_n+h\left(y'_n+\frac{1}{6}(k_1+2k_2)\right)+O(h^4)$$

$$y'_{n+1}=y'_n+\frac{1}{6}k_1+\frac{2}{3}k_2+\frac{1}{6}k_3$$

$$k_1=hf(x_n,y_n)$$

$$k_2=hf\left(x_n+\frac{h}{2}, y_n+\frac{h}{2}y'_n+\frac{h}{8}k_1\right)$$

$$k_3=hf\left(x_n+h, y_n+hy'_n+\frac{h}{2}k_2\right).$$

How about the following translation, where the position r, velocity v, and a are all scalar quantities, since we are considering an effectively one-dimensional problem:

$$\begin{array}{ccc}
x & \rightarrow & t \\
y & \rightarrow & r \\
f(x,y) & \rightarrow & f(r,t) \\
y' = \dfrac{dy}{dx} & \rightarrow & \dot{r} = \dfrac{dr}{dt} = v \\
y'' = \dfrac{d^2y}{dx^2} & \rightarrow & \ddot{r} = \dfrac{d^2r}{dt^2} = \dfrac{dv}{dt} = a \\
h & \rightarrow & \Delta t = \tau \\
k_i & \rightarrow & k_i\tau \\
n & \rightarrow & 0
\end{array}$$

The equations, as proposed by Abramowitz and Stegun, then become:

$$\begin{aligned}
\ddot{r} &= f(r,t) \\
\\
r_1 &= r_0 + \left(v_0 + \tfrac{1}{6}\left(k_1 + 2k_2\right)\tau\right)\tau \\
v_1 &= v_0 + \tfrac{1}{6}\left(k_1 + 4k_2 + k_3\right)\tau \\
\\
k_1 &= f(r_0, 0) \\
k_2 &= f(r_0 + \tfrac{1}{2}v_0\tau + \tfrac{1}{8}k_1\tau^2, \tfrac{1}{2}\tau) \\
k_3 &= f(r_0 + v_0\tau + \tfrac{1}{2}k_2\tau^2, \tau)
\end{aligned} \tag{5.3}$$

5.3 The Auxiliary k Variables

Alice: Yes, I find that notation much more congenial, and I'm glad to see the explicitly space and time dependence now in the definitions of the three k variables.

Let us work out what these k variables actually look like, when we substitute the orbit dependence in the right hand side. What we see there is an interesting interplay between a physical interaction term f that has as an argument the mathematical orbit $r(t)$.

Since you have introduced τ as the increment in time, starting from time zero, let me introduce the variable ρ_1 to indicate the increment in space, the increment in position at the time that we evaluate k_2

$$\rho_1 = \tfrac{1}{2}v_0\tau + \tfrac{1}{8}k_1\tau^2 \tag{5.4}$$

For consistency, let me also use

$$\tau_1 = \tfrac{1}{2}\tau \tag{5.5}$$

for the increment in time, when we evaluate k_2. Let me also abbreviate $f_0 = f(r_0, 0)$.

What I propose to do now is to expand k_1 in a double Taylor series, simultaneously in space and in time. This will hopefully repair the oversight we made before, when we only considered a time expansion.

Bob: How do you do a two-dimensional Taylor series expansion?

Alice: Whenever you have mixed derivatives, such as

$$\frac{\partial^3 f}{\partial r^2 \partial t} = f_{rrt} \tag{5.6}$$

you multiply the coefficients you would give to each dimension separately; in this case you would get a factor of one half, because of the second partial derivative with respect to space; the single time derivative just gives a factor one. You can easily check how this works when you look at a two-dimensional landscape, and you imagine how you pick up difference in height by moving North-South and East-West. Anyway, here is the concrete example, for our case, where I use the abbreviation that I introduced above, with partial differentiation indicated with subscripts:

$$\begin{aligned} k_2 = f_0 \;&+\; f_r\rho_1 + f_t\tau_1 \\ &+\; \tfrac{1}{2}f_{rr}\rho_1^2 + f_{rt}\rho_1\tau_1 + \tfrac{1}{2}f_{tt}\tau_1^2 \\ &+\; \tfrac{1}{6}f_{rrr}\rho_1^3 + \tfrac{1}{2}f_{rrt}\rho_1^2\tau_1 + \tfrac{1}{2}f_{rtt}\rho_1\tau_1^2 + \tfrac{1}{6}f_{ttt}\tau_1^3 + O(\tau_1^4) \end{aligned} \tag{5.7}$$

where I have grouped the different orders of expansion in terms of small variables on different lines. Using the fact that $\tau_1 = \frac{1}{2}\tau$, we can also write this as

$$\begin{aligned} k_2 = f_0 \;&+\; f_r\rho_1 + \tfrac{1}{2}f_t\tau \\ &+\; \tfrac{1}{2}f_{rr}\rho_1^2 + \tfrac{1}{2}f_{rt}\rho_1\tau + \tfrac{1}{8}f_{tt}\tau^2 \\ &+\; \tfrac{1}{6}f_{rrr}\rho_1^3 + \tfrac{1}{4}f_{rrt}\rho_1^2\tau + \tfrac{1}{8}f_{rtt}\rho_1\tau^2 + \tfrac{1}{48}f_{ttt}\tau^3 + O(\tau_1^4) \end{aligned} \tag{5.8}$$

Similarly, we can introduce

$$\rho_2 = v_0\tau + \tfrac{1}{2}k_2\tau^2 \tag{5.9}$$

as the spatial offset in the argument for the interaction term for the calculation of k_3, for which we get:

$$
\begin{aligned}
k_3 \;=\; f_0 \;&+\; f_r\rho_2 + f_t\tau \\
&+\; \tfrac{1}{2}f_{rr}\rho_2^2 + f_{rt}\rho_2\tau + \tfrac{1}{2}f_{tt}\tau^2 \\
&+\; \tfrac{1}{6}f_{rrr}\rho_2^3 + \tfrac{1}{2}f_{rrt}\rho_2^2\tau + \tfrac{1}{2}f_{rtt}\rho_2\tau^2 + \tfrac{1}{6}f_{ttt}\tau^3 + O(\tau_1^4) \quad (5.10)
\end{aligned}
$$

Now the next step is to substitute the definitions of the ρ_i variables in the k_{i+1} equations, and to keep terms up to the third order in τ.

5.4 The Next Position

Bob: That's going to be quite messy.

Alice: Well, yes, but we have no choice. Let me start with the substitution indicated in Eq. (5.4), where we may as well replace k_1 by its value f_0:

$$\rho_1 \;=\; \tfrac{1}{2}v_0\tau + \tfrac{1}{8}f_0\tau^2 \qquad (5.11)$$

which is just its Taylor series expansion in time, up to second order. Using this substitution in Eq. (5.8), we get:

$$
\begin{aligned}
k_2 \;=\; f_0 \;&+\; \left\{\tfrac{1}{2}v_0f_r + \tfrac{1}{2}f_t\right\}\tau \\
&+\; \left\{\tfrac{1}{8}f_0f_r + \tfrac{1}{8}v_0^2f_{rr} + \tfrac{1}{4}v_0f_{rt} + \tfrac{1}{8}f_{tt}\right\}\tau^2 \\
&+\; \left\{\tfrac{1}{16}v_0f_0f_{rr} + \tfrac{1}{16}f_0f_{rt} + \right. \\
&\quad \left. \tfrac{1}{48}v_0^3f_{rrr} + \tfrac{1}{16}v_0^2f_{rrt} + \tfrac{1}{16}v_0f_{rtt} + \tfrac{1}{48}f_{ttt}\right\}\tau^3 + O(\tau_1^4) \quad (5.12)
\end{aligned}
$$

Similarly, we have to take Eq. (5.10), where we have to use the substitution indicated in Eq. (5.9), which we can rewrite, using the expression for k_2 that we just derived, up till third order in τ, as;

$$\rho_2 \;=\; v_0\tau + \tfrac{1}{2}f_0\tau^2 + \left\{\tfrac{1}{4}v_0f_r + \tfrac{1}{4}f_t\right\}\tau^3 + O(\tau_1^4) \qquad (5.13)$$

Bob: And now I see concretely what you meant, when you said that our previous attempt at proving the fourth-order nature of the Runge-Kutta integrator neglected the treatment of the right-hand side of the differential equation, of the physical interaction terms. They would not have showed up if we were trying to prove the correctness of a second-order Runge-Kutta scheme. It is only at third order in τ that they appear, in the form of partial derivatives of the interaction term.

Alice: Yes, that must be right. Okay, ready for a next round of substitution? Eq. (5.10) then becomes:

$$\begin{aligned}
k_3 \;=\; f_0 \quad &+ \quad \{v_0 f_r + f_t\}\,\tau \\
&+ \quad \left\{\tfrac{1}{2} f_0 f_r + \tfrac{1}{2} v_0^2 f_{rr} + v_0 f_{rt} + \tfrac{1}{2} f_{tt}\right\}\tau^2 \\
&+ \quad \Big\{\tfrac{1}{4} v_0 \left(f_r\right)^2 + \tfrac{1}{4} f_r f_t + \tfrac{1}{2} v_0 f_0 f_{rr} + \tfrac{1}{2} f_0 f_{rt} + \\
&\qquad \tfrac{1}{6} v_0^3 f_{rrr} + \tfrac{1}{2} v_0^2 f_{rrt} + \tfrac{1}{2} v_0 f_{rtt} + \tfrac{1}{6} f_{ttt}\Big\}\tau^3 + O(\tau_1^4) \qquad (5.14)
\end{aligned}$$

We are now in a position to evaluate the solution that our Runge-Kutta scheme gives us for the value of the position and velocity in the next time step, as given in Eq. (5.3), since there we can now substitute k_2 and k_3 in the expressions for r_1 and v_1.

Starting with the position, we find:

$$\begin{aligned}
r_1 \;=\; r_0 \quad &+ \quad v_0\tau + \tfrac{1}{2} f_0 \tau^2 + \tfrac{1}{6}\{v_0 f_r + f_t\}\,\tau^3 \\
&+ \quad \tfrac{1}{24}\left\{f_0 f_r + v_0^2 f_{rr} + 2 v_0 f_{rt} + f_{tt}\right\}\tau^4 + O(\tau_1^5) \qquad (5.15)
\end{aligned}$$

5.5 The Next Velocity

Bob: And the last term in curly brackets, in the first line of the last equation must be the jerk. I begin to see some structure in these expressions.

Alice: Yes. But let me get the velocity as well, and then we can take stock of the whole situation. Let's see. To solve for v_1 in Eq. (5.3), we need to use the following combination of k_i values:

$$\begin{aligned}
4k_2 + k_3 \;=\; 5f_0 \quad &+ \quad \{3 v_0 f_r + 3 f_t\}\,\tau \\
&+ \quad \left\{f_0 f_r + v_0^2 f_{rr} + 2 v_0 f_{rt} + f_{tt}\right\}\tau^2 \\
&+ \quad \Big\{\tfrac{1}{4} v_0 \left(f_r\right)^2 + \tfrac{1}{4} f_r f_t + \tfrac{3}{4} v_0 f_0 f_{rr} + \tfrac{3}{4} f_0 f_{rt} + \\
&\qquad \tfrac{1}{4} v_0^3 f_{rrr} + \tfrac{3}{4} v_0^2 f_{rrt} + \tfrac{3}{4} v_0 f_{rtt} + \tfrac{1}{4} f_{ttt}\Big\}\tau^3 + O(\tau_1^4)
\end{aligned}$$

Plugging this back into the expression for v_1 in Eq. (5.3), we get:

$$\begin{aligned}
v_1 \;=\; v_0 \quad &+ \quad f_0\tau + \tfrac{1}{2}\{v_0 f_r + f_t\}\,\tau^2 \\
&+ \quad \tfrac{1}{6}\left\{f_0 f_r + v_0^2 f_{rr} + 2 v_0 f_{rt} + f_{tt}\right\}\tau^3 \\
&+ \quad \tfrac{1}{24}\Big\{v_0 \left(f_r\right)^2 + f_r f_t + 3 v_0 f_0 f_{rr} + 3 f_0 f_{rt} \\
&\qquad v_0^3 f_{rrr} + 3 v_0^2 f_{rrt} + 3 v_0 f_{rtt} + f_{ttt}\Big\}\tau^4 + O(\tau_1^5) \qquad (5.16)
\end{aligned}$$

Bob: Quite an expression. Well done!

Alice: And an expression that makes sense: we see that indeed the velocity, Eq. (5.16), is the time derivative of the position, Eq. (5.15), up to the order given there. So everything is consistent.

Bob: But this is really astonishing. Why would this last expression be also fourth-order accurate? Even if the fourth-order term had been totally wrong, the velocity in Eq. (5.16) would still have been the time derivative of Eq. (5.15), up to the accuracy to which that expression holds. We had no right to expect more!

Alice: It is quite remarkable. But I presume that whoever derived this equation know what he or she was doing, in choosing the right coefficients! Well, most likely a he, I'm afraid. I think this equation goes back to a time in which there were even fewer women in mathematics than there are now.

Bob: Now that you mention it, I do wonder who derived this marvel. It still surprises me, that you really can get fourth-order accurate, while using only three function evaluations.

Alice: Yes, that is neat. It must have something to do with the fact that we are dealing with a second-order equation. Even though the force term for the acceleration, as the derivative of the velocity, can be arbitrarily complex, the derivative of the position couldn't be simpler: it is always just the velocity.

Bob: I bet you're right. After all, Abramowitz and Stegun list this algorithm specifically under the heading of second-order differential equations. They must have had a reason.

Alice: I'd like to trace the history, now that we've come this far. But first, let's fiish our derivation.

Bob: I thought we had just done that? We've shown that both position and velocity are fourth-order accurate.

Alice: Well, yes, but by now I've grown rather suspicious. We have already seen instances where a derivation looked perfect on paper, but then it turned out to be only formally valid, and not in practice.

Bob: Hard to argue with that!

5.6 Everything Falls into Place

Alice: What we have done now is to compute the predicted positions and velocities after one time step, and we have checked that if we vary the length of the time step, we indeed find that the velocity changes like the time derivative of the position. The question is: are they really solutions of the equations of motion that we started out with, up to the fourth order in the time step size?

Let us go back to the equations of motion, in the form we gave them just after we translated the expressions from Abramowitz and Stegun:

$$a(t) = \ddot{r}(t) \;=\; f(r(t),t) \tag{5.17}$$

For time zero this gives:

$$a_0 = f_0 \tag{5.18}$$

We can differentiate the equation for $a(t)$, to find the jerk:

$$j(t) \;=\; \frac{d}{dt}f(r(t),t) \;=\; f_r\frac{dr}{dt} + f_t \;=\; v(t)f_r + f_t \tag{5.19}$$

which gives us at the starting time:

$$j_0 \;=\; v_0 f_r + f_t \tag{5.20}$$

When we differentiate the equation for $j(t)$, we find the snap:

$$s(t) \;=\; \frac{d}{dt}j(t) \;=\; a(t)f_r + (v(t))^2 f_{rr} + 2v(t)f_{rt} + f_{tt} \tag{5.21}$$

or at time $t = 0$:

$$s_0 \;=\; f_0 f_r + v_0^2 f_{rr} + 2v_0 f_{rt} + f_{tt} \tag{5.22}$$

Next, with one more differentiation of the equation for $s(t)$, we can derive the crackle:

$$\begin{aligned} c(t) \;=\; \frac{d}{dt}s(t) \;&=\; j(t)f_r \;+\; kira4now3v(t)a(t)f_{rr} \;+\; 3a(t)f_{rt} \\ &+\; (v(t))^3 f_{rrr} + 3\,(v(t))^2 f_{rrt} + 3v(r)f_{rtt} + f_{ttt} \end{aligned} \tag{5.23}$$

Using the expression for the jerk, that we found above, we find for time zero:

$$c_0 \;=\; v_0\,(f_r)^2 + f_r f_t + 3v_0 f_0 f_{rr} + 3f_0 f_{rt} + v_0^3 f_{rrr} + 3v_0^2 f_{rrt} + 3v_0 f_{rtt} + f_{ttt} \tag{5.24}$$

Bob: I see, now the clouds are clearing, and the sun is shining through! When you substitute these values back into the equation for the new position, Eq. (5.15), we find:

$$r_1 \;=\; r_0 +\; v_0\tau + \tfrac{1}{2}a_0\tau^2 + \tfrac{1}{6}j_0\tau^3 + \tfrac{1}{24}s_0\tau^4 \tag{5.25}$$

and similarly, for Eq. (5.16), we find:

$$v_1 = v_0 + a_0\tau + \frac{1}{2}j_0\tau^2 + \frac{1}{6}s_0\tau^3 + \frac{1}{24}c_0\tau^4 \qquad (5.26)$$

Both are nothing else but the Taylor series for their orbits.

Alice: And now we have derived them in a fully space-time dependent way.

Bob: Congratulations! I'm certainly happy with this result. But I must admit, I do wonder whether this conclusion would satisfy a mathematician working in numerical analysis.

Alice: Probably not. Actually, I hope not. I'm glad some people are more careful than we are, to make sure that our algorithms really do what we hope they do. At the same time, I'm sure enough now that we have a reasonable understanding of what is going on, and I'm ready to move on.

Bob: I certainly wouldn't ask my students to delve deeper into this matter than we have been doing so far. At some point they will have to get results from their simulations.

Alice: At the same time, unless they have a fair amount of understanding of the algorithms that they rely on, it will be easy for them to use those algorithms in the wrong way, or in circumstances where their use is not appropriate. But I agree, too much of a good thing may simply be too much. Still, if a student would come to be with a burning curiosity to find out more about what is *really* going on with these higher-order integration methods, I would encourage that him or her to get deeper into the matter, through books or papers in numerical analysis.

Bob: I don't think that is too likely to happen, but if such students appear on my doorsteps, I'm happy to send them to you!

Alice: And I would be happy too, since I might well learn from them. I've the feeling that we've only scratched the surface so far.

5.7 Debugging All and Everything?

Bob: Yes, and our scratching had some hits and some misses. If we are ever going to write up all this exploring we are doing now, for the benefit of our students, we have to decide what to tell them, and what to sweep under the rug.

Alice: I suggest we tell them everything, the whole story of how and where we went wrong. If we're going to clean up everything, we will wind up with a text book, and text books there are already enough of in this world.

Bob: I wasn't thinking about any formal publication, good grief, that would be far too much work! I was only thinking about handing out notes to the students in my class. And I must say, I like the idea of showing them a look into our kitchen. I think it will be much more interesting for the students, and after all, if you and I are making these kinds of mistakes, we can't very well pretend that the students would be above making those types of mistakes, can we?

Alice: Precisely. They may learn from our example, in two ways: they will realize that everyone makes mistakes all the time, and that every mistake is a good opportunity to learn more about the situation at hand. Debugging is not only something you have to do with computer programs. When doing analytical work, you also have to debug both the equations you write down, in which you will make mistake, as well as the understanding you bring to the writing down in the first place.

You might even want to go so far as to say that life is one large debugging process. I heard Gerald Sussman saying something like that, some day. He was in a talkative mood, as usual. Somebody asked him what he meant when he said that he considers himself to be a philosophical engineer, and that was his answer.

Bob: I doubt that I will have to debug my beer, I sure hope not! But as far as codes and math and physics, sure, I'd buy that. So yes, let us present the students our temporary failures on the level of math as well as on the level of coding. I'm all for it!

Alice: So, what's next? A sixth order integrator?

Bob: I hadn't thought about that, but given the speed-up we got with going from `rk2` to `rk4`, it would be interesting to see whether this happy process will continue when we would go to a `rk6` method, for a sixth-order Runge-Kutta method. The problem is that Abramowitz and Stegun don't go beyond fourth order methods. Hmm. I'll look around, and see what I can find.

Alice: Meanwhile, I will try and track down the history of our mysterious formulae.

Chapter 6

A Sixth-Order Integrator

6.1 A Multiple leapfrog

Alice: Hi, Bob! To judge from the look on your face, I would bet you found a good recipe for a sixth-order Runge Kutta method.

Bob: As a matter of fact, yes, I did. But before I show it to you, let me know, did you find out the history behind the mysterious formula in Abramowitz and Stegun, that we have been sweating over so much?

Alice: Yes, indeed. It was presented in 1925 by a Finnish mathematician, in a Finnish publication. Here is the full reference: Nystrom, E.J., 1925, Acta Soc. Sci. Fenn., 50, No.13, 1-55. I dug it up in the library.

Bob: I'm impressed! I hope it was not written in Finnish?

Alice: No, it was written in German.

Bob: That wouldn't have helped me.

Alice: Fortunately, I had some German in high school, and I remembered just enough to figure out the gist of the story.

Bob: But how did you find this article?

Alice: I first went to some text books on numerical methods. After some searching, I learned that our mystery formula is an example of what are called Runge-Kutta-Nystrom algorithms. It was Nystrom who made the extension from normal Runge-Kutta methods to special treatments for second-order differential equations. And there is a whole lot more to the history. In recent years, quite a number of interesting books have appeared that have a bearing on the general topic. I found it hard to put them down.

Bob: Now that you got bitten by the bug, perhaps you'll be deriving new methods for us, from scratch?

Alice: It's tempting to do so, just to see how it's done. Like having a peak into the kitchen, rather than ordering from a menu. But not now.

Bob: No, let me show you first what I found on my menu, a Japanese menu it turned out, which featured a sixth-order Runge-Kutta method prepared by an astronomer working at the National Observatory in Tokyo, Haruo Yoshida.

Alice: Let me guess how long the code will be. The `rk2` method was five lines long, and the `rk4` method was eight lines long. I seem to remember that the complexity of Runge-Kutta methods increases rapidly with order, so the new code must at least be twice as long, more likely a bit more. Okay, I'll bet on twenty lines.

Bob: How much are you willing to bet?

Alice: I'm not much of a gambler. Let's say you'll get my admiration if you did it in less than twenty lines.

Bob: How much admiration will I get if it would be less than ten lines?

Alice: I don't know, since I wouldn't believe you.

Bob: It that case you must be ready for a really wild move. How much are you willing to bet against my having written a sixth-order Runge-Kutta code in five lines?

Alice: Hahaha! I will bet a good laugh. You can't be serious.

Bob: I am.

Alice: Huh? You're not going to tell me that your sixth-order Runge-Kutta is as long as the second-order Runge-Kutta?

Bob: Seeing is believing. Here it is, in the file `yo6body.rb`

```
def yo6(dt)
  d = [0.784513610477560e0, 0.235573213359357e0, -1.17767998417887e0,
       1.31518632068391e0]
  for i in 0..2 do leapfrog(dt*d[i]) end
  leapfrog(dt*d[3])
  for i in 0..2 do leapfrog(dt*d[2-i]) end
end
```

Alice: Seven leapfrog calls? And that does it?

Bob: Yes, that does it, or so is the claim. It was invented by Yoshida, a Japanese astronomer, around 1990, that's why I called it `yo6` instead of `rk6`. I haven't gotten the time to test it out yet. I just finished it when you came in.

6.2 Great!

Alice: I'm not sure whether to call this incredible, shocking, or what. But before I look for the right adjective, let me first see that it does what you claim it does.

Bob: I'm curious too. Let us take our good old Kepler orbit for a spin: ten time units, as before. We will start with the fourth-order Runge-Kutta method:

```
|gravity> ruby integrator_driver2g.rb < euler.in
dt = 0.001
dt_dia = 10
dt_out = 10
dt_end = 10
method = rk4
at time t = 0, after 0 steps :
  E_kin = 0.125 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 0
  (E_tot - E_init) / E_init =-0
at time t = 10, after 10000 steps :
  E_kin = 0.554 , E_pot =  -1.43 , E_tot = -0.875
             E_tot - E_init = -2.46e-09
  (E_tot - E_init) / E_init =2.81e-09
  1.0000000000000000e+00
  5.9961758437074986e-01 -3.6063455639926667e-01
  1.0308068733946525e+00  2.1389536225475009e-01
```

for which we found this reasonable result, with a relative accuracy in the conservation of the total energy of a few in a billion. Presumably the sixth-order method should do significantly better, for the same time step value:

```
|gravity> ruby integrator_driver3a.rb < euler.in
dt = 0.001
dt_dia = 10
dt_out = 10
dt_end = 10
method = yo6
at time t = 0, after 0 steps :
  E_kin = 0.125 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 0
  (E_tot - E_init) / E_init =-0
at time t = 10, after 10000 steps :
  E_kin = 0.554 , E_pot =  -1.43 , E_tot = -0.875
             E_tot - E_init = 1.72e-14
```

```
  (E_tot - E_init) / E_init =-1.97e-14
  1.0000000000000000e+00
  5.9961755487188750e-01 -3.6063458346955279e-01
  1.0308069102782800e+00  2.1389530415211538e-01
```

6.3 Too Good?

Alice: Significant, all right: almost machine accuracy. Can you give a larger time step? Make it ten times as large! That may be way too large, but we can always make it smaller again.

Bob: My pleasure!

```
|gravity> ruby integrator_driver3b.rb < euler.in
dt = 0.01
dt_dia = 10
dt_out = 10
dt_end = 10
method = yo6
at time t = 0, after 0 steps :
  E_kin = 0.125 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 0
  (E_tot - E_init) / E_init =-0
at time t = 10, after 1000 steps :
  E_kin = 0.554 , E_pot =  -1.43 , E_tot = -0.875
             E_tot - E_init = 1.23e-14
  (E_tot - E_init) / E_init =-1.41e-14
  1.0000000000000000e+00
  5.9960497793690160e-01 -3.6065834429401844e-01
  1.0308122043933747e+00  2.1385575804694398e-01
```

Alice: No! This is too good to be true. Still essentially machine accuracy, for only a thousand steps, while `rk2` with ten-thousand steps did far worse, by a whopping factor of a hundred thousand. Well, let's really push it: can you make the time step yet again ten times larger?

Bob: Going around a rather eccentric Kepler orbits a few times, with only a hundred steps in total? Well, we can always try.

```
|gravity> ruby integrator_driver3c.rb < euler.in
dt = 0.1
dt_dia = 10
dt_out = 10
```

```
dt_end = 10
method = yo6
at time t = 0, after 0 steps :
  E_kin = 0.125 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 0
  (E_tot - E_init) / E_init =-0
at time t = 10, after 100 steps :
  E_kin = 0.125 , E_pot =  -0.605 , E_tot = -0.481
             E_tot - E_init = 0.394
  (E_tot - E_init) / E_init =-0.451
  1.0000000000000000e+00
  1.0619008919577413e+00 -1.2659613458920589e+00
 -2.3237432724190762e-02  4.9855659376135231e-01
```

6.4 Spooky

Alice: All good things come to an end. We finally found the limit of your new code, but only barely: while the error is of order unity, the system hasn't really exploded numerically yet.

Bob: Let's change time step values a bit more carefully. How about something in between 0.1 and 0.01. I'll take 0.04:

```
|gravity> ruby integrator_driver3d.rb < euler.in
dt = 0.04
dt_dia = 10
dt_out = 10
dt_end = 10
method = yo6
at time t = 0, after 0 steps :
  E_kin = 0.125 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 0
  (E_tot - E_init) / E_init =-0
at time t = 10, after 250 steps :
  E_kin = 0.513 , E_pot =  -1.39 , E_tot = -0.879
             E_tot - E_init = -0.00351
  (E_tot - E_init) / E_init =0.00402
  1.0000000000000000e+00
  5.8605062658932228e-01 -4.1556477918009915e-01
  1.0033546518756853e+00  1.4169619805243588e-01
```

Alice: Ah, good, less than a percent error in relative total energy conservations. We're getting somewhere in our testing. Let's halve the time step. If your `yo6` method is really sixth-order, halving the time step should make the result 64 times more accurate.

Bob: Here we go:

```
|gravity> ruby integrator_driver3e.rb < euler.in
dt = 0.02
dt_dia = 10
dt_out = 10
dt_end = 10
method = yo6
at time t = 0, after 0 steps :
  E_kin = 0.125 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 0
  (E_tot - E_init) / E_init =-0
at time t = 10, after 500 steps :
  E_kin = 0.554 , E_pot =  -1.43 , E_tot = -0.875
             E_tot - E_init = -1.49e-07
  (E_tot - E_init) / E_init =1.7e-07
  1.0000000000000000e+00
  5.9887919973409587e-01 -3.6203156818146032e-01
  1.0311098923820705e+00  2.1157132982705190e-01
```

Alice: Hmm, that's much much more improvement than a factor 64. I guess the run with a time step of 0.04 was not yet in the linear regime.

Bob: I'll halve the time step again, to 0.01:

```
|gravity> ruby integrator_driver3b.rb < euler.in
dt = 0.01
dt_dia = 10
dt_out = 10
dt_end = 10
method = yo6
at time t = 0, after 0 steps :
  E_kin = 0.125 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 0
  (E_tot - E_init) / E_init =-0
at time t = 10, after 1000 steps :
  E_kin = 0.554 , E_pot =  -1.43 , E_tot = -0.875
             E_tot - E_init = 1.23e-14
  (E_tot - E_init) / E_init =-1.41e-14
  1.0000000000000000e+00
  5.9960497793690160e-01 -3.6065834429401844e-01
  1.0308122043933747e+00  2.1385575804694398e-01
```

Alice: Again an improvement of far far more than a factor of 64. What is going on?

Bob: This is almost spooky. The integrator has no right to converge *that* fast.

6.5 An Orbit Segment

Alice: The problem may be that we follow a particle through pericenter with enormously long time steps, compared with the pericenter passage time. Who knows how all the local errors accumulate and/or cancel out. It may be better to take only a small orbit segment. If we integrate for less than one time unit, we know for sure that the two bodies have not reached each other at periastron, according to what we saw before. Remember, we start our integration at apastron.

Bob: I'm happy to try that. Let us take a total integration time of 0.2, and compare a few large time step choices. Unless we make those steps pretty large, we will just get machine accuracy back, and we won't learn anything. Let's take a time step of 0.1 first:

```
|gravity> ruby integrator_driver3f.rb < euler.in
dt = 0.1
dt_dia = 0.2
dt_out = 0.2
dt_end = 0.2
method = yo6
at time t = 0, after 0 steps :
  E_kin = 0.125 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 0
  (E_tot - E_init) / E_init =-0
at time t = 0.2, after 2 steps :
  E_kin = 0.14 , E_pot =  -1.02 , E_tot = -0.875
             E_tot - E_init = 4.58e-11
  (E_tot - E_init) / E_init =-5.24e-11
  1.0000000000000000e+00
  9.7991592024615404e-01  9.9325553458239929e-02
 -2.0168916126858463e-01  4.8980438271673599e-01
```

And then twice as long a time step, of 0.2. Perhaps we will finally get a factor of 64 in degradation of energy conservation:

```
|gravity> ruby integrator_driver3g.rb < euler.in
dt = 0.2
dt_dia = 0.2
dt_out = 0.2
dt_end = 0.2
method = yo6
at time t = 0, after 0 steps :
  E_kin = 0.125 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 0
  (E_tot - E_init) / E_init =-0
at time t = 0.2, after 1 steps :
  E_kin = 0.14 , E_pot =  -1.02 , E_tot = -0.875
             E_tot - E_init = 2.16e-09
  (E_tot - E_init) / E_init =-2.46e-09
  1.0000000000000000e+00
  9.7991596638987577e-01  9.9325498370889442e-02
 -2.0168893933388904e-01  4.8980439348592314e-01
```

Alice: Much better this time. A factor of 47, in between 32 and 64. This by itself would suggest that your new method is somewhere in between fifth order and sixth order in accuracy.

Bob: And most likely sixth order: the ratio of 64/47=1.4 is smaller than the ratio of 47/32=1.5, so we are logarithmically closer to 64 than to 32.

6.6 Victory

Alice: However, we should be able to settle this more unequivocally. The problem is, we don't have much wiggle room: we don't want the total time to become so long that the particles reach pericenter, yet when we make the total integration time much shorter, we don't have room left for even a single time step – unless we invite machine accuracy again.

Bob: Of course, we can look for a different type of orbit, but then we have to start our analysis all over again. How about this: let's take a total time of 0.5 units, while giving our particles the choice to cross this sea of time in either four or five time steps. We can start with a time step of 0.1:

```
|gravity> ruby integrator_driver3h.rb < euler.in
dt = 0.1
dt_dia = 0.5
dt_out = 0.5
dt_end = 0.5
method = yo6
```

```
at time t = 0, after 0 steps :
  E_kin = 0.125 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 0
  (E_tot - E_init) / E_init =-0
at time t = 0.5, after 5 steps :
  E_kin = 0.232 , E_pot =  -1.11 , E_tot = -0.875
             E_tot - E_init = 9.08e-10
  (E_tot - E_init) / E_init =-1.04e-09
  1.0000000000000000e+00
  8.7155094516550113e-01  2.3875959971050609e-01
 -5.2842606676242798e-01  4.2892868844542126e-01
```

And then repeat the procedure for a time step of 0.125.

Alice: Ah, before you start the run, let's make a prediction: the energy conservation degradation should be $(5/4)^6 = 3.81$.

```
|gravity> ruby integrator_driver3i.rb < euler.in
dt = 0.125
dt_dia = 0.5
dt_out = 0.5
dt_end = 0.5
method = yo6
at time t = 0, after 0 steps :
  E_kin = 0.125 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 0
  (E_tot - E_init) / E_init =-0
at time t = 0.5, after 4 steps :
  E_kin = 0.232 , E_pot =  -1.11 , E_tot = -0.875
             E_tot - E_init = 3.35e-09
  (E_tot - E_init) / E_init =-3.83e-09
  1.0000000000000000e+00
  8.7155095947304040e-01  2.3875959630280436e-01
 -5.2842603945420896e-01  4.2892869095118885e-01
```

Bob: And by dividing the total energy errors of the last two runs we get a ratio of 3.69.

Alice: Not bad at all! Just three percent short of what we had expected for a sixth-order integration scheme. I think we can now officially declare your `yo6` to be worthy of the `6` in it's name!

But I must say, I'm still deeply puzzled and surprised that seven leapfrog calls, involving four rather funny numbers, can suddenly emulate a Runge-Kutta.

Bob: According to my source, it is more than 'emulate': this particular combination of leapfrog calls *is* nothing else but a Runge-Kutta scheme.

Alice: I want to know more about this. What did you find out about the theory behind this curious procedure?

Chapter 7

Yoshida's Algorithms

7.1 Recall Baker, Campbell and Hausdorff

Bob: Well, Alice, I tried my best to learn more about Yoshida's algorithms.

Alice: Tell me!

Bob: The good news is: I learned a lot more, and I even managed to construct an eighth-order integrator, following his recipes. I also wrote a fourth-order version, which is a lot simpler, and easier to understand intuitively.

Alice: A lot of good news indeed. What's the bad news?

Bob: I'm afraid I couldn't really follow the theory behind his ideas, even though I was quite curious to see where these marvelous recipes came from.

I went back to the original article in which he presents his ideas, *Construction of higher order symplectic integrators*, by Haruo Yoshida, 1990, Phys. Lett. A **150**, 262. The problem for me was that he starts off in the first paragraph talking about symplectic two-forms, whatever they may be, and then launches into a discussion of non-commutative operators, Poisson brackets, and so on. It all has to do with Lie groups, it seems, something I don't know anything about.

To give you an idea, the first sentence in his section 3, basic formulas, starts with "First we recall the Baker-Campbell-Hausdorff formula". But since I have never heard of this formula, I couldn't even begin to recall it.

Alice: I remember the BCH formula! I came across it when I learned about path integrals in quantum field theory. It was an essential tool for composing propagators. And in Yoshida's case, he is adding a series of leapfrog mappings together, in order to get one higher-order mapping. Yes, I can see the analogy. The flow of the evolution of a dynamical system can be modeled as driven by a Lie group, for which the Lie algebra is non-commutative. Now with the BCH formula . . .

Bob: Alright, I'm glad it makes sense to you, and maybe some day we can sit down and teach me the theory of Lie groups. But today, let's continue our work toward getting an N-body simulation going. We haven't gotten further than the 2-body problem yet. I'll listen to the stories by Mr. Lie some other day, later.

Alice: I expect that these concepts will come in naturally when we start working on Kepler regularization, which is a problem we'll have to face sooner or later, when we start working with the dynamics of a thousand point masses, and we encounter frequent close encounters in tight double stars.

Bob: There, the trick seems to be to map the three-dimensional Kepler problem onto the four-dimensional harmonic oscillator. I've never heard any mention of Lie or the BCH formula in that context.

Alice: We'll see. I expect that when we have build a good laboratory, equipped with the right tools to do a detailed exploration, we will find new ways to treat perturbed motion in locally interacting small-N systems. I would be very surprised if those explorations wouldn't get us naturally involved in symplectic methods and Lie group applications.

7.2 An Eighth-Order Integrator

Bob: We'll see indeed. At the end of Yoshida's paper, at least his techniques get more understandable for me: he solves rather complicated algebraic equations in order to get the coefficients for various integration schemes. What I implemented for the sixth order integrator before turns out to be based on just one set of his coefficients, in what I called the `d` array, but there are two other sets as well, which seem to be equally good.

What is more, he gives no less than *seven* different sets of coefficients for his eighth-order scheme! I had no idea which of those seven to choose, so I just took the first one listed. Here is my implementation, in the file `yo8body.rb`

```
def yo8(dt)
  d = [0.104242620869991e1, 0.182020630970714e1, 0.157739928123617e0,
       0.244002732616735e1, -0.716989419708120e-2, -0.244699182370524e1,
       -0.161582374150097e1, -0.17808286265894516e1]
  for i in 0..6 do leapfrog(dt*d[i]) end
  leapfrog(dt*d[7])
  for i in 0..6 do leapfrog(dt*d[6-i]) end
end
```

For comparison, let me repeat a run from our sixth-order experimentation, with the same set of parameters. We saw earlier:

```
|gravity> ruby integrator_driver3h.rb < euler.in
dt = 0.1
dt_dia = 0.5
dt_out = 0.5
dt_end = 0.5
method = yo6
at time t = 0, after 0 steps :
  E_kin = 0.125 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 0
  (E_tot - E_init) / E_init =-0
at time t = 0.5, after 5 steps :
  E_kin = 0.232 , E_pot =  -1.11 , E_tot = -0.875
             E_tot - E_init = 9.08e-10
  (E_tot - E_init) / E_init =-1.04e-09
  1.0000000000000000e+00
  8.7155094516550113e-01  2.3875959971050609e-01
 -5.2842606676242798e-01  4.2892868844542126e-01
```

My new eighth-order method yo8 gives:

```
|gravity> ruby integrator_driver3j.rb < euler.in
dt = 0.1
dt_dia = 0.5
dt_out = 0.5
dt_end = 0.5
method = yo8
at time t = 0, after 0 steps :
  E_kin = 0.125 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 0
  (E_tot - E_init) / E_init =-0
at time t = 0.5, after 5 steps :
  E_kin = 0.232 , E_pot =  -1.11 , E_tot = -0.875
             E_tot - E_init = 4.2e-05
  (E_tot - E_init) / E_init =-4.8e-05
  1.0000000000000000e+00
  8.7156845267947847e-01  2.3879462060443227e-01
 -5.2848151560751322e-01  4.2888364744600843e-01
```

Significantly worse for the same time step than the sixth order case, but of course there no *a priori* reason for it to be better or worse for any particular choice of parameters. The point is that it should get rapidly better when we go to smaller time steps. And it does!

Here, let me make the time step somewhat smaller, and similarly the integration time, so that we still do five integration steps. With a little bit of luck, that will bring us in a regime where the error scaling will behave the way it should. This last error may still be too large to make meaningful comparisons

```
|gravity> ruby integrator_driver3k.rb < euler.in
dt = 0.04
dt_dia = 0.2
dt_out = 0.2
dt_end = 0.2
method = yo8
at time t = 0, after 0 steps :
  E_kin = 0.125 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 0
  (E_tot - E_init) / E_init =-0
at time t = 0.2, after 5 steps :
  E_kin = 0.14 , E_pot =  -1.02 , E_tot = -0.875
             E_tot - E_init = 7.5e-10
  (E_tot - E_init) / E_init =-8.58e-10
  1.0000000000000000e+00
  9.7991592001699501e-01  9.9325555445578834e-02
 -2.0168916703866913e-01  4.8980438183737618e-01
```

7.3 Seeing is Believing

Alice: How about halving the time step? That should make the error 256 times smaller, if the integrator is indeed eighth-order accurate.

Bob: Here you are:

```
|gravity> ruby integrator_driver3l.rb < euler.in
dt = 0.02
dt_dia = 0.2
dt_out = 0.2
dt_end = 0.2
method = yo8
at time t = 0, after 0 steps :
  E_kin = 0.125 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 0
  (E_tot - E_init) / E_init =-0
at time t = 0.2, after 10 steps :
  E_kin = 0.14 , E_pot =  -1.02 , E_tot = -0.875
             E_tot - E_init = 2.82e-12
  (E_tot - E_init) / E_init =-3.22e-12
  1.0000000000000000e+00
```

```
   9.7991591952094304e-01  9.9325554314944414e-02
  -2.0168916469198325e-01  4.8980438255589787e-01
```

Not bad, I would say. I can give you another factor two shrinkage in time step, before we run out of digits in machine accuracy:

```
 |gravity> ruby integrator_driver3m.rb < euler.in
 dt = 0.01
 dt_dia = 0.2
 dt_out = 0.2
 dt_end = 0.2
 method = yo8
 at time t = 0, after 0 steps :
   E_kin = 0.125 , E_pot =  -1 , E_tot = -0.875
              E_tot - E_init = 0
   (E_tot - E_init) / E_init =-0
 at time t = 0.2, after 20 steps :
   E_kin = 0.14 , E_pot =  -1.02 , E_tot = -0.875
              E_tot - E_init = 9.77e-15
   (E_tot - E_init) / E_init =-1.12e-14
   1.0000000000000000e+00
   9.7991591951908552e-01  9.9325554310707803e-02
  -2.0168916468313569e-01  4.8980438255859476e-01
```

Alice: Again close to a factor 256 better, as behooves a proper eighth-order integrator. Good! I believe the number 8 in yo8.

7.4 The Basic Idea

Bob: Now, even though I did not follow the abstract details of Yoshida's paper, I did get the basic idea, I think. It helped to find his recipe for a fourth-order integrator. I implemented that one as well. Here it is:

```
  def yo4(dt)
    d = [1.351207191959657, -1.702414383919315]
    leapfrog(dt*d[0])
    leapfrog(dt*d[1])
    leapfrog(dt*d[0])
  end
```

Alice: Only three leapfrog calls this time.

Bob: Yes, the rule seems to be that a for an $(2k)^{th}$ order Yoshida integrator, you need to combine $2^k - 1$ leapfrog leaps to make one Yoshida leap, at least up to eighth-order, which is how far Yoshida's paper went. You can check the numbers for $k = 2, 3, 4$: three leaps for the 4th-order scheme, seven for the sixth-order scheme, and fifteen for the 8th-order scheme.

Alice: Not only that, it even works for $k = 0, 1$: a second-order integrator uses exactly one leapfrog, and a zeroth-order integrator by definition does not do anything, so it makes zero leaps.

Bob: You always like to generalize, don't you! But you're right, the expression works for $k = 0 \mathrm{through} 4$ alright.

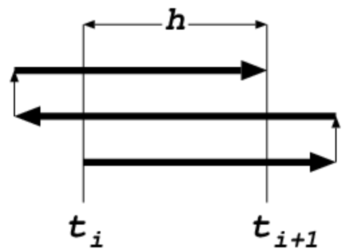
Alice: And the fourth-order is indeed the first order for which the leaps of the leapfrog are composed into a larger dance.

Bob: Perhaps we should call Yoshida's algorithm the leapdance scheme.

Alice: Or the dancefrog? Now, I would prefer the dancecat. you see, cats are more likely to dance than frogs do. And while a cat is trying to catch a frog, she may look like dancing while trying to follow the frog.

Bob: Do cats eat frogs? I thought they stuck to mammals and birds.

Alice: I've never seen a cat eating a frog, but I bet that they like to chase anything that moves; surely my cat does. Anyway, let's make a picture of the fourth-order dance:



This is what your `yo4` is doing, right? Starting at the bottom, at time t_i, you jump forward a little further than the time step h would ask you to do for a single leapfrog. Then you jump backward to such an extent that you have to jump forward again, over the same distance as you jumped originally, in order to reach the desired time at the end of the time step: $t = t + h = t_{i+1}$.

Bob: Yes, this is precisely what happens. And the length of the first and the third time step can be calculated analytically, according to Yoshida, a result that he ascribes to an earlier paper by Neri, in 1988. In units of what you called h, it is the expression in the first coefficient in the `d` array in `yo4`:

$$\mathtt{d}[0] = \frac{1}{2 - 2^{1/3}} \tag{7.1}$$

And since the three-leap dance has to land at a distance one into the future, in units of the time step, the middle leap backward has to have a length of:

$$\mathtt{d}[1] = 1 - 2\mathtt{d}[0] = -\frac{2^{1/3}}{2 - 2^{1/3}} \tag{7.2}$$

Somehow, the third-order and fourth-order errors generated by each of the leaping frogs are canceled by this precise fine tuning of step lengths.

7.5 Testing the Wrong Scheme

Alice: That will be easy to check. How about choosing somewhat different values. Let's take a round number, $d[0] = 1.5$, which forces the other number to be $d[1] = -2$. If there is a matter of fine tuning involved, these wrong values should give only second-order behavior, since a random combination of three second-order integrator steps should still scale as a second-order combination.

Bob: Good idea. Let me call the file `yo8body_wrong.rb` to make sure we later don't get confused about which is which. I will leave the correct methods `yo4`, `yo6`, and `yo8` all in the file `yo8body.rb`. Here is your suggestion for the wrong version:

```
  def yo4(dt)

#    d = [1.351207191959657, -1.702414383919315]
    d = [1.5, -2]            # WRONG values, just for testing!
    leapfrog(dt*d[0])
    leapfrog(dt*d[1])
    leapfrog(dt*d[0])
  end
```

Let's first take a previous run with the fourth-order Runge-Kutta `rk4` method, to have a comparison run:

```
|gravity> ruby integrator_driver2i.rb < euler.in
dt = 0.1
dt_dia = 0.1
dt_out = 0.1
dt_end = 0.1
method = rk4
at time t = 0, after 0 steps :
  E_kin = 0.125 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 0
```

```
  (E_tot - E_init) / E_init =-0
at time t = 0.1, after 1 steps :
  E_kin = 0.129 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 1.75e-08
  (E_tot - E_init) / E_init =-2.01e-08
  1.0000000000000000e+00
  9.9499478923153439e-01  4.9916431937376750e-02
 -1.0020915515250550e-01  4.9748795077019681e-01
```

Here is our faulty `yo4` result, for the same parameters:

```
|gravity> ruby integrator_driver3n.rb < euler.in
dt = 0.1
dt_dia = 0.1
dt_out = 0.1
dt_end = 0.1
method = yo4
at time t = 0, after 0 steps :
  E_kin = 0.125 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 0
  (E_tot - E_init) / E_init =-0
at time t = 0.1, after 1 steps :
  E_kin = 0.129 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = -1.45e-05
  (E_tot - E_init) / E_init =1.66e-05
  1.0000000000000000e+00
  9.9498863274056060e-01  4.9808366540742943e-02
 -1.0007862035430568e-01  4.9750844492670115e-01
```

and here for a ten times smaller time step:

```
|gravity> ruby integrator_driver3o.rb < euler.in
dt = 0.01
dt_dia = 0.1
dt_out = 0.1
dt_end = 0.1
method = yo4
at time t = 0, after 0 steps :
  E_kin = 0.125 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 0
  (E_tot - E_init) / E_init =-0
at time t = 0.1, after 10 steps :
  E_kin = 0.129 , E_pot =  -1 , E_tot = -0.875
```

```
           E_tot - E_init = -1.48e-07
 (E_tot - E_init) / E_init =1.7e-07
 1.0000000000000000e+00
 9.9499471442505816e-01  4.9915380903880313e-02
-1.0020772164145644e-01  4.9748816373440058e-01
```

Alice: And it is clearly second-order. We can safely conclude that a random choice of three leapfrog leaps doesn't help us much. Now how about a well-orchestrated dance of three leaps, according to Neri's algorithm?

7.6 Testing the Right Scheme

Bob: That should be better. Let's make really sure I have the correct version back again:

```
def yo4(dt)
  d = [1.351207191959657, -1.702414383919315]
  leapfrog(dt*d[0])
  leapfrog(dt*d[1])
  leapfrog(dt*d[0])
end
```

```
|gravity> ruby integrator_driver3p.rb < euler.in
dt = 0.1
dt_dia = 0.1
dt_out = 0.1
dt_end = 0.1
method = yo4
at time t = 0, after 0 steps :
  E_kin = 0.125 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 0
  (E_tot - E_init) / E_init =-0
at time t = 0.1, after 1 steps :
  E_kin = 0.129 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 9.16e-08
  (E_tot - E_init) / E_init =-1.05e-07
  1.0000000000000000e+00
  9.9499490507620858e-01  4.9915249744859044e-02
 -1.0020899341473008e-01  4.9748801781965912e-01
```

Less accurate than the rk4 method, but the important thing is the scaling. Here goes, for a time step shorter by a factor of ten:

```
|gravity> ruby integrator_driver3q.rb < euler.in
dt = 0.01
dt_dia = 0.1
dt_out = 0.1
dt_end = 0.1
method = yo4
at time t = 0, after 0 steps :
  E_kin = 0.125 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 0
  (E_tot - E_init) / E_init =-0
at time t = 0.1, after 10 steps :
  E_kin = 0.129 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 9.16e-12
  (E_tot - E_init) / E_init =-1.05e-11
  1.0000000000000000e+00
  9.9499478010211795e-01  4.9916426099720732e-02
 -1.0020902859703379e-01  4.9748796006619145e-01
```

Perfect! Yoshida was right, once again.

Alice: Indeed, very nicely fourth order. Great!

Chapter 8

A Two-Step Method

8.1 A Matter of Memory

Bob: So far we have used only one-step methods: in each case we start with the position and velocity at one point in time, in order to calculate the position and velocity at the next time step. The higher-order schemes jump around a bit, to in-between times in case of the traditional Runge-Kutta algorithms, or slightly before or beyond the one-step interval in case of Yoshida's algorithms. But even so, all of our schemes have been self-starting.

As an alternative to jumping around, you can also remember the results from a few earlier steps. Fitting a polynomial to previous interaction calculations will allow you to calculate higher time derivatives for the orbit . . .

Alice: . . . wonderful! Applause!

Bob: Huh?

Alice: You said it just right, using both the term *orbit* and *interaction* correctly.

Bob: What did I say?

Alice: You made the correct distinction between the physical interactions on the right-hand side of the equations of motion, which we agreed to call the interaction side, and the mathematical description of the orbit characteristics on the left-hand side of the equations of motion, which we decided to call the orbit side.

Bob: I guess your lessons were starting to sink in. In any case, let me put my equations where my mouth is, and let show the idea first for a second-order multi-step scheme, a two-step scheme in fact. We start with the *orbit* part, where we expand the acceleration in a Taylor series with just one extra term:

$$\mathbf{a}(t) \;=\; \mathbf{a}_0 + \mathbf{j}_0 t + O(t^2) \tag{8.1}$$

Alice: Our old friend, the jerk, evaluated also at time $t = 0$.

Bob: Speak for yourself, my friends are not jerks. We can determine the jerk at the beginning of our new time step if we can remember the value of the acceleration at the beginning of the previous time step, at time $t = -\Delta t$, as follows:

$$\mathbf{a}_{-1} \;=\; \mathbf{a}_0 - \mathbf{j}_0 \Delta t \quad \Rightarrow \tag{8.2}$$

$$\mathbf{j}_0 \;=\; \frac{1}{\Delta t}\,\{\mathbf{a}_0 - \mathbf{a}_{-1}\} \tag{8.3}$$

With this information, we can use the same approach as we did with the forward Euler algorithm, but we can go one order higher in the Taylor series expansion for the position and the velocity. The forward Euler method gave us:

$$\begin{aligned} \mathbf{r}_1 &= \mathbf{r}_0 + \mathbf{v}_0 \Delta t \\ \mathbf{v}_1 &= \mathbf{v}_0 + \mathbf{a}_0 \Delta t \end{aligned}$$

but now we can write:

$$\begin{aligned} \mathbf{r}_1 &= \mathbf{r}_0 + \mathbf{v}_0 \Delta t + \tfrac{1}{2}\mathbf{a}_0 (\Delta t)^2 \\ \mathbf{v}_1 &= \mathbf{v}_0 + \mathbf{a}_0 \Delta t + \tfrac{1}{2}\mathbf{j}_0 (\Delta t)^2 \end{aligned} \tag{8.4}$$

and we can use the value for the jerk given in Eq. (8.3), without any need for half-steps, such as in the leapfrog or Runga-Kutta methods.

A more direct, but less easily understandable way of writing these equations is to substitute Eq. (8.3) into Eq. (8.4), in order to get an equation written purely in terms of the accelerations at different times:

$$\begin{aligned} \mathbf{r}_1 &= \mathbf{r}_0 + \mathbf{v}_0 \Delta t + \tfrac{1}{2}\mathbf{a}_0 (\Delta t)^2 \\ \mathbf{v}_1 &= \mathbf{v}_0 + \tfrac{1}{2}\,(3\mathbf{a}_0 - \mathbf{a}_1)\,\Delta t \end{aligned} \tag{8.5}$$

I have seen this expression before, but I did not realize then that it had such a simple explanation in terms of the jerk.

8.2 Implementation

Alice: That is an interesting twist, and indeed a very different approach.

Bob: I have implemented this, starting from the file `yo8body.rb`, and renaming it `ms2body.rb`. I had to make one modification, though. Previously, we kept track of the number of time steps with a counter `nsteps` that was a local variable within the method `evolve`, but now we need that information within the multi-step integrator, which I called `ms2`.

Alice: Why's that?

Bob: A multi-step method is not self-starting, since at the beginning of the integration, there is no information about previous steps, simply because there are no previous steps. I solve this by borrowing another 2nd-order method, `rk2` for the first time step. But this means that `ms2` has to know whether we are at the beginning of the integration or not, and the easiest way to get that information there is to make the number of time steps into an instance variable within the `Body` class.

What I did was replace `nsteps` by `@nsteps`, in the two places where it was used within the method `evolve`. I made the same change in the method `write_diagnostics`, which simplified matters, since before we needed to pass `nsteps` as an argument to that method, and now this is no longer necessary.

Here is the code:

```
def ms2(dt)
  if @nsteps == 0
    @prev_acc = acc
    rk2(dt)
  else
    old_acc = acc
    jdt = old_acc - @prev_acc
    @pos += vel*dt + old_acc*0.5*dt*dt
    @vel += old_acc*dt + jdt*0.5*dt
    @prev_acc = old_acc
  end
end
```

Alice: So the variable `@prev_acc` serves as your memory, to store the previous acceleration. At the start, when `@nsteps` is still zero, you initialize `@prev_acc`, and during each next step, you update that variable at the end, so that it always contains the value of the acceleration at the *start* of the previous step.

Bob: Yes. The acceleration at the *end* of the previous step would be the same as the acceleration at the start of the current step, and that value is stored in the local variable `old_acc`, as you can see. This allows me to calculate the jerk.

Alice: Or more precisely, the product of jerk $\mathbf{j}$ and time step Δt, in the form of $\mathbf{j}\Delta t$, since it is only that combination that appears in Eq. (8.4), which you write in the next two lines of code. Okay, that is all clear!

Bob: When I first wrote this, I was wondering whether it was correct to use this expression for the jerk, since strictly speaking, it gives an approximation for the value of the jerk that is most accurate at a time that is half a time step in the past, before the beginning of the current time step:

$$\mathbf{j}_{-1/2} = \mathbf{a}_0 - \mathbf{a}_{-1}$$

But then I realized that the difference does not matter, for a second order integration scheme. In terms of the next time derivative of position, the snap $\mathbf{s} = d\mathbf{j}/dt$, the leading term of the difference would be:

$$\mathbf{j}_0 = \mathbf{j}_{-1/2} + \tfrac{1}{2}\mathbf{c}_0\Delta t$$

All this would do is to add a term of order $(\Delta t)^3$ to the last line in Eq. (8.4), beyond the purview of a second-order scheme.

Alice: Yes, from the point of view of a second-order integrator, the jerk is simply constant, and we can evaluate it at whatever point in time we want.

Bob: And before I forget, there is one more change I made, in the file `vector.rb`, where I gave an extra method to our `Vector` class:

```
def -(a)
  diff = Vector.new
  self.each_index{|k| diff[k] = self[k]-a[k]}
  diff
end
```

Alice: Ah, yes, we only had addition and multiplication methods before, but when you compute the jerk in `ms2` it is most natural to use subtraction.

Bob: And for good measure, I added a division method as well. Note that `raise` is a standard way for Ruby to report an error condition. Here it is:

```
def /(a)
  if a.class == Vector
    raise
  else
    quotient = Vector.new           # scalar quotient
    self.each_index{|k| quotient[k] = self[k]/a}
  end
  quotient
end
```

Alice: I'm sure that will come in handy sooner or later.

8.3 Testing

Bob: Let me show that the two-step integration method works. I'll start again with a fourth-order Runge-Kutta result, as a check:

```
|gravity> ruby integrator_driver2j.rb < euler.in
dt = 0.01
dt_dia = 0.1
dt_out = 0.1
dt_end = 0.1
method = rk4
at time t = 0, after 0 steps :
  E_kin = 0.125 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 0
  (E_tot - E_init) / E_init =-0
at time t = 0.1, after 10 steps :
  E_kin = 0.129 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 1.79e-12
  (E_tot - E_init) / E_init =-2.04e-12
  1.0000000000000000e+00
  9.9499478009063858e-01  4.9916426216739009e-02
 -1.0020902861389222e-01  4.9748796005932194e-01
```

Here is what the two-step method gives us:

```
|gravity> ruby integrator_driver4a.rb < euler.in
dt = 0.01
dt_dia = 0.1
dt_out = 0.1
dt_end = 0.1
method = ms2
at time t = 0, after 0 steps :
  E_kin = 0.125 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 0
  (E_tot - E_init) / E_init =-0
at time t = 0.1, after 10 steps :
  E_kin = 0.129 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 9.98e-08
  (E_tot - E_init) / E_init =-1.14e-07
```

```
  1.0000000000000000e+00
  9.9499509568711564e-01  4.9917279823914654e-02
 -1.0020396747499755e-01  4.9748845505609013e-01
```

Less accurate, but hey, it is only a second-order scheme, or so we hope. Let's check:

```
|gravity> ruby integrator_driver4b.rb < euler.in
dt = 0.001
dt_dia = 0.1
dt_out = 0.1
dt_end = 0.1
method = ms2
at time t = 0, after 0 steps :
  E_kin = 0.125 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 0
  (E_tot - E_init) / E_init =-0
at time t = 0.1, after 100 steps :
  E_kin = 0.129 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 1.57e-09
  (E_tot - E_init) / E_init =-1.79e-09
  1.0000000000000000e+00
  9.9499478370909766e-01  4.9916434810162169e-02
 -1.0020897588268213e-01  4.9748796564271547e-01
```

Alice: It looks like 2nd order, but can you decrease the time step by another factor of ten?

```
|gravity> ruby integrator_driver4c.rb < euler.in
dt = 0.0001
dt_dia = 0.1
dt_out = 0.1
dt_end = 0.1
method = ms2
at time t = 0, after 0 steps :
  E_kin = 0.125 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 0
  (E_tot - E_init) / E_init =-0
at time t = 0.1, after 1000 steps :
  E_kin = 0.129 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 1.63e-11
  (E_tot - E_init) / E_init =-1.86e-11
  1.0000000000000000e+00
  9.9499478012623654e-01  4.9916426302151512e-02
 -1.0020902807186732e-01  4.9748796011702123e-01
```

Bob: This makes it pretty clear: it is a second-order scheme.

8.4 Stepping Back

Alice: You have been talking about multi-step algorithms. How does this relate to predictor-corrector methods?

Bob: It is the predictor part of a predictor-corrector scheme. It is possible to squeeze some extra accuracy out of a multi-step scheme, by using the information that you get at the end of a step, to redo the last step a bit more accurately, 'correcting' the step you have just 'predicted.'

In this procedure, you do *not* increase the order of the integration scheme. So you only decrease the coefficient of the error, not its dependence on time step length. Still, if the coefficient is sufficiently smaller, it may well be worthwhile to add a correction step, in order to get this extra accuracy.

If you would like to know how these methods have been used historically, you can read Sverre Aarseth's 2003 book *Gravitational N-Body Simulations*. He implemented a predictor-corrector scheme in the early sixties, and his algorithm became the standard integration scheme for collisional N-body calculations, during three decades. It was only with the invention of the Hermite scheme, by Jun makino, in the early nineties, that an alternative was offered.

Alice: I'd like to see a Hermite scheme implementation, too, in that case. As for the older scheme, is there an intuitive reason that the corrector step would add more accuracy?

Bob: The key idea is that the predictor step involves an *extrapolation*, while the corrector step involves an *interpolation*. For the predictor step, we are moving forwards, starting from the information at times $t = -\Delta t$ and $t = 0$, to estimate the situation at time $t = \Delta t$. For the corrector step, effectively what we do is to go back in time, from $t = \Delta t$ back to $t = 0$. Of course, we don't get precisely back to the same position and velocity that we had before, so the difference between the old and the new values is used to correct instead the position and velocity at time $t = \Delta t$.

Alice: I see. So the main point is that extrapolation is inherently less accurate, for the same distance, as interpolation.

Bob: Exactly.

Alice: Let me see whether I really got the idea, by writing it out in equations. The predicted position $\mathbf{r}_{p,1}$ and velocity $\mathbf{v}_{p,1}$ are determined as follows:

$$\mathbf{r}_{p,1} = \mathbf{r}_0 + \mathbf{v}_0\Delta t + \tfrac{1}{2}\mathbf{a}_0(\Delta t)^2$$

$$\mathbf{v}_{p,1} = \mathbf{v}_0 + \mathbf{a}_0\Delta t + \tfrac{1}{2}\mathbf{j}_0(\Delta t)^2 \tag{8.6}$$

This is the extrapolation part. If we now compute the acceleration and jerk at time $t = 1$ using these new values for the position and the velocity, we can take a step backwards.

Of course, going back one step will not return us to the same position $\mathbf{r}_0$ and velocity $\mathbf{v}_0$ that we started with. Instead, we will obtain slightly different values $\tilde{\mathbf{r}}_0$ and $\tilde{\mathbf{v}}_0$ for the position and velocity at time $t = 0$, as follows:

$$\begin{aligned} \tilde{\mathbf{r}}_0 &= \mathbf{r}_{p,1} + \mathbf{v}_{p,1}(-\Delta t) + \tfrac{1}{2}\mathbf{a}_{p,1}(-\Delta t)^2 \\ \tilde{\mathbf{v}}_0 &= \mathbf{v}_{p,1} + \mathbf{a}_{p,1}(-\Delta t) + \tfrac{1}{2}\mathbf{j}_{p,1}(-\Delta t)^2 \end{aligned} \tag{8.7}$$

where $\mathbf{a}_{p,1} = \mathbf{a}(\mathbf{r}_{p,1})$ and where $\mathbf{j}_{p,1}$ is the jerk obtained now from the last two values of the acceleration:

$$\mathbf{j}_{p,1} = \frac{1}{\Delta t}\left(\mathbf{a}_{p,1} - \mathbf{a}_0\right) \tag{8.8}$$

which is the same procedure that we used to obtain

$$\mathbf{j}_0 = \frac{1}{\Delta t}\left(\mathbf{a}_0 - \mathbf{a}_{-1}\right)$$

but just shifted by one unit in time. Note that our last step has really been an interpolation step, since we are now using the acceleration and jerk values at both ends of the step. In contrast, during the extrapolation step, we used information based on the last two accelerations, both of which are gained at the past side of that step.

Bob: Yes, it is nice to show that so explicitly.

8.5 Wishful Thinking.

Alice: Now let us engage in some wishful thinking. We start with Eq. (8.7) which tells us that we return to the wrong starting point when we go backward in time. Wouldn't it be nice if we somehow could find out at which point we should start so that we would be able to go backwards in time and return to the right spot? Let us call these alternative position and velocity at time $t = 1$ the *corrected* values: $\mathbf{r}_{c,1}$ for the position and $\mathbf{v}_{c,1}$ for the velocity. If our wish would be granted, and someone would hand us those values, then they would by definition obey the equations:

$$\mathbf{r}_0 = \mathbf{r}_{c,1} + \mathbf{v}_{c,1}(-\Delta t) + \tfrac{1}{2}\mathbf{a}_{c,1}(-\Delta t)^2$$

$$\mathbf{v}_0 \;\;=\;\; \mathbf{v}_{c,1} + \mathbf{a}_{c,1}(-\Delta t) + \tfrac{1}{2}\mathbf{j}_{c,1}(-\Delta t)^2 \qquad (8.9)$$

Wouldn't that be nice! In that case, we could form expressions for those desired quantities, by bringing them to the left-hand side of these equations:

$$\begin{aligned} \mathbf{r}_{c,1} &= \mathbf{r}_0 + \mathbf{v}_{c,1}\Delta t - \tfrac{1}{2}\mathbf{a}_{c,1}(\Delta t)^2 \\ \mathbf{v}_{c,1} &= \mathbf{v}_0 + \mathbf{a}_{c,1}\Delta t - \tfrac{1}{2}\mathbf{j}_{c,1}(-\Delta t)^2 \end{aligned} \qquad (8.10)$$

Unfortunately, there is no fairy to fulfill our wishes, since the quantities on the right-hand side are not known, as long as the quantities on the left-hand side are not known, since $\mathbf{a}_{c,1} = \mathbf{a}(\mathbf{r}_{c,1})$ and $\mathbf{j}_{c,1}$ is the jerk that would be obtained from the last two values of the acceleration, including the as yet unknown corrected one:

$$\mathbf{j}_{c,1} \;=\; \frac{1}{\Delta t}\,(\mathbf{a}_{c,1} - \mathbf{a}_0) \qquad (8.11)$$

However, we do have an approximate way out: we can start with Eq. (8.10), and try to solve them in an iterative way.

Let us start with the second line in Eq. (8.10), since that only involves quantities that are freshly recomputed, namely the acceleration and jerk, reserving the first equation for later, since that has a velocity term in the right-hand side. As a first step in our iteration process, we will simply replace the corrected values on the right-hand side by the predicted values:

$$\mathbf{v}_{c,1} \;=\; \mathbf{v}_0 + \mathbf{a}_{p,1}\Delta t - \tfrac{1}{2}\mathbf{j}_{p,1}(-\Delta t)^2 \qquad (8.12)$$

Having done that, we can do the same trick for our first guess for the corrected value for the position, since now we can use our first guess for the corrected value for the velocity in the right-hand side:

$$\mathbf{r}_{c,1} \;=\; \mathbf{r}_0 + \mathbf{v}_{c,1}\Delta t - \tfrac{1}{2}\mathbf{a}_{p,1}(-\Delta t)^2 \qquad (8.13)$$

In principle, we could repeat this procedure, plugging the corrected values back into the right-hand sides of these last two equations, getting an even better approximation for the truly corrected position and velocity. If we denote these doubly-corrected values by $\mathbf{r}_{cc,1}$ and $\mathbf{v}_{cc,1}$, we can obtain those as:

$$\begin{aligned} \mathbf{v}_{cc,1} &= \mathbf{v}_0 + \mathbf{a}_{c,1}\Delta t - \tfrac{1}{2}\mathbf{j}_{c,1}(-\Delta t)^2 \\ \mathbf{r}_{cc,1} &= \mathbf{r}_0 + \mathbf{v}_{cc,1}\Delta t - \tfrac{1}{2}\mathbf{a}_{c,1}(\Delta t)^2 \end{aligned} \qquad (8.14)$$

And we can even continue to higher iterations. In practice, however, it is quite likely that most of the improvement will already be gained with the first iteration. So let us implement that first, and see how much more accuracy we will obtain. Let me put those first-iteration equations together:

$$\begin{aligned} \mathbf{v}_{c,1} &= \mathbf{v}_0 + \mathbf{a}_{p,1}\Delta t - \tfrac{1}{2}\mathbf{j}_{p,1}(\Delta t)^2 \\ \mathbf{r}_{c,1} &= \mathbf{r}_0 + \mathbf{v}_{c,1}\Delta t - \tfrac{1}{2}\mathbf{a}_{p,1}(\Delta t)^2 \end{aligned} \tag{8.15}$$

Bob: Yes, this is exactly the procedure, in all detail!

8.6 An Old Friend

Alice: Well, shouldn't we implement this, too?

Bob: We may as well. It shouldn't be hard.

Alice: So? You look puzzled, still staring at the last set of equations. Something wrong?

Bob: I'm bothered by something. They just look too familiar.

Alice: What do you mean?

Bob: I've seen these before. You know what? I bet these are just one way to express the leapfrog algorithm!

Alice: The leapfrog? Do you think we have reinvented the leapfrog algorithm in such a circuitous way?

Bob: I bet we just did.

Alice: It's not that implausible, actually: there are only so many ways in which to write second-order algorithms. As soon as you go to fourth order and higher, there are many more possibilities, but for second order you can expect some degeneracies to occur.

Bob: But before philosophizing further, let me see whether my hunch is actually correct. First I'll get rid of the predicted value for the jerk, using Eq. (8.8) in order to turn Eq. (8.15) into expressions only in terms of the acceleration:

$$\begin{aligned} \mathbf{v}_{c,1} &= \mathbf{v}_0 + \tfrac{1}{2}\left(\mathbf{a}_0 + \mathbf{a}_{p,1}\right)\Delta t \\ \mathbf{r}_{c,1} &= \mathbf{r}_0 + \mathbf{v}_{c,1}\Delta t - \tfrac{1}{2}\mathbf{a}_{p,1}(\Delta t)^2 \end{aligned} \tag{8.16}$$

And hey, there we have our old friend the leapfrog already!

Alice: I guess your friendship with the frog is stronger than mine. Can you show it to me more explicitly?

Bob: So that you can kiss the frog?

Alice: I'd rather not take my chances; I prefer to let it leap.

Bob: If you plug the expression for $\mathbf{v}_{c,1}$ in the first line of Eq. (8.16) back into the right-hand side of the second line, you get:

$$\begin{aligned}\mathbf{v}_{c,1} &= \mathbf{v}_0 + \tfrac{1}{2}\left(\mathbf{a}_0 + \mathbf{a}_{p,1}\right)\Delta t \\ \mathbf{r}_{c,1} &= \mathbf{r}_0 + \mathbf{v}_0\Delta t + \tfrac{1}{2}\mathbf{a}_0(\Delta t)^2\end{aligned} \tag{8.17}$$

And this is *exactly* how we started to implement the leapfrog, way back when, with Eq. (1.2), which we wrote as:

$$\begin{aligned}\mathbf{r}_{i+1} &= \mathbf{r}_i + \mathbf{v}_i\Delta t + \mathbf{a}_i(\Delta t)^2/2 \\ \mathbf{v}_{i+1} &= \mathbf{v}_i + (\mathbf{a}_i + \mathbf{a}_{i+1})\Delta t/2\end{aligned}$$

Alice: I see, yes, no arguing with that. Good! This will save us a bit of work, since now we don't have to implement a corrected version of your method `ms2`, given that we already have it in the code, in the form of the `leapfrog` method.

Bob: Yes, apart from one very small detail: if we were to write a corrected version of `ms2`, then the very first step would still have to be a Runge-Kutta step, using `rk2`. From there on, we would *exactly* follow the same scheme as the `leapfrog` method.

But I agree, no need to implement that.

Alice: And guess what, we got a bonus, in addition!

Bob: A bonus?

Alice: Yes, there is no need for higher iterations, since we have *exactly* $\mathbf{r}_{cc,1} = \mathbf{r}_{c,1}$ and $\mathbf{v}_{cc,1} = \mathbf{v}_{c,1}$. This is because the leapfrog is time reversible as we have shown before.

Bob: Right you are, how nice! And I'm afraid this is just a fluke for the second-order case.

Alice: Yes, something tells me that we won't be so lucky when we go to higher order.

Chapter 9

A Four-Step Method

9.1 Combinatorics

Bob: Ready to go for a fourth-order multi-step algorithm?

Alice: Ready whenever you are!

Bob: I'll start by repeating the steps I showed you for the second-order algorithm. Instead of Eq. (8.1), we now have to expand the acceleration in a Taylor series up to the crackle:

$$\mathbf{a} = \mathbf{a}_0 + \mathbf{j}_0 t + \tfrac{1}{2}\mathbf{s}_0 t^2 + \tfrac{1}{6}\mathbf{c}_0 t^3 + O(t^4) \tag{9.1}$$

And instead of Eq. (8.2) for the previous value of the acceleration, we now need to remember *three* previous values:

$$\begin{aligned}
\mathbf{a}_{-1} &= \mathbf{a}_0 - \mathbf{j}_0 \Delta t + \tfrac{1}{2}\mathbf{s}_0(\Delta t)^2 - \tfrac{1}{6}\mathbf{c}_0(\Delta t)^3 \\
\mathbf{a}_{-2} &= \mathbf{a}_0 - 2\mathbf{j}_0 \Delta t + 2\mathbf{s}_0(\Delta t)^2 - \tfrac{4}{3}\mathbf{c}_0(\Delta t)^3 \\
\mathbf{a}_{-3} &= \mathbf{a}_0 - 3\mathbf{j}_0 \Delta t + \tfrac{9}{2}\mathbf{s}_0(\Delta t)^2 - \tfrac{9}{2}\mathbf{c}_0(\Delta t)^3
\end{aligned} \tag{9.2}$$

Alice: And these we have to invert, in order to regain the three values for $\mathbf{j}_0$, $\mathbf{s}_0$ and $\mathbf{c}_0$, which we can then use in Eq. (9.1). The only difference is that the inversion is a little harder than it was for getting Eq. (8.3).

Bob: Just a little. Good luck!

Alice: Oh, I have to do it again? Well, I guess I sort-of like doing this, up to a point.

Bob: And if you dislike it, you certainly dislike it far less than I do.

Alice: Okay, I'll do it. From Eq. (9.2) we can form the following linear combination, subtracting the second equation from twice the first one, in order to eliminate the term with the jerk in it. Similarly, we can subtract the third equation from thrice the first one, and we wind up with two equations for the two remaining variables, the snap $\mathbf{s}_0$ and the crackle $\mathbf{c}_0$:

$$\begin{aligned} 2\mathbf{a}_{-1} - \mathbf{a}_{-2} &= \mathbf{a}_0 - \mathbf{s}_0(\Delta t)^2 + \mathbf{c}_0(\Delta t)^3 \\ 3\mathbf{a}_{-1} - \mathbf{a}_{-3} &= 2\mathbf{a}_0 - 3\mathbf{s}_0(\Delta t)^2 + 4\mathbf{c}_0(\Delta t)^3 \end{aligned} \tag{9.3}$$

The next step is to eliminate $\mathbf{s}_0$, in order to obtain an equation for $\mathbf{c}_0$, as follows:

$$3\,(2\mathbf{a}_{-1} - \mathbf{a}_{-2}) - (3\mathbf{a}_{-1} - \mathbf{a}_{-3}) \,=\, \mathbf{a}_0 - \mathbf{c}_0(\Delta t)^3 \quad \Rightarrow$$

$$3\mathbf{a}_{-1} - 3\mathbf{a}_{-2} + \mathbf{a}_{-3} \,=\, \mathbf{a}_0 - \mathbf{c}_0(\Delta t)^3 \quad \Rightarrow$$

$$\mathbf{c}_0 \,=\, \frac{1}{(\Delta t)^3}\left\{\mathbf{a}_0 - 3\mathbf{a}_{-1} + 3\mathbf{a}_{-2} - \mathbf{a}_{-3}\right\} \tag{9.4}$$

Bob: One down, two to go!

Alice: Each next one will only get easier. I'll substitute Eq. (9.4) in the top line of Eq. (9.3), and that should do it:

$$\begin{aligned} \mathbf{s}_0(\Delta t)^2 &= -2\mathbf{a}_{-1} + \mathbf{a}_{-2} + \mathbf{a}_0 + \mathbf{c}_0(\Delta t)^3 \\ &= -2\mathbf{a}_{-1} + \mathbf{a}_{-2} + \mathbf{a}_0 + \{\mathbf{a}_0 - 3\mathbf{a}_{-1} + 3\mathbf{a}_{-2} - \mathbf{a}_{-3}\} \quad \Rightarrow \end{aligned}$$

$$\mathbf{s}_0 \,=\, \frac{1}{(\Delta t)^2}\left\{2\mathbf{a}_0 - 5\mathbf{a}_{-1} + 4\mathbf{a}_{-2} - \mathbf{a}_{-3}\right\} \tag{9.5}$$

You see, now that wasn't that hard, was it?

Bob: Not if you do it, no. Keep up the good work!

Alice: Now for the home run. I'll substitute Eqs. (9.4) and (9.5) in the top line of Eq. (9.2), and get:

$$\begin{aligned} \mathbf{j}_0\Delta t \;=\; -\mathbf{a}_1 \;+\; \mathbf{a}_0 \;&+\; \tfrac{1}{2}\mathbf{s}_0(\Delta t)^2 - \tfrac{1}{6}\mathbf{c}_0(\Delta t)^3 \\ -\mathbf{a}_1 \;+\; \mathbf{a}_0 \;&+\; \left\{\mathbf{a}_0 - \tfrac{5}{2}\mathbf{a}_{-1} + 2\mathbf{a}_{-2} - \tfrac{1}{2}\mathbf{a}_{-3}\right\} \\ &+\; \left\{-\tfrac{1}{6}\mathbf{a}_0 + \tfrac{1}{2}\mathbf{a}_{-1} - \tfrac{1}{2}\mathbf{a}_{-2} + \tfrac{1}{6}\mathbf{a}_{-3}\right\} \quad \Rightarrow \end{aligned}$$

$$\mathbf{j}_0 = \frac{1}{\Delta t}\left\{\tfrac{11}{6}\mathbf{a}_0 - 3\mathbf{a}_{-1} + \tfrac{3}{2}\mathbf{a}_{-2} - \tfrac{1}{3}\mathbf{a}_{-3}\right\} \tag{9.6}$$

So here are the answers!

Bob: If you didn't make a mistake anywhere, that is.

9.2 Checking

Alice: I thought it was your job to check whether I was not making any mistake!

Bob: I *think* you didn't make any. But as you so often say, it doesn't hurt to check.

Alice: I can't very well argue with myself, can I? Okay, how shall we check things. Doing the same things twice is not the best way to test my calculations, since I may well make the same mistake the second time, too.

It would be much better to come up with an independent check. Ah, I see what we can do: in Eq. (9.2), I can check whether the solutions I just obtained for for $\mathbf{j}_0$, $\mathbf{s}_0$ and $\mathbf{c}_0$ indeed give us those previous three accelerations back again. In each of those three cases, I'll start off with the right-hand sides of the equation, to see whether I'll get the left-hand side back.

Here is the first one:

$$\mathbf{a}_0 - \mathbf{j}_0\Delta t + \tfrac{1}{2}\mathbf{s}_0(\Delta t)^2 - \tfrac{1}{6}\mathbf{c}_0(\Delta t)^3 =$$

$$\begin{array}{l} +\mathbf{a}_0 \\ -\tfrac{11}{6}\mathbf{a}_0 + 3\mathbf{a}_{-1} - \tfrac{3}{2}\mathbf{a}_{-2} + \tfrac{1}{3}\mathbf{a}_{-3} \\ +\mathbf{a}_0 - \tfrac{5}{2}\mathbf{a}_{-1} + 2\mathbf{a}_{-2} - \tfrac{1}{2}\mathbf{a}_{-3} \\ -\tfrac{1}{6}\mathbf{a}_0 + \tfrac{1}{2}\mathbf{a}_{-1} - \tfrac{1}{2}\mathbf{a}_{-2} + \tfrac{1}{6}\mathbf{a}_{-3} \;=\; \mathbf{a}_{-1} \end{array}$$

Bob: Good start!

$$\mathbf{a}_0 - 2\mathbf{j}_0\Delta t + 2\mathbf{s}_0(\Delta t)^2 - \tfrac{4}{3}\mathbf{c}_0(\Delta t)^3 =$$

$$\begin{array}{l} +\mathbf{a}_0 \\ -\tfrac{11}{3}\mathbf{a}_0 + 6\mathbf{a}_{-1} - 3\mathbf{a}_{-2} + \tfrac{2}{3}\mathbf{a}_{-3} \\ +4\mathbf{a}_0 - 10\mathbf{a}_{-1} + 8\mathbf{a}_{-2} - 2\mathbf{a}_{-3} \\ -\tfrac{4}{3}\mathbf{a}_0 + 4\mathbf{a}_{-1} - 4\mathbf{a}_{-2} + \tfrac{4}{3}\mathbf{a}_{-3} \;=\; \mathbf{a}_{-2} \end{array}$$

Alice: And a good middle part, too. Now for the finish:

$$
\begin{aligned}
&\mathbf{a}_0 - 3\mathbf{j}_0\Delta t + \tfrac{9}{2}\mathbf{s}_0(\Delta t)^2 - \tfrac{9}{2}\mathbf{c}_0(\Delta t)^3 \;=\\
&\quad \mathbf{a}_0\\
&-\tfrac{11}{2}\mathbf{a}_0 \;+\; 9\mathbf{a}_{-1} \;-\; \tfrac{9}{2}\mathbf{a}_{-2} \;+\; \mathbf{a}_{-3}\\
&\;+9\mathbf{a}_0 - \tfrac{45}{2}\mathbf{a}_{-1} + 18\mathbf{a}_{-2} - \tfrac{9}{2}\mathbf{a}_{-3}\\
&\;-\tfrac{9}{2}\mathbf{a}_0 + \tfrac{27}{2}\mathbf{a}_{-1} - \tfrac{27}{2}\mathbf{a}_{-2} + \tfrac{9}{2}\mathbf{a}_{-3} \;=\; \mathbf{a}_{-3}
\end{aligned}
$$

Good! Clearly, the expressions in Eqs. (9.6), (9.5) and (9.4) are correct. Phew! And now that that is done . . . what was it again we set out to do?

Bob: We were trying to get an expression for the acceleration at time zero, to third order in the time step, expressed in terms of the previous three accelerations. So all we have to do is to plug your results back into Eq. (9.1).

Alice: Ah, yes, it's easy to forget the thread of a story, if you get lost in eliminations.

9.3 Implementation

Bob: So now it's my turn, I guess. Let me code up your results. I'll open a new file, `ms4body.rb`. Let me first have a look at the previous second-order method `ms2`, for comparison:

```
def ms2(dt)
  if @nsteps == 0
    @prev_acc = acc
    rk2(dt)
  else
    old_acc = acc
    jdt = old_acc - @prev_acc
    @pos += vel*dt + old_acc*0.5*dt*dt
    @vel += old_acc*dt + jdt*0.5*dt
    @prev_acc = old_acc
  end
end
```

Our fourth-order multi-step method should be called `ms4`, clearly. I'll use `rk4` to rev up the engine, something we'll have to do three times now. And instead of using snap and crackle out-of-the-box, so to speak, I'll use them with the appropriate factors of the time step, $\mathbf{s}(\Delta t)^2$ and $\mathbf{c}(\Delta t)^3$, respectively.

Here are the equations that we have to code up, as a generalization of Eq. (8.4):

$$\begin{aligned}
\mathbf{r}_1 &= \mathbf{r}_0 + \mathbf{v}_0\Delta t + \tfrac{1}{2}\mathbf{a}_0(\Delta t)^2 + \tfrac{1}{6}\mathbf{j}_0(\Delta t)^3 + \tfrac{1}{24}\mathbf{s}_0(\Delta t)^4 \\
\mathbf{v}_1 &= \mathbf{v}_0 + \mathbf{a}_0\Delta t + \tfrac{1}{2}\mathbf{j}_0(\Delta t)^2 + \tfrac{1}{6}\mathbf{s}_0(\Delta t)^3 + \tfrac{1}{24}\mathbf{c}_0(\Delta t)^4
\end{aligned} \tag{9.7}$$

Here is my implementation:

```
def ms4(dt)
  if @nsteps == 0
    @ap3 = acc
    rk4(dt)
  elsif @nsteps == 1
    @ap2 = acc
    rk4(dt)
  elsif @nsteps == 2
    @ap1 = acc
    rk4(dt)
  else
    ap0 = acc
    jdt = ap0*(11.0/6.0) - @ap1*3 + @ap2*1.5 - @ap3/3.0
    sdt2 = ap0*2 - @ap1*5 + @ap2*4 - @ap3
    cdt3 = ap0 - @ap1*3 + @ap2*3 - @ap3
    @pos += vel*dt + (ap0/2.0 + jdt/6.0 + sdt2/24.0)*dt*dt
    @vel += ap0*dt + (jdt/2.0 + sdt2/6.0 + cdt3/24.0)*dt
    @ap3 = @ap2
    @ap2 = @ap1
    @ap1 = ap0
  end
end
```

Alice: That's not so much longer than `ms2`, and it still looks neat and orderly! And wait, you have used the division method for our `Vector` class, in dividing by those number like 6.0 and so on.

Bob: That's true! I hadn't even thought about it; it was so natural to do that. What did you say again, when I told you I had added division, together with subtraction, to the `Vector` class?

Alice: I said that I was sure it would come in handy sooner or later.

Bob: I guess it was sooner rather than later.

Alice: And I see that you have abbreviated the names of the variables somewhat. Instead of `@prev_acc` for the previous acceleration, you now use `@ap1`, for the first previous acceleration, I guess. And then, for the yet more previous acceleration before that, you use `@ap2`, and so on.

Bob: Yes. I would have like to write `a-2` for a_{-2} but that would be interpreted as a subtraction of the number `2` from the variable `a`, so I choose `@ap2`, for *a-previous-two*.

9.4 Testing

Alice: Time to test `ms4`.

Bob: Yes, and we may as well start again with the same values as we have been using recently:

```
|gravity> ruby integrator_driver4d.rb < euler.in
dt = 0.01
dt_dia = 0.1
dt_out = 0.1
dt_end = 0.1
method = ms4
at time t = 0, after 0 steps :
  E_kin = 0.125 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 0
  (E_tot - E_init) / E_init =-0
at time t = 0.1, after 10 steps :
  E_kin = 0.129 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 1.29e-10
  (E_tot - E_init) / E_init =-1.48e-10
  1.0000000000000000e+00
  9.9499478015881193e-01  4.9916426246428156e-02
 -1.0020902652762116e-01  4.9748796059474770e-01
```

A good start. Now for time steps that are ten times smaller:

```
|gravity> ruby integrator_driver4e.rb < euler.in
dt = 0.001
dt_dia = 0.1
dt_out = 0.1
dt_end = 0.1
method = ms4
at time t = 0, after 0 steps :
  E_kin = 0.125 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 0
  (E_tot - E_init) / E_init =-0
at time t = 0.1, after 100 steps :
  E_kin = 0.129 , E_pot =  -1 , E_tot = -0.875
```

```
             E_tot - E_init = 2.24e-14
   (E_tot - E_init) / E_init =-2.56e-14
   1.0000000000000000e+00
   9.9499478008957187e-01  4.9916426216151437e-02
  -1.0020902860087451e-01  4.9748796006061335e-01
```

Alice: Congratulations! But this is close to machine accuracy. Can you just double the time step, to check whether the errors get 16 times larger?

```
|gravity> ruby integrator_driver4f.rb < euler.in
dt = 0.002
dt_dia = 0.1
dt_out = 0.1
dt_end = 0.1
method = ms4
at time t = 0, after 0 steps :
  E_kin = 0.125 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 0
  (E_tot - E_init) / E_init =-0
at time t = 0.1, after 50 steps :
  E_kin = 0.129 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 3.45e-13
  (E_tot - E_init) / E_init =-3.95e-13
  1.0000000000000000e+00
  9.9499478008976872e-01  4.9916426216220194e-02
 -1.0020902859668304e-01  4.9748796006170143e-01
```

Bob: There you are! We indeed have a fourth-order multi-step integrator!

9.5 A Predictor-Corrector Version

Alice: Now that everything works, how about trying to apply a corrector loop? In the case of your second-order version, we saw that there was no need to bother doing so, since we convinced ourselves that we just got the leapfrog method back. But that must have been a low-order accident. The fourth-order predictor-corrector version will surely lead to a new algorithm, one that you haven't already implemented in our N-body code.

Bob: I bet you're right. Okay, let's do it!

Alice: Given that I already reconstructed your instructions on paper before, I only have to extend it to two orders higher. Here are the results on a backward extrapolation, starting with the predicted values at time $t = 1$, and going back to time $t = 0$, as a higher-order analogy of Eq. (8.7) :

$$
\begin{aligned}
\tilde{\mathbf{r}}_0 &= \mathbf{r}_{p,1} + \mathbf{v}_{p,1}(-\Delta t) + \tfrac{1}{2}\mathbf{a}_{p,1}(-\Delta t)^2 + \tfrac{1}{6}\mathbf{j}_{p,1}(-\Delta t)^3 + \tfrac{1}{24}\mathbf{s}_{p,1}(-\Delta t)^4 \\
\tilde{\mathbf{v}}_0 &= \mathbf{v}_{p,1} + \mathbf{a}_{p,1}(-\Delta t) + \tfrac{1}{2}\mathbf{j}_{p,1}(-\Delta t)^2 + \tfrac{1}{6}\mathbf{s}_{p,1}(-\Delta t)^3 + \tfrac{1}{24}\mathbf{c}_{p,1}(-\Delta t)^4
\end{aligned} \tag{9.8}
$$

where again $\mathbf{a}_{p,1} = \mathbf{a}(\mathbf{r}_{p,1})$ and where $\mathbf{j}_{p,1}$, $\mathbf{s}_{p,1}$, and $\mathbf{c}_{p,1}$ are the jerk, snap, and crackle as determined from the values of the last four accelerations, which are now $\{\mathbf{a}_{-2}, \mathbf{a}_{-1}, \mathbf{a}_0, \mathbf{a}_{p,1}\}$, instead of the previous series of four that were used in the prediction step, which were $\{\mathbf{a}_{-3}, \mathbf{a}_{-2}, \mathbf{a}_{-1}, \mathbf{a}_0\}$.

The same arguments as I gave before will now lead to the following expression for the first-iteration solution for the corrected quantities, as an extension of Eq. (8.15)

$$
\begin{aligned}
\mathbf{v}_{c,1} &= \mathbf{v}_0 + \mathbf{a}_{p,1}\Delta t - \tfrac{1}{2}\mathbf{j}_{p,1}(\Delta t)^2 + \tfrac{1}{6}\mathbf{s}_{p,1}(\Delta t)^3 - \tfrac{1}{24}\mathbf{c}_{p,1}(\Delta t)^4 \\
\mathbf{r}_{c,1} &= \mathbf{r}_0 + \mathbf{v}_{c,1}\Delta t - \tfrac{1}{2}\mathbf{a}_{p,1}(\Delta t)^2 + \tfrac{1}{6}\mathbf{j}_{p,1}(\Delta t)^3 - \tfrac{1}{24}\mathbf{s}_{p,1}(\Delta t)^4
\end{aligned} \tag{9.9}
$$

Note that once more I have interchanged the order of the calculations of the position and velocity: by calculating the corrected value of the velocity first, I am able to use that one in the expression for the corrected value of the position.

Bob: And this time I certainly do not recognize these equations. It seems that we really have a new algorithm here.

Alice: Let me see whether I can extend your code, to implement this scheme. It would be nice to keep the number of acceleration calculations the same as we had before, doing it only once per time step. After all, when we will generalize this to a true N-body system, it is the calculation of the acceleration that will be most expensive, given that for each particle we will have to loop over all other particles.

However, I need to compute the acceleration at the end of the calculation, at the newly predicted position. This means that I then have to remember the value of the acceleration, so that I could reuse it at the beginning of the next time step. This means that I will have to change the variable `ap0` to a true instance variable `@ap0`, so that it will still be there, the next time we enter `ms4pc`.

I will have to initialize `@ap0` during the last time we use the `rk4` method, so that it is ready to be used when we finally enter the main body of the `ms4pc` method, as I will call it, to distinguish it from your previous implementation `ms4`.

Bob: The `pc` stands for predictor-corrector version, I take it.

9.6 Implementation

Alice: Indeed. Well, the rest is just a matter of coding up the equations we just wrote out. Of course, in the predictor step, there is no need to compute the predicted velocity. It is only the predicted position that we need in order to obtain the acceleration at the new time, and through that, the jerk.

What do you think of this:

```
def ms4pc(dt)
  if @nsteps == 0
    @ap3 = acc
    rk4(dt)
  elsif @nsteps == 1
    @ap2 = acc
    rk4(dt)
  elsif @nsteps == 2
    @ap1 = acc
    rk4(dt)
    @ap0 = acc
  else
    old_pos = pos
    old_vel = vel
    jdt = @ap0*(11.0/6.0) - @ap1*3 + @ap2*1.5 - @ap3/3.0
    sdt2 = @ap0*2 - @ap1*5 + @ap2*4 - @ap3
    cdt3 = @ap0 - @ap1*3 + @ap2*3 - @ap3
    @pos += vel*dt + (@ap0/2.0 + jdt/6.0 + sdt2/24.0)*dt*dt
    @ap3 = @ap2
    @ap2 = @ap1
    @ap1 = @ap0
    @ap0 = acc
    jdt = @ap0*(11.0/6.0) - @ap1*3 + @ap2*1.5 - @ap3/3.0
    sdt2 = @ap0*2 - @ap1*5 + @ap2*4 - @ap3
    cdt3 = @ap0 - @ap1*3 + @ap2*3 - @ap3
    @vel = old_vel + @ap0*dt + (-jdt/2.0 + sdt2/6.0 - cdt3/24.0)*dt
    @pos = old_pos + vel*dt + (-@ap0/2.0 + jdt/6.0 - sdt2/24.0)*dt*dt
  end
end
```

Bob: That ought to work, with a little bit of luck, if we didn't overlook any typos or logopos.

Alice: Logopos?

Bob: Errors in logic, as opposed to typing.

Alice: Yes, both are equally likely to have slipped in. Let's try to run it and if it runs, let's see how well we're doing:

```
|gravity> ruby integrator_driver4dpc_old.rb < euler.in
dt = 0.01
dt_dia = 0.1
dt_out = 0.1
dt_end = 0.1
method = ms4pc
at time t = 0, after 0 steps :
  E_kin = 0.125 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 0
  (E_tot - E_init) / E_init =-0
./ms4body_old.rb:158:in `ms4pc': undefined method `-@' for [-0.000500866850078996, -0.00502304344952675]:Vector (NoMethodError)
from ./ms4body_old.rb:22:in `send'
from ./ms4body_old.rb:22:in `evolve'
from integrator_driver4dpc_old.rb:21
```

9.7 A Logopo

Bob: And it is . . . . a logopo!

Alice: And an inscrutable logopo to boot: what does that mean, `undefined method '-@'` ?!? Just when I grew fond of Ruby, it gives us *that* kind of message – just to keep us on our toes, I guess.

Bob: Beats me. I know what the method `-` is, that is just the subtraction method, which I have neatly defined for our `Vector` class, while I was implementing the second-order version of our series of multi-step integrators. I have no idea as to what the method `-@` could possibly be.

But let us look at the code first, maybe that will tell us something. I'll scrutinize all the minus signs, to see whether we can find a clue there. The middle part of the method `ms4pc` is the same as in the predictor-only method `ms4`, and that one did not give any errors. So we'll have to look in the last five lines. The first three of those are verbatim copies of the lines above, which were already tested, we the culprit really should lurk somewhere in the last *two* lines.

Hmm, minus sign, last two lines, ahaha! Unary minus!!

Alice: Unary minus?? As opposed to a dualistic minus?

Bob: As opposed to a binary minus. There are two different meanings of a minus sign, logically speaking: the minus in $3 - 2 = 1$ and the minus in $-(+1) = -1$. In the first example, the minus indicates that you are subtracting *two* numbers. You could write that operation as $minus(3, 2) = 1$. In the second example, you are dealing only with one number, and you take the negative value of that number. You could write that as $minus(+1) = -1$.

So in the first case, you have a *binary* operator, with two arguments, and in the second case, you have a *unary* operator, with only one argument. Hence the name 'unary minus.'

Alice: A binary minus, hey? I'm getting more and more confused with all those different things that are called binaries. In astronomy we call double stars binaries, in computer science they use binary trees in data structures, and binary notation for the ones and zeros in a register, and we talk about binary data as opposed to ASCII data, and now even a lowly minus can become a binary. Oh well, I guess each use has its own logic.

Bob: And many things in the universe come in pairs.

Alice: Even people trying to write a toy model and stumbling upon unary minuses – or unary mini? Anyway, I see now that the minus sign that I added in front of the jerk in the middle of the line

```
@vel = old_vel + @ap0*dt + (-jdt/2.0 + sdt2/6.0 - cdt3/24.0)*dt
```

was a unary minus. All very logical. And yes, while we had defined a binary minus before, we had not yet defined a unary minus.

Bob: So you committed a logopo.

Alice: And I could correct my logopo by writing `jdt/(-2.0)` instead of `-jdt/2.0`. However, I don't like to do that, it looks quite ugly.

Bob: The alternative is to add a unary minus to our `Vector` class.

Alice: I would like that much better! How do you do that?

Bob: Judging from the error message, Ruby seems to use the convention that `-@` stands for unary minus. But just to make sure, let me look at the manual. Just a moment . . .

. . . and yes, my interpretation was correct. So let's just add it to our collections of methods in the file `vector.rb`. How about the following extension, within the `Vector` class:

```
  def -@
    self.map{|x| -x}.to_v
  end
```

and for good measure, I may as well add a unary plus too:

```
  def +@
    self
  end
```

Alice: I see, just in case you write something like `r_new = +r_old`.

Bob: Exactly. No need to do so, but also no need to let that give you an error message if you do.

9.8 Testing

Alice: Do you think that your one-liner will work?

Bob: Only one way to find out! Let's check it:

```
|gravity> ruby integrator_driver4dpc.rb < euler.in
dt = 0.01
dt_dia = 0.1
dt_out = 0.1
dt_end = 0.1
method = ms4pc
at time t = 0, after 0 steps :
  E_kin = 0.125 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 0
  (E_tot - E_init) / E_init =-0
at time t = 0.1, after 10 steps :
  E_kin = 0.129 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = -9.56e-12
  (E_tot - E_init) / E_init =1.09e-11
  1.0000000000000000e+00
  9.9499478008669873e-01  4.9916426232219237e-02
 -1.0020902876280345e-01  4.9748796001291246e-01
```

Alice: What a charm! Not only does the unary minus fly, also the corrected version turns out to be more than ten times more accurate than the predictor-only version, the method **ms4**. Let's give a ten times smaller time step:

```
|gravity> ruby integrator_driver4epc.rb < euler.in
dt = 0.001
dt_dia = 0.1
dt_out = 0.1
dt_end = 0.1
method = ms4pc
at time t = 0, after 0 steps :
  E_kin = 0.125 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 0
  (E_tot - E_init) / E_init =-0
at time t = 0.1, after 100 steps :
  E_kin = 0.129 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = -1.11e-15
  (E_tot - E_init) / E_init =1.27e-15
  1.0000000000000000e+00
  9.9499478008955766e-01  4.9916426216148800e-02
 -1.0020902860118561e-01  4.9748796006053242e-01
```

Even better, by a factor of about twenty this time. Great! I now feel I understand the idea of predictor-corrector schemes. We're building up a good laboratory for exploring algorithms!

Bob: Yes, I've never been able to plug and play so rapidly with different integration schemes. Using a scripting language like Ruby definitely has its advantages.

Chapter 10

Multi-Step Methods

10.1 An Six-Step Method

Alice: Hi, Bob! You've been up to something again, I can tell.

Bob: I got hooked on exploring multi-step methods.

Alice: Don't tell me that you went to 6th order!

Bob: No, in the sense that I did not *just* go to 6th order. I actually went to 6th *and* 8th order.

Alice: No! That can't be, with your dislike for combinatorics. You're pulling my leg.

Bob: No, I did not do the combinatorics myself. Instead, I used a symbolic manipulation package. When I saw you struggling with all those substitutions when we tracked the fourth-order nature of the Runge-Kutta integrator, I was already wondering whether we shouldn't use a symbolic package. At that point I didn't bring it up, since we were in the middle of trying to figure out how things worked in the first place, and introducing yet another tool might have been confusing.

And also when we generalized my two-step method to a four-step method, I could see the merit of tracking things done with pen and paper. Certainly when we're going to show this to our students, we should demand that they are at least *able* to do it with pen and paper. But now that the procedure has become crystal clear, we are just faced with solving a large number of linear equations with the same number of unknowns, and that is really not something I want to do by hand; not would I learn much from doing so.

Incidentally, I also had a look at the literature, and I found that people general solve this type of problem using what they call divided differentces, really a form of Newton interpolation. In that case they do not have to solve the system of equations that we set out to do. Even so, their approach is not that simple either when going to higher order. Given the ease of using a symbolic manipulation program, I think that our approach is just as good as the standard one.

Here is what I found, with the results for the coefficients directly plugged back into the code:

```
def ms6(dt)
  if @nsteps == 0
    @ap5 = acc
    yo6(dt)
  elsif @nsteps == 1
    @ap4 = acc
    yo6(dt)
  elsif @nsteps == 2
    @ap3 = acc
    yo6(dt)
  elsif @nsteps == 3
    @ap2 = acc
    yo6(dt)
  elsif @nsteps == 4
    @ap1 = acc
    yo6(dt)
  else
    ap0 = acc
    jdt = (ap0*137 - @ap1*300 + @ap2*300 - @ap3*200 + @ap4*75 - @ap5*12)/60
    sdt2 = (ap0*45 - @ap1*154 + @ap2*214 - @ap3*156 + @ap4*61 - @ap5*10)/12
    cdt3 = (ap0*17 - @ap1*71 + @ap2*118 - @ap3*98 + @ap4*41 - @ap5*7)/4
    pdt4 = ap0*3 - @ap1*14 + @ap2*26 - @ap3*24 + @ap4*11 - @ap5*2
    xdt5 = ap0 - @ap1*5 + @ap2*10 - @ap3*10 + @ap4*5 - @ap5
    @pos += (vel+(ap0+ (jdt+(sdt2+(cdt3+pdt4/6)/5)/4)/3)*dt/2)*dt
    @vel += (ap0+(jdt+(sdt2+(cdt3+(pdt4+xdt5/6)/5)/4)/3)/2)*dt
    @ap5 = @ap4
    @ap4 = @ap3
    @ap3 = @ap2
    @ap2 = @ap1
    @ap1 = ap0
  end
end
```

10.2 An Eight-Step Method

Alice: Again not so bad, and each time derivative still fits on one line. I see a jerk, a snap, a crackle, and a pop. But what is that next line, this `x` variable?

It must be the derivative of the pop, given that you multiply it with yet one higher power of the time step. What does it stand for?

Bob: To be honest, I had no idea what to call it, so I just called it `x`. The advantage was that I could then call the next two derivatives `y` and `z`, for the eighth-order scheme.

Alice: Iron logic! And you used Yoshida's sixth-order integrator, to rev up the engine, five times.

Bob: Yes, I am glad we had both `yo6` and `yo8` lying around. They sure came in handy!

Alice: And you'd like to show me the eighth-order version as well, don't you?

Bob: I can't wait! Here it is, in file `ms8body.rb`.

```
def ms8(dt)
  if @nsteps == 0
    @a7 = acc
    yo8(dt)
  elsif @nsteps == 1
    @a6 = acc
    yo8(dt)
  elsif @nsteps == 2
    @a5 = acc
    yo8(dt)
  elsif @nsteps == 3
    @a4 = acc
    yo8(dt)
  elsif @nsteps == 4
    @a3 = acc
    yo8(dt)
  elsif @nsteps == 5
    @a2 = acc
    yo8(dt)
  elsif @nsteps == 6
    @a1 = acc
    yo8(dt)
  else
    a0 = acc
    j = (a0*1089 - @a1*2940 + @a2*4410 - @a3*4900 + @a4*3675 - @a5*1764 +
         @a6*490 - @a7*60)/420
    s = (a0*938 - @a1*4014 + @a2*7911 - @a3*9490 + @a4*7380 - @a5*3618 +
         @a6*1019 - @a7*126)/180
    c = (a0*967 - @a1*5104 + @a2*11787 - @a3*15560 +
        @a4*12725 - @a5*6432 + @a6*1849 - @a7*232)/120
    p = (a0*56 - @a1*333 + @a2*852 - @a3*1219 + @a4*1056 - @a5*555 +
```

```
          @a6*164 - @a7*21)/6
     x = (a0*46 - @a1*295 + @a2*810 - @a3*1235 +
          @a4*1130 - @a5*621 + @a6*190 - @a7*25)/6
     y = a0*4 - @a1*27 + @a2*78 - @a3*125 + @a4*120 - @a5*69 + @a6*22 - @a7*3
     z = a0 - @a1*7 + @a2*21 - @a3*35 + @a4*35 - @a5*21 + @a6*7 - @a7
     @pos += (vel+(a0+(j+(s+(c+(p+(x+y/8)/7)/6)/5)/4)/3)*dt/2)*dt
     @vel += (a0 +(j +(s+(c+(p+(x+(y+z/8)/7)/6)/5)/4)/3)/2)*dt
     @a7 = @a6
     @a6 = @a5
     @a5 = @a4
     @a4 = @a3
     @a3 = @a2
     @a2 = @a1
     @a1 = a0
   end
 end
```

As you can see, this time I abbreviated the variables yet a little more: instead of `@ap3`, for example, I wrote `@a3`, since by now the meaning is clear, I hope.

10.3 Very Strange

Alice: Let's see whether your symbolic manipulator has done its job correctly . . .

Bob: . . . and whether I have done my job correctly, typing in the results. We'll start with our, by now almost canonical, set of values:

```
|gravity> ruby integrator_driver4g.rb < euler.in
dt = 0.01
dt_dia = 0.1
dt_out = 0.1
dt_end = 0.1
method = ms6
at time t = 0, after 0 steps :
  E_kin = 0.125 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 0
  (E_tot - E_init) / E_init =-0
at time t = 0.1, after 10 steps :
  E_kin = 0.129 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 3.38e-13
  (E_tot - E_init) / E_init =-3.86e-13
  1.0000000000000000e+00
  9.9499478008960474e-01  4.9916426216165405e-02
```

```
 -1.0020902859905861e-01  4.9748796006154566e-01
```

Hmm, almost too good. Let's be bold and try a much longer time step, to get some more playing room, away from machine accuracy.

```
|gravity> ruby integrator_driver4h.rb < euler.in
dt = 0.05
dt_dia = 0.1
dt_out = 0.1
dt_end = 0.1
method = ms6
at time t = 0, after 0 steps :
  E_kin = 0.125 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 0
  (E_tot - E_init) / E_init =-0
at time t = 0.1, after 2 steps :
  E_kin = 0.129 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 1.53e-13
  (E_tot - E_init) / E_init =-1.75e-13
  1.0000000000000000e+00
  9.9499478009234266e-01  4.9916426209296906e-02
 -1.0020902857520513e-01  4.9748796006113827e-01
```

Hey, that is *very* strange! Now the errors are getting better, for a five time longer time step? That can't be!

Alice: That is strange indeed. Independent of the order, the results should at least get worse, certainly for such a short part of a Kepler orbit. It's not like we just encountered a singularity or something.

Bob: It feels as if I have just violated the second law of thermodynamics or something. How can that be?

10.4 Wiggling the Wires

Alice: As you say in cases like this, let's wiggle the wires, just to see what happens. How about taking an even larger time step, by a factor of two?

Bob: May as well. Here is the result:

```
|gravity> ruby integrator_driver4i.rb < euler.in
dt = 0.1
dt_dia = 0.1
```

```
dt_out = 0.1
dt_end = 0.1
method = ms6
at time t = 0, after 0 steps :
  E_kin = 0.125 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 0
  (E_tot - E_init) / E_init =-0
at time t = 0.1, after 1 steps :
  E_kin = 0.129 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 9.12e-12
  (E_tot - E_init) / E_init =-1.04e-11
  1.0000000000000000e+00
  9.9499478026806454e-01  4.9916425775239165e-02
 -1.0020902692758932e-01  4.9748796009965129e-01
```

Alice: And now the error finally gets worse, and by a factor of about 60, just what you would expect for a 6th-order scheme.

Bob: If we would have just done the last two runs, we would have been very happy and content. How difficult it is, to test a piece of software! We know that there is something *seriously* wrong, because the first run of the last three gave a result that was way off, and yet the last two runs with much larger time steps seem to behave like a charm.

Alice: Like a charm indeed. Now there must be *something* different, between the first run, and the last two.

Bob: But what can there be different? Same code, some procedure, same Bob, same Alice.

Alice: Only larger time steps. And since you kept the total integration time constant, fewer steps in total. In fact, the last run had just one step, and the run before that only two steps. But how can that make a diff . . . . Ah! You talked about revving up the engine . . .

Bob: . . . so if I am only doing *one* or *two* time steps, as I've done in the last two runs . . .

Alice: . . . you're not testing Bob, you're testing Yoshida!

Bob: And clearly, Yoshida has done a very good job indeed. And in the next-to-last run, where I took ten time steps, I just started to run the eight-step method, but only in the last three steps. Okay! What a relief. No problems with the law of thermodynamics, after all!

10.5 Testing the Six-Step Method

Alice: So we have to do many more steps, much more than the minimum number of seven, needed to switch over from the startup `yo6` method to the real `ms6` method. Let's shoot for a full hundred time steps:

```
|gravity> ruby integrator_driver4j.rb < euler.in
dt = 0.01
dt_dia = 1
dt_out = 1
dt_end = 1
method = ms6
at time t = 0, after 0 steps :
  E_kin = 0.125 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 0
  (E_tot - E_init) / E_init =-0
at time t = 1, after 100 steps :
  E_kin = 0.867 , E_pot =  -1.74 , E_tot = -0.875
             E_tot - E_init = 1.31e-08
  (E_tot - E_init) / E_init =-1.5e-08
  1.0000000000000000e+00
  4.3185799584762230e-01  3.7795822363439124e-01
 -1.3171720029068033e+00  5.0109728337030257e-03
```

Bob: Wow, Yoshida's integrator is great, in comparison. Let's see what happens if we go to a thousand time steps instead, making each step ten times smaller. For a sixth order scheme that should make the error a full million times smaller.

```
|gravity> ruby integrator_driver4k.rb < euler.in
dt = 0.001
dt_dia = 1
dt_out = 1
dt_end = 1
method = ms6
at time t = 0, after 0 steps :
  E_kin = 0.125 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 0
  (E_tot - E_init) / E_init =-0
at time t = 1, after 1000 steps :
  E_kin = 0.867 , E_pot =  -1.74 , E_tot = -0.875
             E_tot - E_init = 2.02e-14
  (E_tot - E_init) / E_init =-2.31e-14
  1.0000000000000000e+00
  4.3185799595666452e-01  3.7795822148734887e-01
 -1.3171719961439259e+00  5.0109410148471960e-03
```

Alice: And yes, the errors *are* almost a million times smaller. Now we're getting there. Let's do a final test, making the time step twice as long again:

```
|gravity> ruby integrator_driver4l.rb < euler.in
dt = 0.002
dt_dia = 1
dt_out = 1
dt_end = 1
method = ms6
at time t = 0, after 0 steps :
  E_kin = 0.125 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 0
  (E_tot - E_init) / E_init =-0
at time t = 1, after 500 steps :
  E_kin = 0.867 , E_pot =  -1.74 , E_tot = -0.875
             E_tot - E_init = 1.36e-12
  (E_tot - E_init) / E_init =-1.55e-12
  1.0000000000000000e+00
  4.3185799595664653e-01  3.7795822148753511e-01
 -1.3171719961446775e+00  5.0109410176396871e-03
```

Great! Close to factor 64, and now really with the multi-step integrator, not with a little startup help from Yoshida. We're talking 500 and 1000 steps here, so we're finally testing the six-step code itself.

10.6 Testing the Eight-Step Method

Bob: Time to test the eighth-order implementation. Given that we are now warned about not using few time steps, let's start with one hundred steps, as we did before:

```
|gravity> ruby integrator_driver4m.rb < euler.in
dt = 0.01
dt_dia = 1
dt_out = 1
dt_end = 1
method = ms8
at time t = 0, after 0 steps :
  E_kin = 0.125 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 0
  (E_tot - E_init) / E_init =-0
at time t = 1, after 100 steps :
  E_kin = 0.867 , E_pot =  -1.74 , E_tot = -0.875
             E_tot - E_init = 5.61e-10
  (E_tot - E_init) / E_init =-6.41e-10
  1.0000000000000000e+00
```

```
  4.3185799594296315e-01  3.7795822152601549e-01
 -1.3171719965318329e+00  5.0109417456880440e-03
```

Alice: Not bad. But with an eighth-order scheme, we can't make the time step ten times smaller, since that would go way beyond machine accuracy, for double-precision calculations. Let's just halve the time step. For an eighth-order scheme that should gain us an accuracy factor of $2^8 = 256$.

```
|gravity> ruby integrator_driver4n.rb < euler.in
dt = 0.005
dt_dia = 1
dt_out = 1
dt_end = 1
method = ms8
at time t = 0, after 0 steps :
  E_kin = 0.125 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 0
  (E_tot - E_init) / E_init =-0
at time t = 1, after 200 steps :
  E_kin = 0.867 , E_pot =  -1.74 , E_tot = -0.875
             E_tot - E_init = 3.44e-12
  (E_tot - E_init) / E_init =-3.93e-12
  1.0000000000000000e+00
  4.3185799595658086e-01  3.7795822148755803e-01
 -1.3171719961463324e+00  5.0109410188389162e-03
```

Bob: That's getting closer to a factor 256, but we're not quite there yet. I'll try to shrink the time step by another factor of two:

```
|gravity> ruby integrator_driver4o.rb < euler.in
dt = 0.0025
dt_dia = 1
dt_out = 1
dt_end = 1
method = ms8
at time t = 0, after 0 steps :
  E_kin = 0.125 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 0
  (E_tot - E_init) / E_init =-0
at time t = 1, after 400 steps :
  E_kin = 0.867 , E_pot =  -1.74 , E_tot = -0.875
             E_tot - E_init = 1.45e-14
  (E_tot - E_init) / E_init =-1.66e-14
```

```
   1.0000000000000000e+00
   4.3185799595666458e-01  3.7795822148734654e-01
  -1.3171719961439252e+00  5.0109410148204553e-03
```

Alice: Closer! And certainly closer to 256 than 128.

Bob: And close to machine accuracy as well.

Alice: I think we can declare victory, once again. Congratulations, Bob: your second implementation of an eighth-order integration scheme, after your `yo8` implementation. And this time you found all the coefficients yourself!

Bob: Or, to be honest, the symbolic integrator found them. Well, I'm glad it all worked, especially after that scare with that start-up mystery!

And of course, if we really wanted to go all out, I should implement full predictor-corrector versions for each of them. Well, perhaps some other day.

Chapter 11

The Hermite Algorithm

11.1 A Self-Starting Fourth-Order Scheme

Alice: I'm quite impressed with the large collection of integrators we have built up. But perhaps it is time to move on, and adapt all of them to the real N-body problem, now that we have their two-body form?

Bob: Yes, but there is one more integrator that I wanted to add to our pile. I had mentioned the Hermite algorithm, when we started to look at multiple-step methods, as an alternative to get fourth-order accuracy, without using multiple steps and without using half steps. Actually, coding it up was even simpler than I had expected. You could argue that it is the simplest self-starting Fourth-Order Scheme, once you allow the direct calculation of the jerk, in addition to that of the acceleration.

Alice: Can you show me first the idea behind the algorithm?

Bob: The trick is to differentiate Newton's gravitational equations of motion

$$\mathbf{a}_i = G \sum_{\substack{j=1 \\ j \neq i}}^{N} \frac{M_j}{r_{ji}^2} \hat{\mathbf{r}}_{ji} \qquad (11.1)$$

which I have written here for the general N-body problem. Differentiation with respect to time gives us the jerk directly:

$$\mathbf{j}_i = G \sum_{\substack{j=1 \\ j \neq i}}^{N} M_j \left[\frac{\mathbf{v}_{ji}}{r_{ji}^3} - 3 \frac{(\mathbf{r}_{ji} \cdot \mathbf{v}_{ji})\mathbf{r}_{ji}}{r_{ji}^5} \right] \qquad (11.2)$$

where $\mathbf{v}_{ji} = \mathbf{v}_j - \mathbf{v}_i$.

Note that the jerk has one very convenient property. Although the expression above looks quite a bit more complicated than Newton's original equations, they can still be evaluated through one pass over the whole N-body system. This is no longer true for higher derivatives. For example, we can obtain the fourth derivative of the position of particle i, the *snap*, by differentiating the above equation one more time:

$$\begin{aligned}\mathbf{s}_i = G\sum_{\substack{j=1\\j\neq i}}^{N} M_j \Bigg[& \frac{\mathbf{a}_{ji}}{r_{ji}^3} - 6\frac{(\mathbf{r}_{ji}\cdot\mathbf{v}_{ji})}{r_{ji}^5}\mathbf{v}_{ji} \\ & + \left\{-3\frac{v_{ji}^2}{r_{ji}^5} - 3\frac{(\mathbf{r}_{ji}\cdot\mathbf{a}_{ji})}{r_{ji}^5} + 15\frac{(\mathbf{r}_{ji}\cdot\mathbf{v}_{ji})^2}{r_{ji}^7}\right\}\mathbf{r}_{ji}\Bigg] \end{aligned} \tag{11.3}$$

where $\mathbf{a}_{ji} = \mathbf{a}_j - \mathbf{a}_i$, and this is the expression that thickens the plot. Unlike the $\mathbf{r}_{ji}$ and $\mathbf{v}_{ji}$ expressions, that are given by the initial conditions, $\mathbf{a}_{ji}$ has to be calculated from the positions and velocities. However, this calculation does not only involve the pairwise attraction of particle j on particle i, but in fact all pairwise attractions of all particles on each other! This follows immediately when we write out what the shorthand implies:

$$\mathbf{a}_{ji} = \mathbf{a}_j - \mathbf{a}_i = G\sum_{\substack{k=1\\k\neq j}}^{N}\frac{M_k}{r_{kj}^3}\mathbf{r}_{kj} - G\sum_{\substack{k=1\\k\neq i}}^{N}\frac{M_k}{r_{ki}^3}\mathbf{r}_{ki} \tag{11.4}$$

When we substitute this back into Eq. (11.3), we see that we have to do a double pass over the N-body system, summing over both indices k and j in order to compute a single fourth derivative for the position of particle i.

11.2 A Higher-Order Leapfrog Look-Alike

Alice: I see. So it is quite natural to differentiate Newton's equations of motion just once, and then to use both the acceleration and the jerk, obtained directly from Eqs. (11.1) and (11.2). Can you show me what the expressions for the position and velocity look like, in terms of those two quantities?

Bob: Here they are:

$$\begin{aligned}\mathbf{r}_{i+1} &= \mathbf{r}_i + \tfrac{1}{2}(\mathbf{v}_i + \mathbf{v}_{i+1})\Delta t + \tfrac{1}{12}(\mathbf{a}_i - \mathbf{a}_{i+1})(\Delta t)^2 \\ \mathbf{v}_{i+1} &= \mathbf{v}_i + \tfrac{1}{2}(\mathbf{a}_i + \mathbf{a}_{i+1})\Delta t + \tfrac{1}{12}(\mathbf{j}_i - \mathbf{j}_{i+1})(\Delta t)^2 \end{aligned} \tag{11.5}$$

Alice: That looks like a direct generalization of the leapfrog! I had not expected such simplicity and elegance.

Bob: Yes, I think you can look at the Hermite scheme as a generalization of the leapfrog, in some sense. If you neglect the terms that are quadratic in Δt, you get exactly the leapfrog back.

Alice: Yes, since in that case you can write the terms linear in Δt as effectively half-step quantities, up to higher powers in Δt :

$$\begin{aligned}
\mathbf{r}_{i+1} &= \mathbf{r}_i + \mathbf{v}_{i+1/2}\Delta t + O\left(\Delta t\right)^3 \\
\mathbf{v}_{i+1} &= \mathbf{v}_i + \mathbf{a}_{i+1/2}\Delta t + O\left(\Delta t\right)^3
\end{aligned} \tag{11.6}$$

since the quadratic terms are eliminated by the clever choice of half-step, which characterizes the leapfrog.

In general, you might have expected two extra terms, one of order $(\Delta t)^3$ and one of order $(\Delta t)^4$, for the position, as well as for the velocity. Instead there is just one term, in each case. And that one term is written with a misleading factor of $(\Delta t)^2$, but of course there is at least one more factor Δt lurking in the factor $\mathbf{a}_i - \mathbf{a}_{i+1}$, which to first order in Δt is nothing else than $-\mathbf{j}_i\Delta t$; or $-\mathbf{j}_{i+1}\Delta t$ for that matter. And similarly, there is a factor Δt hidden in $\mathbf{j}_i - \mathbf{j}_{i+1}$.

The question is, can we understand why the remaining terms are what they are, and especially why they can be written in such a simple way?

Bob: I thought you would ask that, and yes, this time I've done my homework, anticipating your questions. Here is the simplest derivation that I could come up with.

Since we are aiming at a fourth-order scheme, all we have to do is to expand the position and velocity up to fourth-order terms in the time step Δt, while we can drop one factor of Δt for the acceleration, and two factors for the jerk, to the same order of consistency.

Alice: You mean that a term proportional to, say, $(\Delta t)^3$ in the jerk would be derived from a term proportional to $(\Delta t)^5$ in the velocity or from a term proportional to $(\Delta t)^6$ in the position, both of which are beyond our approximation.

Bob: Exactly. Here it is, in equation form:

$$\begin{aligned}
\mathbf{r}_{i+1} &= \mathbf{r}_i + \mathbf{v}_i\Delta t + \tfrac{1}{2}\mathbf{a}_i(\Delta t)^2 + \tfrac{1}{6}\mathbf{j}_i(\Delta t)^3 + \tfrac{1}{24}\mathbf{s}_i(\Delta t)^4 \\
\mathbf{v}_{i+1} &= \mathbf{v}_i + \mathbf{a}_i\Delta t + \tfrac{1}{2}\mathbf{j}_i(\Delta t)^2 + \tfrac{1}{6}\mathbf{s}_i(\Delta t)^3 + \tfrac{1}{24}\mathbf{c}_i(\Delta t)^4 \\
\mathbf{a}_{i+1} &= \mathbf{a}_i + \mathbf{j}_i\Delta t + \tfrac{1}{2}\mathbf{s}_i(\Delta t)^2 + \tfrac{1}{6}\mathbf{c}_i(\Delta t)^3 \\
\mathbf{j}_{i+1} &= \mathbf{j}_i + \mathbf{s}_i\Delta t + \tfrac{1}{2}\mathbf{c}_i(\Delta t)^2
\end{aligned} \tag{11.7}$$

Alice: I'll buy that.

11.3 A Derivation

Bob: We can now eliminate snap and crackle at time t_i, expressing them in terms of the acceleration and jerk at times t_i and t_{i+1}, using the last two lines of Eq. (11.7). This gives us for the snap

$$6\mathbf{a}_{i+1} - 2\mathbf{j}_{i+1}\Delta t \;=\; 6\mathbf{a}_i + 4\mathbf{j}_i\Delta t + \mathbf{s}_i(\Delta t)^2 \qquad \Rightarrow$$

$$\mathbf{s}_i(\Delta t)^2 \;=\; -6\mathbf{a}_i + 6\mathbf{a}_{i+1} + -4\mathbf{j}_i\Delta t - 2\mathbf{j}_{i+1}\Delta t \tag{11.8}$$

Substituting this back into the last line of Eq. (11.7), we get for the crackle:

$$\mathbf{j}_{i+1}\Delta t \;=\; -6\mathbf{a}_i + 6\mathbf{a}_{i+1} - 3\mathbf{j}_i\Delta t - 2\mathbf{j}_{i+1}\Delta t + \tfrac{1}{2}\mathbf{c}_i(\Delta t)^3 \qquad \Rightarrow$$

$$\mathbf{c}_i(\Delta t)^3 \;=\; 12\mathbf{a}_i - 12\mathbf{a}_{i+1} + 6\mathbf{j}_i\Delta t + 6\mathbf{j}_{i+1}\Delta t \tag{11.9}$$

Substituting these expressions for the snap and crackle in the second line of Eq. (11.7) we find:

$$\begin{aligned}
\mathbf{v}_{i+1} - \mathbf{v}_i \;&=\; \mathbf{a}_i\Delta t + \tfrac{1}{2}\mathbf{j}_i(\Delta t)^2 + \tfrac{1}{6}\mathbf{s}_i(\Delta t)^3 + \tfrac{1}{24}\mathbf{c}_i(\Delta t)^4 \\
&=\; \mathbf{a}_i\Delta t + \tfrac{1}{2}\mathbf{j}_i(\Delta t)^2 \\
&+\; -\mathbf{a}_i\Delta t + \mathbf{a}_{i+1}\Delta t - \tfrac{2}{3}\mathbf{j}_i(\Delta t)^2 - \tfrac{1}{3}\mathbf{j}_{i+1}(\Delta t)^2 \\
&+\; \tfrac{1}{2}\mathbf{a}_i\Delta t - \tfrac{1}{2}\mathbf{a}_{i+1}\Delta t + \tfrac{1}{4}\mathbf{j}_i(\Delta t)^2 + \tfrac{1}{4}\mathbf{j}_{i+1}(\Delta t)^2 \qquad \Rightarrow
\end{aligned}$$

$$\mathbf{v}_{i+1} = \mathbf{v}_i + \tfrac{1}{2}(\mathbf{a}_i + \mathbf{a}_{i+1})\Delta t + \tfrac{1}{12}(\mathbf{j}_i - \mathbf{j}_{i+1})(\Delta t)^2 \tag{11.10}$$

Indeed, we have recovered the second line of Eq. (11.5), and thereby explained the mysterious factor $\frac{1}{12}$ in the last term.

The only thing left to do is to check the expression for the position, the first line in Eq. (11.5). Let us bring everything to the left-hand side there, except the acceleration terms. In other words, let us split off the leapfrog expression, and see what is left over:

$$\begin{aligned}
\mathbf{r}_{i+1} - \mathbf{r}_i - \tfrac{1}{2}(\mathbf{v}_i + \mathbf{v}_{i+1})\Delta t \;&=\; \{\mathbf{r}_{i+1} - \mathbf{r}_i\} - \tfrac{1}{2}\mathbf{v}_i\Delta t + \left\{-\tfrac{1}{2}\mathbf{v}_{i+1})\right\}\Delta t \\
&=\; \left\{\mathbf{v}_i\Delta t + \tfrac{1}{2}\mathbf{a}_i(\Delta t)^2 + \tfrac{1}{6}\mathbf{j}_i(\Delta t)^3 + \tfrac{1}{24}\mathbf{s}_i(\Delta t)^4\right\} \\
&-\; \tfrac{1}{2}\mathbf{v}_i\Delta t \\
&+\; \left\{-\tfrac{1}{2}\mathbf{v}_i\Delta t - \tfrac{1}{4}(\mathbf{a}_i + \mathbf{a}_{i+1})(\Delta t)^2 - \tfrac{1}{24}(\mathbf{j}_i - \mathbf{j}_{i+1})(\Delta t)^3\right\}
\end{aligned}$$

$$
\begin{aligned}
&= \tfrac{1}{2}\mathbf{a}_i(\Delta t)^2 + \tfrac{1}{6}\mathbf{j}_i(\Delta t)^3 \\
&- \tfrac{1}{4}\mathbf{a}_i(\Delta t)^2 + \tfrac{1}{4}\mathbf{a}_{i+1}(\Delta t)^2 - \tfrac{1}{6}\mathbf{j}_i(\Delta t)^3 - \tfrac{1}{12}\mathbf{j}_{i+1}(\Delta t)^3 \\
&- \tfrac{1}{4}\mathbf{a}_i(\Delta t)^2 - \tfrac{1}{4}\mathbf{a}_{i+1}(\Delta t)^2 - \tfrac{1}{24}\mathbf{j}_i(\Delta t)^3 + \tfrac{1}{24}\mathbf{j}_{i+1}(\Delta t)^3 \\
&= -\tfrac{1}{24}\mathbf{j}_i(\Delta t)^3 - \tfrac{1}{24}\mathbf{j}_{i+1}(\Delta t)^3 \\
&= -\tfrac{1}{24}\mathbf{j}_i(\Delta t)^3 - \tfrac{1}{24}\left\{\mathbf{j}_i + \mathbf{s}_i\Delta t\right\}(\Delta t)^3 \\
&= -\tfrac{1}{12}\mathbf{j}_i(\Delta t)^3 - \tfrac{1}{24}\mathbf{s}_i(\Delta t)^4 \qquad (11.11)
\end{aligned}
$$

To within our order of accuracy, this is indeed what we were trying to prove, since the right-hand term of the first line in Eq. (11.5) can be written as

$$
\begin{aligned}
\tfrac{1}{12}(\mathbf{a}_i - \mathbf{a}_{i+1})(\Delta t)^2 &= \tfrac{1}{12}\mathbf{a}_i(\Delta t)^2 - \tfrac{1}{12}\left\{\mathbf{a}_i + \mathbf{j}_i\Delta t + \tfrac{1}{2}\mathbf{s}_i(\Delta t)^2\right\}(\Delta t)^2 \\
&= -\tfrac{1}{12}\mathbf{j}_i(\Delta t)^3 - \tfrac{1}{24}\mathbf{s}_i(\Delta t)^4 \qquad (11.12)
\end{aligned}
$$

This proves the desired result:

$$
\mathbf{r}_{i+1} = \mathbf{r}_i + \tfrac{1}{2}(\mathbf{v}_i + \mathbf{v}_{i+1})\Delta t + \tfrac{1}{12}(\mathbf{a}_i - \mathbf{a}_{i+1})(\Delta t)^2 \qquad (11.13)
$$

11.4 Implementation

Alice: Great! Can you show me how you have implemented this scheme?

Bob: It was surprisingly simple. I opened a file called `hbody.rb`, with `yoshida6.rb` as my starting point. And all I had to do was to add a method `jerk` to compute the jerk, and then a method `hermite` to do the actual Hermite integration.

The `jerk` method, in fact, is rather similar to the `acc` method:

```
def acc
  r2 = @pos*@pos
  r3 = r2*sqrt(r2)
  @pos*(-@mass/r3)
end
```

apart from the last line:

```
def jerk
  r2 = @pos*@pos
  r3 = r2*sqrt(r2)
  (@vel+@pos*(-3*(@pos*@vel)/r2))*(-@mass/r3)
end
```

Alice: Before you show me the integrator, let me look at the equations again, which you have derived, to get an idea as to what to expect. Here they are:

$$\begin{aligned}\mathbf{r}_{i+1} &= \mathbf{r}_i + \tfrac{1}{2}(\mathbf{v}_i + \mathbf{v}_{i+1})\Delta t + \tfrac{1}{12}(\mathbf{a}_i - \mathbf{a}_{i+1})(\Delta t)^2 \\ \mathbf{v}_{i+1} &= \mathbf{v}_i + \tfrac{1}{2}(\mathbf{a}_i + \mathbf{a}_{i+1})\Delta t + \tfrac{1}{12}(\mathbf{j}_i - \mathbf{j}_{i+1})(\Delta t)^2\end{aligned}$$

Now I remember what bothered me about these expressions. On the left-hand side you express the positions and velocities at time t_{i+1}. However, on the right-hand side you rely also on the acceleration and jerk at time t_{i+1}, which you can only compute *after* you have determined the position and velocity at time t_{i+1}. In addition, you also have the future velocity at the right-hand side as well.

In other words, these are implicit equations for the position and the velocity. How do you deal with them?

Bob: Through iteration. I first determine trial values for the position and velocity, simply by expanding both as a Taylor series, using only what we know at time t_i, which are the position, velocity, acceleration and jerk. In fact, this is a type of predictor-corrector method. If I express those trial values with a subscript p, I start with:

$$\begin{aligned}\mathbf{r}_{p,i+1} &= \mathbf{r}_i + \mathbf{v}_i\Delta t + \tfrac{1}{2}\mathbf{a}_i(\Delta t)^2 + \tfrac{1}{6}\mathbf{j}_i(\Delta t)^3 \\ \mathbf{v}_{p,i+1} &= \mathbf{v}_i + \mathbf{a}_i\Delta t + \tfrac{1}{2}\mathbf{j}_i(\Delta t)^2\end{aligned} \tag{11.14}$$

Then, with those trial values, I determine the trial values for the acceleration and jerk. Using all those trial values as proxies for the actual values at time t_{i+1}, I can solve the equations for the corrector step, where I indicate the final values for the position and velocity at time t_{i+1} with a subscript s :

$$\begin{aligned}\mathbf{r}_{c,i+1} &= \mathbf{r}_i + \tfrac{1}{2}(\mathbf{v}_i + \mathbf{v}_{p,i+1})\Delta t + \tfrac{1}{12}(\mathbf{a}_i - \mathbf{a}_{p,i+1})(\Delta t)^2 \\ \mathbf{v}_{c,i+1} &= \mathbf{v}_i + \tfrac{1}{2}(\mathbf{a}_i + \mathbf{a}_{p,i+1})\Delta t + \tfrac{1}{12}(\mathbf{j}_i - \mathbf{j}_{p,i+1})(\Delta t)^2\end{aligned} \tag{11.15}$$

Alice: Does that really work? Let's see whether your procedure is actally fourth-order accurate. In Eq. (11.14), you leave out terms of order $(\Delta t)^3$ in the velocity, and terms of order $(\Delta t)^4$ in the position, but those predictor values for position and velocity are only used to compute acceleration and jerk, terms that are always multiplied with factors of $(\Delta t)^2$, so the resulting errors are of order $(\Delta t)^5$. Good!

However, I am worried about that velocity term in the first line on the right-hand side of Eq. (11.15). According to Eq. (11.14), your predicted velocity has an error of order $(\Delta t)^3$, so by multiplying that with a single factor Δt, you wind up with an error of order $(\Delta t)^4$, which is certainly not acceptable.

Bob: Huh, you're right. How can that be? Is that really what I implemented? That cannot be correct. I'd better check my code.

Here it is, the `hermite` method that is supposed to do the job:

```
def hermite(dt)
  old_pos = @pos
  old_vel = @vel
  old_acc = acc
  old_jerk = jerk
  @pos += @vel*dt + old_acc*(dt*dt/2.0) + old_jerk*(dt*dt*dt/6.0)
  @vel += old_acc*dt + old_jerk*(dt*dt/2.0)
  @vel = old_vel + (old_acc + acc)*(dt/2.0) + (old_jerk - jerk)*(dt*dt/12.0)
  @pos = old_pos + (old_vel + vel)*(dt/2.0) + (old_acc - acc)*(dt*dt/12.0)
end
```

Ah, I see what I did! Of course, and now I remember also why I did it that way. I was wrong in what I wrote above. I should have written Eq. (11.15) as follows:

$$\begin{aligned} \mathbf{v}_{c,i+1} &= \mathbf{v}_i + \tfrac{1}{2}(\mathbf{a}_i + \mathbf{a}_{p,i+1})\Delta t + \tfrac{1}{12}(\mathbf{j}_i - \mathbf{j}_{p,i+1})(\Delta t)^2 \\ \mathbf{r}_{c,i+1} &= \mathbf{r}_i + \tfrac{1}{2}(\mathbf{v}_i + \mathbf{v}_{c,i+1})\Delta t + \tfrac{1}{12}(\mathbf{a}_i - \mathbf{a}_{p,i+1})(\Delta t)^2 \qquad (11.16) \end{aligned}$$

Alice: I see. By switching the order of computation for the corrected form of the position and velocity, you are able to use the *corrected* version of the velocity, rather than the predicted version, in determining the corrected version of the position. So all is well!

Bob: Yes, indeed. And it is all coming back now: at first I computed $\mathbf{r}_{c,i+1}$ and $\mathbf{v}_{c,i+1}$ in the normal order, and I got a pretty bad energy behavior, until I realized that I should switch the order of computations. How easy to make a quick mistake and a quick fix, and then to forget all about it!

Alice: It shows once again the need for detailed documentation, not only for the mythical user, but at least as much for the actual producer of the code!

11.5 Testing

Bob: To test `hermite`, let us start with a comparison run, taking our old `rk` method:

```
|gravity> ruby integrator_driver2j.rb < euler.in
dt = 0.01
dt_dia = 0.1
dt_out = 0.1
dt_end = 0.1
method = rk4
at time t = 0, after 0 steps :
  E_kin = 0.125 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 0
  (E_tot - E_init) / E_init =-0
at time t = 0.1, after 10 steps :
  E_kin = 0.129 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 1.79e-12
  (E_tot - E_init) / E_init =-2.04e-12
  1.0000000000000000e+00
  9.9499478009063858e-01  4.9916426216739009e-02
 -1.0020902861389222e-01  4.9748796005932194e-01
```

I'll give the Hermite scheme the same task:

```
|gravity> ruby integrator_driver5a.rb < euler.in
dt = 0.01
dt_dia = 0.1
dt_out = 0.1
dt_end = 0.1
method = hermite
at time t = 0, after 0 steps :
  E_kin = 0.125 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 0
  (E_tot - E_init) / E_init =-0
at time t = 0.1, after 10 steps :
  E_kin = 0.129 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 5.31e-13
  (E_tot - E_init) / E_init =-6.07e-13
  1.0000000000000000e+00
  9.9499478009151798e-01  4.9916426220332356e-02
 -1.0020902857150518e-01  4.9748796006319129e-01
```

Alice: Not bad! Can you give it a somewhat larger time step, to check whether you really have a fourth-order scheme?

Bob: I'll increase the time step by a factor two:

```
|gravity> ruby integrator_driver5b.rb < euler.in
dt = 0.02
dt_dia = 0.1
dt_out = 0.1
dt_end = 0.1
method = hermite
at time t = 0, after 0 steps :
  E_kin = 0.125 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 0
  (E_tot - E_init) / E_init =-0
at time t = 0.1, after 5 steps :
  E_kin = 0.129 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 7.93e-12
  (E_tot - E_init) / E_init =-9.07e-12
  1.0000000000000000e+00
  9.9499478011948561e-01  4.9916426283208984e-02
 -1.0020902812740490e-01  4.9748796010457508e-01
```

Alice: Close to a factor of 16 degradation, as expected. So we have a working fourth-order Hermite implementation.

Bob: And all that by just adding two short methods to an existing code, without *any* need for *any* other modifications! Ruby really invites you to a game of playful prototyping and exploring new algorithms. I guess I've become converted to using Ruby, by now.

Alice: I'm glad to hear that, but perhaps it is time to come back to our original project, to provide some toy models for your students, to do some actual N-body integrations.

Bob: Yes, I'm afraid I got a bit too much carried away with all these integrators. But it was so much fun! I've never in my life written an eighth-order integrator. Somehow, working in Fortran or C++ made such a thing seem like a tremendous project. But somehow, while I was playing around, one thing led to another, and before I knew it, I had no less than *two* very different eighth-order integrators.

You're right, though, let's get back to our original plan. While we have reached considerable sophistication on the level of algorithms, we are still stuck with $N = 2$. High time to go to large N values.

Chapter 12

Literature References

Handbook of Mathematical Functions, by M. Abramowitz and I. A. Stegun, eds. [Dover, 1965]

Construction of higher order symplectic integrators, by Haruo Yoshida, 1990, Phys. Lett. A **150**, 262.

Gravitational N-Body Simulations, by Sverre Aarseth, 2003 [Cambridge University Press].

Nystrom, E.J., 1925, Acta Soc. Sci. Fenn., 50, No.13, 1-55.

Sprott, J.C., 1997, *Some Simple Chaotic Jerk Functions*, Am. J. Phys. 65, 537.