The Art of Computational Science

The Kali Code

vol. 15

Individual Time Steps:
A Four-Dimensional View

Piet Hut and Jun Makino

September 13, 2007

Contents

Preface

0.1 xxx

We thank xxx, xxx, and xxx for their comments on the manuscript.
Piet Hut and Jun Makino

Chapter 1

The Basic Idea

1.1 The Need for a Predictor

Alice: We now have a code for constant time steps, and one for variable but shared time steps. It's time to bite the bullet, and start working on a code where each particle will have its own individual time step.

This will be quite complicated, but it is certainly necessary. When two particles approach each other closely, we don't want to force all other particles that are far away to take the same small time steps as the closely approaching particles will take.

I've never coded up such a scheme, and I've never seen it in text books on numerical integration either. It seems to be a special trick that is used in astrophysics, in the stellar dynamics community. I presume that you have some experience with . . . why are you smiling?

Bob: Yesterday, after we made such strides with our shared time step code, I just couldn't stop. I've been hacking away the whole evening, and I must admit, a good deal of the night. But I now have a working individual time step version! It's right here under our fingertips, in file `nbody_ind1.rb`.

Alice: So that's why you are smiling, you have every right to do so!

Bob: And I must admit, yes, I do have some experience. In no way could I have come up with all the tricks of the trade in just one night. The most tricky thing is to visualize the scheduling requirements. But why talk about it abstractely if I can show you concretely what I did?

Alice: Please do!

Bob: First of all, I limited myself to the Hermite scheme.

Alice: Why? After you had derived all those nice integrators, why not carry

them over?

Bob: I can see that you don't have experience with individual time step schemes. And in that light, it certainly is a fair question. The answer is that not every algorithm comes with a predictor.

1.2 Crossing the Road

Alice: A predictor?

Bob: Yes. A predictor. Okay, lets start at the basics. If particles have individual time steps, how do you think a single particle will make its next step?

Alice: That will depend on the algorithm.

Bob: Let's say that you have a Runge-Kutta scheme. For definiteness, let us take the second-order Runge-Kutta scheme `rk2` that we have been using.

Alice: Well, in that case it will first make a half-step, with a length of $0.5dt$.

Bob: How will our particle take that step?

Alice: Is this a riddle?

Bob: No, I'm serious. How does the particle step forward?

Alice: How does the chicken cross the road. Well, of course, you take the velocity and acceleration . . .

Bob: . . . How do you obtain the acceleration?

Alice: In the usual way, as the inverse square of the distance between . . .

Bob: . . . How do you determine the distance?

Alice: You take the position of the other particle and . . .

Bob: . . . How do you know the position of the other particle?

Alice: You look it up! Will you never let me finish a sentence?

Bob: I just did. Now, how do you look it up?

Alice: You look at . . . ahaha! Now I get it. The other particle has a different time step size, so it will have a position all right, but a position that is valid for a different time than the time for which the position of our particle is valid.

Bob: So we have to extrapolate the position of the other particle, to predict where it will be by the time we need its position.

Alice: Hence the need for a predictor.

Bob: Exactly.

Alice: That is tricky indeed. How do you use a Runge-Kutta scheme to provide a predictor?

Bob: The answer is: you don't. Or at least astrophysicists don't. They tend to stick to a very small set of algorithms with known predictors.

Alice: But wait a minute. For a second order Runge-Kutta, it should not be too hard to construct a predictor!

Bob: That may be, but I'm sure it's non-trivial already for a fourth-order Runge-Kutta. And this is why you've never heard of a code for collisional stellar dynamics that uses a Runge-Kutta.

Alice: Ah, is that the reason that people only use Hermite and multi-step methods?

Bob: It's not the only reason, I think, but certainly one of the reasons.

Alice: Not a very good reason, for sure. Just the fact that something is not so easy to do doesn't mean you shouldn't try! I'm happy to start with your Hermite implementation, but I sure would like to try other ones as well. Let's set ourselves a goal, to make a fourth-order Runge-Kutta integrator!

Bob: That's quite a challenge. In the almost half century that astronomers have worked with individual time steps, I've never seen or heard about a Runge-Kutta implementation. But why not? If we succeed, it would mean that in at least one respect we would be ahead of the pack.

Alice: I'd love to see that! But okay, for now let's be content with Hermite. Can you show me what you did?

1.3 A Latest Sales Date

Bob: Let me skip the start-up phase for now. Often in life, it is not so easy to start at the beginning. Let's start in the middle instead. Imagine that all of our particles have already taken at least one step. Let me show you how which particle takes the next step.

Alice: How which what?

Bob: There are two questions here: at any given time we first have to determine *which* particle needs to take a step, and then we have to figure out *how* that step can be taken.

Alice: I'm all ears.

Bob: First a bit more background. In the past, the state of each particle was characterized by a mass `@mass`, position `@pos`, and velocity `@vel`, all given at a shared system time `@time`, an instance variable of the class `NBody`. In the case of individual time steps, each particle has its own time `@time`, which is now an instance variable of the class `Body`.

In addition, since each body has its own time step, it has a predicted time `@next_time`, again different for each body, which is the time of completion of

the next time step: `@next_time - time` is by definition the indivdual time step of the particle.

The concept of a `@next_time` is an important one: it plays the role of a latest sales date. If you go to the store to buy some cookies, you can look at the box, and read the latest sales date. If that date is in the past, you probably don't want to buy the cookies, since you can't trust them to be fresh enough.

Alice: Though they won't go bad overnight, as soon as the latest sales date has passed.

Bob: Neither will our particles lose predictability completely, after `@next_time`, but still, it is a good measure of the time until which we can predict the position of a particle. If another particle needs to know the position of our particle at time `t`,, there is no problem as long as `t <= @next_time`.

To sum up: each particle carries with it a predictor that allows it to predict its own position during the interval `[ @time, @next_time ]`.

Alice: If another particle asks for the position of our particle at time `t`, within that interval, how does our particle communicate its predicted position?

Bob: Through its instance variable `@pred_pos`. Here is the hand shaking mechanism. First the other particle sends the time for which it wants to get the information. If the time is outside the proper interval, our particle will complain. If the time is within the interval, our particle will oblige, and predict its position and velocity, and store those values in the variables `@pred_pos` and `@pred_vel`. The other particle can then read off the values, and determine the acceleration and jerk exercised on itself by our particle.

Alice: Ah yes, jerk, we are dealing with a Hermite scheme, and therefore you have to predict the velocity as well.

Bob: Indeed. For any other scheme that we have implemented so far, it would suffice just to predict the position, since we would only need the acceleration.

Alice: So our `Body` class now has four extra instance variables, over and above what we had in the shared time stap class: `time`, `@next_time`, `@pred_pos`, and `@pred_vel`.

Bob: Indeed.

1.4 Scheduling

Alice: You mentioned that you wanted to explain how to find the particle that needs to be moved, before explaining how to move it.

Bob: Yes, but we need still more background first. Let us again call `t` the time at which we decide to look at the system. We know for sure that for all particles, `@time <= t` and `@next_time >= t`.

Alice: Why?

Bob: Whenever a particle runs out of predictive power, it has to be updated immediately. In other words, as soon as the latest sales date, `@next_time`, is passed, the particle position needs to be updated. This means that the particle takes a step at this time, and in doing so, it acquires a new latest sales date. So if all marches well, the latest sales date for each particle will remain in the future, or at worst will be in the present, never in the past. This means `@next_time >= t`.

Alice: Fair enough.

Bob: And since you cannot predict particles past their latest sales date, and since latest sales dates can be arbitrarily close to the present time `t`, no particle can take a step in the future. Only when the time `t` has caught up with the latest sales date `@next_time` of a particular particle, will that particle make a step. When it does so, its new time `time` will take on the same value that `@next_time` had, which at the time of the step taking is exactly `t`. From that time on, `t` will march forward, while `@time` is frozen until the next step. Hence `@time <= t`. QED.

Alice: Hmmm. I sort-of get it, but I must admit, I don't see it very clearly yet. I bet looking at the code will makes things clearer.

Bob: It always does. Here is how it works. Let's jump right into the heart of the matter, the method `evolve`. For comparison, here is what `evolve` looked like in the case of shared time steps:

```
def evolve(c)
  @nsteps = 0
  @e0 = ekin + epot
  write_diagnostics
  t_dia = @time + c.dt_dia
  t_out = @time + c.dt_out
  t_end = @time + c.dt_end
  acs_write if c.init_out_flag

  while @time < t_end
    @dt = c.dt_param * collision_time_scale
    if c.exact_time_flag and @time + @dt > t_out
      @dt = t_out - @time
    end
    send(c.method)
    @time += @dt
    @nsteps += 1
    if @time >= t_dia
      write_diagnostics
      t_dia += c.dt_dia
```

```
      end
      if @time >= t_out - 1.0/VERY_LARGE_NUMBER
        acs_write
        t_out += c.dt_out
      end
    end
  end
```

And here is how I rewrote it for the individual time step case:

```
  def evolve(c)
    @nsteps = 0
    startup(c.dt_param)
    write_diagnostics
    t_dia = @time + c.dt_dia
    t_out = @time + c.dt_out
    t_end = @time + c.dt_end
    acs_write if c.init_out

    while @time < t_end
      np = find_next_particle
      @time = np.next_time
      if (@time < t_end)
        np.autonomous_step(@body, c.dt_param)
        @nsteps += 1
      end
      if @time >= t_dia
        sync(t_dia, c.dt_param)
        @nsteps += @body.size
        write_diagnostics
        t_dia += c.dt_dia
      end
      if @time >= t_out
        sync(t_out, c.dt_param)    # we are now syncing twice, if t_dia = t_out
        @nsteps += @body.size
        acs_write
        t_out += c.dt_out
      end
    end
  end
```

1.5 Back to How and Which

Alice: What does `startup` do?

Bob: Let's look at that later. It records the initial total energy `@e0@`, as in the shared time step case, and it does a lot more. Other than that, the parts up to the `while` loop are the same in both case. So let us jump to the `while` loop, for now.

In the case of shared time steps, we first determined the size of the next (shared) time step, and then we pushed all particles forward, through the command `send(c.method)`. Following that, there are only some administrative issues, related to reporting diagnostics and orchestrating output. These issues are quite similar in both cases. Note that in the case of individual time steps, I've made sure to synchronize the particles will a call to `sync`; but again, let's leave that for later.

Alice: Are you leaving anything for now?

Bob: Yes. Do you remember the *how which* question?

Alice: You were going to tell me how to push which particle forward. That was quite a while ago.

Bob: Well, we needed to lay some foundations. Now I can show you. The *which* part is decided right at the start of the `while` loop, by a call to `find_next_particle`. The *how* part is decided three lines further on: the particle `np` that is found is being asked to take a time step, through its instance method `autonomous_step`.

Alice: Why autonomous?

Bob: This is the generic situation, well after the start and well before the finish of a calculation. As you can see from the `if` condition one line earlier, an autonomous step is only allowed if the end of the step `np.next_time` is still earlier than the finishing time `t_end`. If that is not the case, we have to wrap it, and start synchronizing everything.

Alice: Can you show me what happens in the generic case, with an autonomous step?

Bob: For that, we have to go to the `Body` class.

Chapter 2

The `Body` Class

2.1 Code Listing

Alice: It would be good to print out that whole `Body` class, to get an idea of what is happening on the ground level.

Bob: My pleasure! Here it all is.

```
class Body

  attr_accessor :acc, :jerk,
                :pred_pos, :pred_vel,
                :time, :next_time

  def autonomous_step(ba, dt_param)
    take_one_step(ba, @next_time, dt_param)
  end

  def forced_step(ba, t, dt_param)
    take_one_step(ba, t, dt_param)
  end

  def take_one_step(ba, t, dt_param)
    ba.each do |b|
      b.predict_step(t)
    end
    correct_step(ba, t, dt_param)
  end

  def predict_step(t)
```

```
  if t > @next_time
    STDERR.print "predict_step: t = ", t, " > @next_time = "
    STDERR.print @next_time, "\n"
    exit
  end
  dt = t - @time
  @pred_pos = @pos + @vel*dt + @acc*(dt*dt/2.0) + @jerk*(dt*dt*dt/6.0)
  @pred_vel = @vel + @acc*dt + @jerk*(dt*dt/2.0)
end

def correct_step(ba, t, dt_param)
  dt = t - @time
  new_acc, new_jerk = get_acc_and_jerk(ba)
  new_vel = @vel + (@acc + new_acc)*(dt/2.0) +        # first compute new_vel
                   (@jerk - new_jerk)*(dt*dt/12.0)   # since new_vel is used
  new_pos = @pos + (@vel + new_vel)*(dt/2.0) +        # to compute new_pos
                   (@acc - new_acc)*(dt*dt/12.0)
  @pos = new_pos
  @vel = new_vel
  @acc = new_acc
  @jerk = new_jerk
  @pred_pos = @pos
  @pred_vel = @vel
  @time = t
  @next_time = @time + collision_time_scale(ba) * dt_param
end

def collision_time_scale(body_array)
  time_scale_sq = VERY_LARGE_NUMBER
  body_array.each do |b|
    unless b == self
      r = b.pred_pos - @pred_pos
      v = b.pred_vel - @pred_vel
      r2 = r*r
      v2 = v*v
      estimate_sq = r2 / v2            # [distance]^2/[velocity]^2 = [time]^2
      if time_scale_sq > estimate_sq
        time_scale_sq = estimate_sq
      end
      a = (@mass + b.mass)/r2
      estimate_sq = sqrt(r2)/a         # [distance]/[acceleration] = [time]^2
      if time_scale_sq > estimate_sq
        time_scale_sq = estimate_sq
      end
    end
  end
```

```
    sqrt(time_scale_sq)
  end

  def get_acc_and_jerk(body_array)
    a = j = @pos*0                          # null vectors of the correct length
    body_array.each do |b|
      unless b == self
        r = b.pred_pos - @pred_pos
        r2 = r*r
        r3 = r2*sqrt(r2)
        v = b.pred_vel - @pred_vel
        a += r*(b.mass/r3)
        j += (v-r*(3*(r*v)/r2))*(b.mass/r3)
      end
    end
    [a, j]
  end

  def ekin                          # kinetic energy
    0.5*@mass*(@vel*@vel)
  end

  def epot(body_array)              # potential energy
    p = 0
    body_array.each do |b|
      unless b == self
        r = b.pos - @pos
        p += -@mass*b.mass/sqrt(r*r)
      end
    end
    p
  end

end
```

2.2 A Familiar Part

Bob: The second half of the `Body` class is familiar, but with some subtle differences. Take the `collision_time_scale` for example. There are only two lines that are not exactly the same as we had before, line four and five of the body of the definition. In the shared time step case, we had:

```
        r = b.pos - @pos
        v = b.vel - @vel
```

where in the individual time step case, we have instead:

```
r = b.pred_pos - @pred_pos
v = b.pred_vel - @pred_vel
```

This is exactly what is needed here: in order to push a particle forward, we predict the position at it latest sales date time `@next_time`, and we ask all other particles to predict where they will be at that very same time. We can then compute the differences in position and velocity at that time, and estimate the new time step.

Similarly, the computation of acceleration and jerk are also almost the same as they were before. Since we are here working only with the Hermite scheme, I thought it would be simpler to combine the previous methods `acc` and `jerk` into one combined method `acc_and_jerk`. And here, too, I have switched to predicted positions for both position and velocity.

Finally, the methods `ekin` and `epot` are completely unchanged.

Alice: Shouldn't they be based on predicted positions and velocity too?

Bob: No, since I invoke these diagnostics methods only after synchronizing the whole system, in which case we can use the old form of the methods verbatim. We'll come to that. First let's look at the top of the `Body` class, where the most important differences reside.

2.3 Predicting and Correcting

Alice: Ah, there at the very top is our friend `autonomous_step`, which invited us to come here in the first place.

Bob: Yes. This method takes as a first argument the reference to the array of all bodies in the N-body system. It needs that as a handle to reach all other particles. The second parameter is the factor with which we multiply the time scale to obtain the new time step. That part is the same as before, the only difference being that each particle now computes its own time scale; the parameter `dt_param` is a constant, an overall scaling factor that the user can set at the beginning of the run.

Alice: And all that `autonomous_step` does, is take a step from the present time `time` to the `@next_time`.

Bob: Exactly. And you see in the next definition why I have added that extra time argument in the middle: when we need to synchronice all particles, we can no longer give them the luxury to step autonomously. Instead of letting them

choose their own next time, we need to force them to all converge at a fixed target time `t`, the middle argument of the method `forced_time_step`. The only difference is that in the autonomous case, a particle can determine itself where it wants to force itself to go to, so to speak.

Alice: So to speak. Trying to explain code in words is always a tricky business, but with the code in hand, I see what you mean. Okay, let's move to where the real work is being done, in the method `take_one_step`. For starters, you ask all other particles to predict their position to time `t`.

Bob: All other particles, yes, and also the particle itself that is doing the asking. Note that I have written `ba.each`, without taking exception to `self`. So in one stroke I push all particles virtually to the target time `t`, self as well as others.

Alice: And then you correct something, I presume, in `correct_step`. What is it that you correct?

Bob: It is in `correct_step` that we finally encounter the full Hermite algorithm. Let's compare this with the shared time step case. There we sent the Hermite algorithm to individual particles using the following method on the `NBody` level:

```
def hermite
  calc(" @old_pos = @pos ")
  calc(" @old_vel = @vel ")
  calc(" @old_acc = acc(ba) ")
  calc(" @old_jerk = jerk(ba) ")
  calc(" @pos += @vel*dt + @old_acc*(dt*dt/2.0) + @old_jerk*(dt*dt*dt/6.0) ")
  calc(" @vel += @old_acc*dt + @old_jerk*(dt*dt/2.0) ")
  calc(" @vel = @old_vel + (@old_acc + acc(ba))*(dt/2.0) +
                   (@old_jerk - jerk(ba))*(dt*dt/12.0) ")
  calc(" @pos = @old_pos + (@old_vel + vel)*(dt/2.0) +
                   (@old_acc - acc(ba))*(dt*dt/12.0) ")
end
```

Lines five and six of the body of this method do exactly what is now done in `predict_step`, which I will show here again:

```
def predict_step(t)
  if t > @next_time
    STDERR.print "predict_step: t = ", t, " > @next_time = "
    STDERR.print @next_time, "\n"
    exit
  end
  dt = t - @time
  @pred_pos = @pos + @vel*dt + @acc*(dt*dt/2.0) + @jerk*(dt*dt*dt/6.0)
  @pred_vel = @vel + @acc*dt + @jerk*(dt*dt/2.0)
end
```

And the next two lines in the old code do exactly what is now done in `correct_step` in the new code:

```
def correct_step(ba, t, dt_param)
  dt = t - @time
  new_acc, new_jerk = get_acc_and_jerk(ba)
  new_vel = @vel + (@acc + new_acc)*(dt/2.0) +        # first compute new_vel
                   (@jerk - new_jerk)*(dt*dt/12.0)   # since new_vel is used
  new_pos = @pos + (@vel + new_vel)*(dt/2.0) +        # to compute new_pos
                   (@acc - new_acc)*(dt*dt/12.0)
  @pos = new_pos
  @vel = new_vel
  @acc = new_acc
  @jerk = new_jerk
  @pred_pos = @pos
  @pred_vel = @vel
  @time = t
  @next_time = @time + collision_time_scale(ba) * dt_param
end
```

Alice: I see. It is beginning to make sense for me now. And I am also starting to remember how we derived the Hermite algorithm in the first place. In the old code, two steps were involved. Starting from a given position, we made a tentative move to a new position, in order to calculate the new acceleration and jerk. Then we combined that result with the knowledge of the old acceleration and jerk, and together we were able to reconstruct the higher derivatives snap and crackle at the start of the step. Finally, we built a Taylor series using the acceleration, jerk, snap and crackle at the starting time, to make a more accurate determination of the ending position.

Bob: Yes, that is exactly what happens. And what you called here a tentative step is what I call `predict_step`, while what you called a more accurate determination I call `correct_step`.

2.4 Shifting Perspective

Alice: That all makes a lot of sense. And the next four lines, following the actual correction part of `correct_step` are the same as the first four lines in our old `hermite` method. The only difference is one of notation: in the `nbody_sh1.rb` we compared the old values with the current values, while here in `nbody_ind1.rb` we compare the current values with the new values.

Bob: Yes, I hadn't quite realized that I had implicitly shifted my perspective by one time step. Why did I do that? Ah, of course. Interesting! In the past we were dealing with shared time steps. So as soon as we were ready to move, we could put ourselves one step ahead, so to speak, in our imagination, while looking back on the previous values as the *old* values.

However, in the individual time step case, most of the time we are moving forward in time only in a virtual way: when particles predict their position, in order to help another particle find its way, they don't really move themselves. Since they stay at their old place, it makes more sense to called the old place the current place, current for each particle that is, and to call the predicted time the *new* time.

Alice: So your shift in perspective makes sense. To predict, you have to look ahead. But if you're walking in lock step, you move with the whole group, and all you can do is look back.

Bob: You may be stretching the analogy here, but if it helps you remember the notation, fine! Now the remaining two lines of `correct_step` update the two time variables that belong to our particle: the new current time `@time`, for which `@pos` and `@vel` are actually calculated, is `t`, and the new `@next_time` is found by calculating the next time step, as we have already seen.

Chapter 3

The NBody Class

3.1 Code Listing

Alice: I think I understand now how the `Body` class works. Could you also print out the whole `NBody` class?

Bob: Sure. It's not very long, about the same size as the `Body` class. I keep being surprised at how compact Ruby code is.

```
class NBody

  def startup(dt_param)
    @e0 = ekin + epot
    @body.each do |b|
      b.pred_pos = b.pos
      b.pred_vel = b.vel
    end
    @body.each do |b|
      b.acc, b.jerk = b.get_acc_and_jerk(@body)
      b.time = @time
      b.next_time = @time + b.collision_time_scale(@body) * dt_param
    end
  end

  def evolve(c)
    @nsteps = 0
    startup(c.dt_param)
    write_diagnostics
    t_dia = @time + c.dt_dia
    t_out = @time + c.dt_out
```

```
  t_end = @time + c.dt_end
  acs_write if c.init_out

  while @time < t_end
    np = find_next_particle
    @time = np.next_time
    if (@time < t_end)
      np.autonomous_step(@body, c.dt_param)
      @nsteps += 1
    end
    if @time >= t_dia
      sync(t_dia, c.dt_param)
      @nsteps += @body.size
      write_diagnostics
      t_dia += c.dt_dia
    end
    if @time >= t_out
      sync(t_out, c.dt_param)    # we are now syncing twice, if t_dia = t_out
      @nsteps += @body.size
      acs_write
      t_out += c.dt_out
    end
  end
end

def find_next_particle
  next_time = VERY_LARGE_NUMBER
  next_particle = nil
  @body.each do |b|
    if next_time > b.next_time
      next_time = b.next_time
      next_particle = b
    end
  end
  next_particle
end

def sync(t, dt_param)
  @body.each{|b| b.forced_step(@body, t, dt_param)}
  @time = t
end

def ekin                          # kinetic energy
  e = 0
  @body.each{|b| e += b.ekin}
  e
```

```
  end

  def epot                          # potential energy
    e = 0
    @body.each{|b| e += b.epot(@body)}
    e/2                             # pairwise potentials were counted twice
  end

  def write_diagnostics
    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

end
```

3.2 Starting Up

Alice: We already talked about the generic case, with a call to `autonomous_step`. Now I would like to see how the whole thing starts up.

Bob: At the very bottom of the file, the `evolve` method is evoked as follows:

```
: inccode:.nbody_ind1.rb-3
```

An N-body system is read in, and all that happens is a single call to `evolve`, while the command line arguments are being passed along, just as we already saw in the previous code.

The startup function makes sure that the acceleration and jerk are being computed properly at the beginning of the integration. But in order to do so, it first has to set the predicted positions and velocities `pred_pos` and `pred_vel` to the right values, namely the values of the initial positions and velocities.

In the `Body` class we saw the following generic order: 1) we predict the positions of all particles, using `predict_step`; 2) we compute the acceleration and jerk on one particle and set the next time step value; 3) we step that particle forwards using `correct_step`.

We use the same order here, at startup, which occurs at the time `@time`, the instance variable for the `NBody` class. This is the time that has been read in from the initial snapshot; and if there is no time listed in that input file, the default time `@time = 0` is used default, since this is the time that has been assinged by the `NBody#initialize` method in file `nbody.rb`.

For startup, step 1), prediction, is trivial: starting at time `@time` we predict where all particles will be at time `@time`, in other words we don't have to do anything; we just copy the position and velocity values that were just read in.

Step 2), the calculation of acceleration and jerk and the next time step value, can only be done after step 1) has been completed for all particles. This is the reason that we need to different `each` loops in `startup`.

Step 3), correcting the positions and velocities, is not necessary at startup: no particle has been moved, so there is nothing to correct.

3.3 Finding the Next Particle

Alice: Thanks, that's very clear. Then, after startup, we enter the `while` loop in evolve, evoking `autonomous_step` for each next particle that needs to be propagated. Let's have a look at how `find_next_particle` is implemented.

```
def find_next_particle
  next_time = VERY_LARGE_NUMBER
  next_particle = nil
  @body.each do |b|
    if next_time > b.next_time
      next_time = b.next_time
      next_particle = b
    end
  end
  next_particle
end
```

Bob: Here we loop over all particles, to find the particle with the earliest next sales date `next_time`. The method `find_next_particle` then returns the body that has the smallest `next_time` value.

Alice: So time is marching forward. When we enter `find_next_particle`, the `NBODY` time `@time` has a certain value. We then are given a body `b` which has a value `b.next_time` for which we know that `b.next_time` $>$ `@time` and we also know that for all other bodies `ob` we have `ob.next_time` $>=$ `b.next_time`.

Bob: Correct.

Alice: This means that in the interval between `@time` and `b.next_time` there is nothing that needs to be done. Then we set the `NBody` time `@time` equal to `b.next_time`, and we push particle `b` forwards by one step.

Having done that, we again look for the next particle, through a new call within `evolve` to `find_next_particle`. Perhaps the time for the newly found particle is the same as that for the previous particle, in which case `@time` doesn't change; perhaps it is later, in which case `@time` is updated to the latest sales date of the new particle.

Okay, I think I got it! I just had to reconstruct the steps for myself. This is not a trivial algorithm!

Bob: As always, it is all pretty clear once you understand it clearly, but I must admit, it took me quite a while to figure it out, the first time I came across an individual time step code.

3.4 Output

Alice: We're almost there. I just have to figure out now how the particles are synchronized before you do an output, either a full particle output in terms of a snapshot, at time `t_out`, or a diagnostics output at time `t_dia`.

Let me put the need for synchronization in perspective. In the case of our constant time step code `nbody_cst1.rb`, synchronization was trivial: in typical usage, you specify output times that are multiples of the prescribed time step. And if you were to set a time step to, say, 0.01 and an output time to an incommensurable time 1/3, then the code would slightly overshoot, and give the output at time 0.34, instead of time 0.333333. We could have added an option to allow us to shrink the last time step in such a case, to reach the value 1/3 exactly, at least within double precision. However, there was no pressing reason to do so.

In the case of our shared time step code `nbody_sh1.rb`, we did build in such an option. Invoking it by default would let the code overshoot, since in general the chance is virtually zero to get commensurability between a prescribed output time and dynamically adjusted time step values. However, by invoking the code with the option `nbody_sh1.rb --exact_time`, we could force the code to halt at the exact time desired. This was essential for us in order to measure the phase space distances between different N-body systems generated in the output of different runs.

Now in the case of our individual time step code `nbody_ind1.rb`, we have no choice. We cannot afford the luxury of skipping synchronization, since in that case we would get an output of a bunch of particles, all at different times. We couldn't even compute the energy of sych a system! And this is the reason that you synchronize, for every type of output.

Bob: Yes, that is the big picture. Note, however, that my insistence on synchronization is not the only solution. I wanted to get something coded up quickly, and I knew it was essential for measuring energy conservation, so I implemented syncing by default. However, if you just want to make a dump of the system, to allow future restarts, there is no reason to insist on all the particles having their state set at the same time.

Alice: Ah, yes, of course, I had forgotten that. Indeed, if you *do* synchronize, you are guaranteed *not* to follow the same trajectory when you restart from an earlier output. We showed this in the previous volume, in the case of a shared time step code.

Well, what do you think, shouldn't we implement such a 'ragged' output, with the orbits of different particles extending in time to different distances?

Bob: Ragged?

Alice: I'm trying to visualize what is going on. I can see a picture in front of me, in spacetime, with each particle following a worldline, where some worldlines are extended further than others, like a ragged carpet where the different strands have different lengths.

3.5 Wordlines and Worldpoints

Bob: You always manage to complicate things! As soon as you understand how something works in three dimensions, you add another dimension for good measure!

Alice: I'm serious, though. I do think it may be an interesting way of looking at the evolution of an N-body system. Normally we associate a spacetime view with special relativity, where particles follow worldlines in space and time, but there is no reason not to use such a picture in classical mechanics as well.

Bob: So instead of talking about the N-body problem, from now on you want to talk about the N-worldline problem?

Alice: Well, why not?

Bob: I know why not. Nobody will understand what you're talking about.

Alice: We'll worry about that later. Why not pursue this idea for a moment. Rather than thinking in terms of a bunch of separate bodies, as isolated point masses in 3-D, it may be more natural to think of them as strings in 4-D. Such a picture could actually be helpful in visualizing the basic idea of an individual time step code.

Bob: A string theory for classical gravity?

Alice: If you like! But without Calabi-Yau manifolds, don't worry! We'll try to keep it simple.

Bob: What are Calabi-Yau manifolds?

Alice: Something quite a bit more difficult to visualize than individual time step algorithms. Forgot about that.

Bob: With pleasure; life is already complicated enough. So instead of having N particles in space, you want us to look at N worldlines in spacetime. And each worldline has a number of knots, one at each point where we compute the positions and velocities. Do you want to call those knots worldpoints?

Alice: That would be a natural term, although I haven't heard it before. But why not? In special relativity textbooks, these points in 4-D are called events, since they are associated with a particular time and place. However, the word worldpoint is more descriptive, in that they convey the notion that they belong to a particular worldline. Different worldpoints on the same worldline then share the same `body_id` identity, but are different in having different values for the time `@time`

Bob: Taking a time step is then a temporal operation, while syncing for output is a spatial operation?

Alice: You're getting it! See how natural it is to view the N-body problem in four dimensions?

Bob: The N-worldline problem.

Alice: I'm happy to keep calling it the N-body problem, not to worry. And I do think this way of looking at things has mileage. Synchronization means constructing a cross section of the full bundle of worldline with a hypersurface at a constant time.

Bob: And what is the mileage in being abstract to that degree?

Alice: Let me think about it a bit more, and continue tomorrow.

Bob: Good idea: with all this talking about spacetime, we forgot the time. It's getting late.

Alice: Ah, but before we forget, did you test your individual time step scheme, to make sure it worked?

Bob: I did, but just to check, and to show you it's in good shape, let's do a quick run.

3.6 Verification

Alice: Let's take all three codes we have now, for constant, shared, and individual time steps. And let's give them a really good workout. Shall we shoot for an accuracy of 10^{-12}? If we get too close to machine accuracy, it may be more difficult to interpret our results.

Bob: Fine with me. Let's start with the same initial conditions:

```
|gravity> kali mkplummer.rb -n 4 -s 1 | kali nbody_set_id.rb > test.in
==> Plummer's Model Builder <==
Number of particles: N = 4
pseudorandom number seed given: 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
             actual seed used: 1
==> Takes an N-body system, and gives each body a unique ID <==
value of @body_id for 1st body: n = 1
Floating point precision: precision = 16
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
```

We will have to try a few different time step accuracy parameters, to get roughly the right relative energy conservation. This may take a few repetitions. I'll start with constant time steps:

```
|gravity> kali nbody_cst1.rb -t 1 < test.in > tmpc.out
==> Constant Time Step Code <==
Integration method: method = hermite
Softening length: eps = 0.0
Time step size: dt = 0.001
Interval between diagnostics output: dt_dia = 1.0
Time interval between snapshot output: dt_out = 1.0
Duration of the integration: t = 1.0
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
at time t = 0, after 0 steps :
  E_kin = 0.25 , E_pot =  -0.5 , E_tot = -0.25
             E_tot - E_init = 0
  (E_tot - E_init) / E_init = -0
at time t = 1, after 1000 steps :
  E_kin = 0.248 , E_pot =  -0.613 , E_tot = -0.365
             E_tot - E_init = -0.115
  (E_tot - E_init) / E_init = 0.461
```

```
|gravity> kali nbody_cst1.rb -t 1 -c 0.0001 < test.in > tmpc.out
==> Constant Time Step Code <==
Integration method: method = hermite
Softening length: eps = 0.0
Time step size: dt = 0.0001
Interval between diagnostics output: dt_dia = 1.0
Time interval between snapshot output: dt_out = 1.0
Duration of the integration: t = 1.0
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
at time t = 0, after 0 steps :
  E_kin = 0.25 , E_pot =  -0.5 , E_tot = -0.25
             E_tot - E_init = 0
  (E_tot - E_init) / E_init = -0
at time t = 1, after 10000 steps :
  E_kin = 0.313 , E_pot =  -0.563 , E_tot = -0.25
             E_tot - E_init = -6.62e-07
  (E_tot - E_init) / E_init = 2.65e-06
```

A bit better than needed, but this will do. Now for shared time steps:

```
|gravity> kali nbody_sh1.rb -t 1 --exact_time < test.in > tmps.out
==> Shared Time Step Code <==
Integration method: method = hermite
Parameter to determine time step size: dt_param = 0.01
Interval between diagnostics output: dt_dia = 1.0
Time interval between snapshot output: dt_out = 1.0
Duration of the integration: t = 1.0
Force all outputs to occur at the exact times
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
at time t = 0, after 0 steps :
  E_kin = 0.25 , E_pot =  -0.5 , E_tot = -0.25
             E_tot - E_init = 0
  (E_tot - E_init) / E_init = -0
at time t = 1, after 1927 steps :
  E_kin = 0.313 , E_pot =  -0.563 , E_tot = -0.25
             E_tot - E_init = -2.33e-10
  (E_tot - E_init) / E_init = 9.32e-10
|gravity> kali nbody_sh1.rb -c 0.003 -t 1 --exact_time < test.in > tmps.out
==> Shared Time Step Code <==
```

```
Integration method: method = hermite
Parameter to determine time step size: dt_param = 0.003
Interval between diagnostics output: dt_dia = 1.0
Time interval between snapshot output: dt_out = 1.0
Duration of the integration: t = 1.0
Force all outputs to occur at the exact times
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
at time t = 0, after 0 steps :
  E_kin = 0.25 , E_pot =  -0.5 , E_tot = -0.25
            E_tot - E_init = 0
  (E_tot - E_init) / E_init = -0
at time t = 1, after 6424 steps :
  E_kin = 0.313 , E_pot =  -0.563 , E_tot = -0.25
            E_tot - E_init = -6.12e-13
  (E_tot - E_init) / E_init = 2.45e-12
|gravity> kali nbody_sh1.rb -c 0.002 -t 1 --exact_time < test.in > tmps.out
==> Shared Time Step Code <==
Integration method: method = hermite
Parameter to determine time step size: dt_param = 0.002
Interval between diagnostics output: dt_dia = 1.0
Time interval between snapshot output: dt_out = 1.0
Duration of the integration: t = 1.0
Force all outputs to occur at the exact times
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
at time t = 0, after 0 steps :
  E_kin = 0.25 , E_pot =  -0.5 , E_tot = -0.25
            E_tot - E_init = 0
  (E_tot - E_init) / E_init = -0
at time t = 1, after 9635 steps :
  E_kin = 0.313 , E_pot =  -0.563 , E_tot = -0.25
            E_tot - E_init = -1.24e-13
  (E_tot - E_init) / E_init = 4.96e-13
```

Okay, that one's below our 10^{-12} threshold too. Now for individual time steps:

```
|gravity> kali nbody_ind1.rb -t 1 < test.in > tmpi.out
==> Individual Time Step Hermite Code <==
Parameter to determine time step size: dt_param = 0.01
```

```
Interval between diagnostics output: dt_dia = 1.0
Time interval between snapshot output: dt_out = 1.0
Duration of the integration: t = 1.0
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
at time t = 0, after 0 steps :
  E_kin = 0.25 , E_pot =  -0.5 , E_tot = -0.25
             E_tot - E_init = 0
  (E_tot - E_init) / E_init = -0
at time t = 1, after 4257 steps :
  E_kin = 0.313 , E_pot =  -0.563 , E_tot = -0.25
             E_tot - E_init = 7.24e-10
  (E_tot - E_init) / E_init = -2.9e-09
|gravity> kali nbody_ind1.rb -c 0.002 -t 1 < test.in > tmpi.out
==> Individual Time Step Hermite Code <==
Parameter to determine time step size: dt_param = 0.002
Interval between diagnostics output: dt_dia = 1.0
Time interval between snapshot output: dt_out = 1.0
Duration of the integration: t = 1.0
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
at time t = 0, after 0 steps :
  E_kin = 0.25 , E_pot =  -0.5 , E_tot = -0.25
             E_tot - E_init = 0
  (E_tot - E_init) / E_init = -0
at time t = 1, after 21282 steps :
  E_kin = 0.313 , E_pot =  -0.563 , E_tot = -0.25
             E_tot - E_init = -9.77e-13
  (E_tot - E_init) / E_init = 3.91e-12
```

Perfect, just on the mark.

Alice: And now let's check whether the results are really similar:

```
|gravity> cat tmpc.out tmps.out | kali nbody_diff.rb
==> 6N-dimensional phase space distance between two N-body systems <==
Floating point precision: precision = 2
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
```

```
6N-dim. phase space dist. for two 4-body systems:   6.8757897588114396e-06
|gravity> cat tmps.out tmpi.out | kali nbody_diff.rb
==> 6N-dimensional phase space distance between two N-body systems <==
Floating point precision: precision = 2
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
6N-dim. phase space dist. for two 4-body systems:   3.5201108698382295e-11
```

Bob: There are larger than I would have expected. Especially the constant time step run is off by an amount that is two orders of magnitude larger than the relative energy conservation.

Alice: Yes, that is a bit much. The good news is that the shared time step code and the individual time step code give results that correspond to about 10^{-11}. That may not be so bad; there is no guarantee that the phase space distance should give the same result as the energy error.

Bob: Let me run the constant time step code with a two times smaller time step size. That should increase the accuracy by more than an order of magnitude, bringing all three codes more in line:

```
|gravity> kali nbody_cst1.rb -t 1 -c 0.00005 < test.in > tmpc.out
==> Constant Time Step Code <==
Integration method: method = hermite
Softening length: eps = 0.0
Time step size: dt = 5.0e-05
Interval between diagnostics output: dt_dia = 1.0
Time interval between snapshot output: dt_out = 1.0
Duration of the integration: t = 1.0
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
at time t = 0, after 0 steps :
  E_kin = 0.25 , E_pot =  -0.5 , E_tot = -0.25
            E_tot - E_init = 0
  (E_tot - E_init) / E_init = -0
at time t = 1, after 20000 steps :
  E_kin = 0.313 , E_pot =  -0.563 , E_tot = -0.25
            E_tot - E_init = -1.35e-08
  (E_tot - E_init) / E_init = 5.4e-08
```

And let's now compare all three pairs:

```
|gravity> cat tmpc.out tmps.out | kali nbody_diff.rb
==> 6N-dimensional phase space distance between two N-body systems <==
Floating point precision: precision = 2
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
6N-dim. phase space dist. for two 4-body systems:   3.5758322798815124e-07
|gravity> cat tmpc.out tmpi.out | kali nbody_diff.rb
==> 6N-dimensional phase space distance between two N-body systems <==
Floating point precision: precision = 2
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
6N-dim. phase space dist. for two 4-body systems:   3.5757721030515277e-07
|gravity> cat tmps.out tmpi.out | kali nbody_diff.rb
==> 6N-dimensional phase space distance between two N-body systems <==
Floating point precision: precision = 2
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
6N-dim. phase space dist. for two 4-body systems:   3.5201108698382295e-11
```

Alice: Now they are indeed all comparable. Good! I am beginning to believe that we may have done the right thing.

Chapter 4

Spacetime

4.1 Mileage

Bob: Yesterday you mentioned that you could see mileage in taking a more abstract, four-dimensional perspective on the N-body problem.

Alice: And you looked skeptical.

Bob: Because I am. What can be the mileage in such a shift, given that we are not dealing with relativistic velocities?

Alice: For one thing, a spacetime view puts our discussion of restarting a run from an output file in a clearer perspective. If we create a new class `Worldline` for each world line, we can rename our current class `Body` to `Worldpoint` and instead of `NBody` we can talk about `World`, in order to point to the whole system.

For quite a while, I've been unhappy with the big gap between the bottom-level `Body` objects and the whole `NBody` system, as a single top-level objects. With a `Worldline` class in between, we can separate many issues in a more fine-grained way.

Look at it this way. If you start in four dimensions, identities of objects are related to world lines, while specific values of positions and velocities are related to worldpoints. But as soon as you project that spacetime view into a single space view, you wind up with point particles that have position and velocity and identity, and these three all look the same: the original distinction between lines and points is lost.

In the three dimensional view, we are dealing with moving particles. Each particle has an identity that has to be carried with it while its position and velocity are changing. And with individual time steps, it becomes rather complicated and potentially very confusing, to keep track of which particle is where at what time, and to make sure that particles interact with each other in the right way.

Bob: Can you be more concrete?

Alice: In your individual time step code `nbody_ind1.rb`, the main engine `evolve` has to juggle the regular phase of evolving particles with the occasional interruptions of output demand, which require synchronizing all particles. As we have seen, this means that a change in diagnostics frequency has physical repercussions: it means that our particles will be forced to move on slightly different orbits.

This is a Bad Thing.

It would be *much* better to implement a form of diagnostics reporting that is totally decoupled from the way particles move. If we can make orbit integration and diagnostics reporting truly modular, without the one influencing the other, I would be much happier.

4.2 Eras and Snapshots

Bob: And introducing the concept of a `Worldline` could do that, you think?

Alice: I think so. We could leave all diagnostics, together with all other output issues to the top level `World` class. The evolution could take place at a lower level.

Bob: At the `Worldline` level?

Alice: No, a `Worldline` class contains only information about a single particle. We need yet again something intermediate, this time in between `Worldline` and `World`. Something like a block of time during which all particles can move forward without worrying about output issues. If we save all the information about all steps that are taking during this time, in other words if we save all `Worldpoint` information, we can then deal with output issues at the end of that block of time.

Bob: Putting a `Worldblock` class in between `World` and `Worldpoint`?

Alice: Yes, but we need a better name. How about a period?

Bob: A dynasty?

Alice: Effectively, yes. It is rather recent that people are counting time in terms of a universal and continuous year count. Most cultures used to talk in terms of the fourth year of the reign of king so-and-so.

Bob: But `Worlddynasty` doesn't roll of the tongue.

Alice: How about calling it an era? That's about the shortest term I can think of.

Bob: Can't make it much shorter. A `Worldera` class?

Alice: That sounds better already. So instead of dealing with two levels, of `Body`

and `NBody` we will now have four levels: `Worldpoint`, `Worldline`, `Worldera`, and finally just `World`.

Bob: But what about output? I don't see any natural place yet to synchronize particles in this hierarchy.

Alice: Good point. The four world-related classes are all intrinsically temporal constructs. We need a separate class for spatial information. This class should contain the results of taking a hypersurface cut through a bunch of world lines.

Bob: You mean, the result of taking a snapshot of the system?

Alice: Yes, that's a good name! Let's call it a `Worldsnapshot` class. By using the word 'world' everywhere, we make it clear that we're dealing with a coherent system of concepts.

Bob: Before you introduce even more classes, shall we start with a toy code, to see whether your grand vision makes any sense, in practice?

Alice: I'd love to! It shouldn't be too hard to rewrite your `nbody_ind1.rb` into `World` form.

Bob: I'm not at all convinced that going from two to five classes will simplify life, but I must admit, your ideas sound intriguing enough to try it out.

Before getting started, let's make sure this time that we obey Ruby's convention of using MixedCaseNotation from the beginning: let's call our classes `World`, `WorldSnapshot`, `WorldEra`, `WorldLine`, and `WorldPoint`.

Alice: MyPleasure.

4.3 Intermezzo

From this point on, Alice and Bob spent quite a bit of time coding up an individual time step algorithm, using their five new classes. As one can imagine, they ran into all kinds of problems, and went through many a debug session. Finally, after several days, it all worked. We will join them now, while they look at their first finished product, in file `world1.rb`. This will be the first of a series of such implementations, and the detailed layout will change quite a bit. However, the current code contains all the basic ingredients for a worldline-based N-body code.

Bob: That was quite a lot of work. The new file `world1.rb` is a complete overhaul of what we had in `nbody_ind1.rb`. It is twice a long, too.

Alice: But it is a lot more structured, and I'm sure it will be much easier to extend to more complicated situations. Besides, it has significantly more functionality.

Bob: Before we congratulate ourselves too much, we should make sure that we get the same answers as before. Let us apply our standard test again, using the

by now familiar initial conditions for our 4-particle system:

```
|gravity> kali mkplummer.rb -n 4 -s 1 | kali nbody_set_id.rb > test.in
==> Plummer's Model Builder <==
Number of particles: N = 4
pseudorandom number seed given: 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
             actual seed used: 1
==> Takes an N-body system, and gives each body a unique ID <==
value of @body_id for 1st body: n = 1
Floating point precision: precision = 16
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
```

Here is our comparison run for the older code:

```
|gravity> kali nbody_ind1.rb -t 1 -c 0.001 < test.in > tmpi.out
==> Individual Time Step Hermite Code <==
Parameter to determine time step size: dt_param = 0.001
Interval between diagnostics output: dt_dia = 1.0
Time interval between snapshot output: dt_out = 1.0
Duration of the integration: t = 1.0
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
at time t = 0, after 0 steps :
  E_kin = 0.25 , E_pot =  -0.5 , E_tot = -0.25
             E_tot - E_init = 0
  (E_tot - E_init) / E_init = -0
at time t = 1, after 42565 steps :
  E_kin = 0.313 , E_pot =  -0.563 , E_tot = -0.25
             E_tot - E_init = -1.42e-12
  (E_tot - E_init) / E_init = 5.68e-12
```

And here is what our new code gives:

```
|gravity> kali world1.rb -t 1 -c 0.001 < test.in > tmpw.out
==> Individual Time Step, Individual Integration Scheme Code <==
Determines the time step size: dt_param = 0.001
Duration of an era: dt_era = 0.01
Maximum time step (units dt_era): dt_max_param = 1.0
Diagnostics output interval: dt_dia = 1.0
Snapshot output interval: dt_out = 1.0
Duration of the integration: t = 1.0
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
at time t = 0 (from interpolation after 0 steps to time 0):
    E_kin = 0.25 ,     E_pot =  -0.5 ,      E_tot = -0.25
       E_tot - E_init = 0
        (E_tot - E_init) / E_init = -0
at time t = 1 (from interpolation after 42578 steps to time 1):
    E_kin = 0.313 ,     E_pot =  -0.563 ,      E_tot = -0.25
       E_tot - E_init = -1.05e-12
        (E_tot - E_init) / E_init = 4.21e-12
```

Aha, at least the number of time steps is very similar, and so is the total energy error. Here is the distance in phase space

---

```
|gravity> cat tmpi.out tmpw.out | kali nbody_diff.rb
==> 6N-dimensional phase space distance between two N-body systems <==
Floating point precision: precision = 2
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
6N-dim. phase space dist. for two 4-body systems:   4.2386924923927053e-12
```

4.4 Almost Perfect Scaling

Alice: I'm glad to see that: the distance is comparable to the energy error. In fact, the correspondance is so good, that I wonder whether we've been a bit lucky. I'd feel more comfortable if we would redo the run with somewhat lower accuracy, just to check whether the phase space distance will become larger.

Bob: Why not. When we take time steps that are ten times longer, we get:

```
|gravity> kali nbody_ind1.rb -t 1 -c 0.01 < test.in > tmpi.out
==> Individual Time Step Hermite Code <==
Parameter to determine time step size: dt_param = 0.01
Interval between diagnostics output: dt_dia = 1.0
Time interval between snapshot output: dt_out = 1.0
Duration of the integration: t = 1.0
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
at time t = 0, after 0 steps :
  E_kin = 0.25 , E_pot =  -0.5 , E_tot = -0.25
            E_tot - E_init = 0
  (E_tot - E_init) / E_init = -0
at time t = 1, after 4257 steps :
  E_kin = 0.313 , E_pot =  -0.563 , E_tot = -0.25
            E_tot - E_init = 7.24e-10
  (E_tot - E_init) / E_init = -2.9e-09
```

I bet the new code will show a similar error:

```
|gravity> kali world1.rb -t 1 -c 0.01 < test.in > tmpw.out
==> Individual Time Step, Individual Integration Scheme Code <==
Determines the time step size: dt_param = 0.01
Duration of an era: dt_era = 0.01
Maximum time step (units dt_era): dt_max_param = 1.0
Diagnostics output interval: dt_dia = 1.0
Snapshot output interval: dt_out = 1.0
Duration of the integration: t = 1.0
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
at time t = 0 (from interpolation after 0 steps to time 0):
    E_kin = 0.25 ,     E_pot =  -0.5 ,      E_tot = -0.25
       E_tot - E_init = 0
        (E_tot - E_init) / E_init = -0
at time t = 1 (from interpolation after 4291 steps to time 1):
    E_kin = 0.313 ,     E_pot =  -0.563 ,      E_tot = -0.25
       E_tot - E_init = -1.98e-10
        (E_tot - E_init) / E_init = 7.92e-10
```

Alice: And so it does. So now let us see whether the phase space difference is also be comparable:

```
|gravity> cat tmpi.out tmpw.out | kali nbody_diff.rb
==> 6N-dimensional phase space distance between two N-body systems <==
Floating point precision: precision = 2
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
6N-dim. phase space dist. for two 4-body systems:   1.0927259036948475e-08
```

Bob: Which it is. Okay, we seem to have done something right.

Chapter 5

The `World` Class

5.1 Of Forests and Trees

Alice: You know, I'd like to go over the whole code once again, now that we make sure it works. I feel I can't see the forest for the trees. I'd like to gain a good overview.

Bob: That's a good idea. How shall we do this, from the beginning to the end, or from the end to the beginning?

Alice: I prefer to go top-down. Given the way we have written the code, this means starting at the end, or at least close to the end, with the `World` class.

Bob: Okay, let's print it out.

```
class World

  def evolve(c)
    while @era.start_time < @t_end
      @new_era, dn = @era.evolve(c.dt_era, @dt_max, c.shared_flag)
      @nsteps += dn
      @time = @era.end_time
      while @t_dia <= @era.end_time and @t_dia <= @t_end
        @era.write_diagnostics(@t_dia, @nsteps, @initial_energy)
        @t_dia += c.dt_dia
      end
      while @t_out <= @era.end_time and @t_out <= @t_end
        output(c)
        @t_out += c.dt_out
      end
      @old_era = @era
      @era = @new_era
```

```
  end
end

def output(c)
  if c.world_output_flag
    acs_write($stdout, false, c.precision, c.add_indent)
  else
    @era.take_snapshot(@t_out).acs_write($stdout, true,
                                         c.precision, c.add_indent)
  end
end

def setup_from_world(c)
  init_output(c)
  @t_out += c.dt_out
  @t_end += c.dt_end
  @dt_max = c.dt_era * c.dt_max_param
  @new_era = @era.next_era(c.dt_era)
  @old_era = @era
  @era = @new_era
end

def setup_from_snapshot(ss, c)
  @era = WorldEra.new
  @era.setup_from_snapshot(ss, c.dt_param, c.dt_era)
  @nsteps = 0
  @dt_max = c.dt_era * c.dt_max_param
  @initial_energy = @era.startup_and_report_energy(@dt_max)
  @time = @era.start_time
  @t_out = @time
  @t_dia = @time
  @t_end = @time
  init_output(c)
  @t_out += c.dt_out
  @t_dia += c.dt_dia
  @t_end += c.dt_end
end

def init_output(c)
  @era.write_diagnostics(@time, @nsteps, @initial_energy, true)
  if c.init_out
    if c.world_output_flag
      acs_write($stdout, false, c.precision, c.add_indent)
    else
      @era.take_snapshot(@t_out).acs_write($stdout, true,
                                           c.precision, c.add_indent)
```

```
      end
    end
  end

  def World.admit(file, c)
    object = acs_read([self, WorldSnapshot], file)
    if object.class == self
      object.setup_from_world(c)
      return object
    elsif object.class == WorldSnapshot
      w = World.new
      w.setup_from_snapshot(object, c) if object.class == WorldSnapshot
      return w
    else
      raise "#{object.class} not recognized"
    end
  end

end
```

5.2 Admittance

Alice: We'd better start at the gate: at the very bottom of our code, we can see how we make our entrance:

```
clop = parse_command_line(options_text)
World.admit($stdin, clop).evolve(clop)
```

The method `World.admit` can be found at the end of our `World` class definition. Unlike most methods we have written so far, this particular function is *not* an instance method, but a class method, as indicated by the fact that the Class name is added here in front of it.

Bob: And we need this feature, otherwise we couldn't read in the data for a `World` object. Just like the general class method `new`, the class method `admit` can operate by itself, without having to be invoked from an existing `World` object.

Alice: The first thing this method does is to invoke our acs_read function, which returns an object called `object`, that can either be a `World` object or a `Worldsnapshot` object.

In the first case, we are dealing with a complete output dump from a previous run. Any invocation of `world1.rb` gives us the choice to produce a snapshot or

a complete World dump, as we will see in a moment. In the case of a dump, the class of the object is World, and since we're looking at a class method of class World, self here is the same as World. In that case, the if statement evaluates to true, and we invoke the method object.setup_from_world, in other words, an instance method of class World, with the name setup_from_world.

Now I'm puzzled. That last function sets up all kind of things, it seems, performing a variety of initializations. But after that, World.admit just returns object. What happens with that object, of type World?

Bob: Ah, the compactness of Ruby notation! On the same line where we invoke World.admit, at the very end of the code, we then invoke the member function evolve of the object that is returned.

Alice: Oh, yes, of course. Okay, what happens if we start with a snapshot output? For example, we could start with a Plummer model input file. Or we could take the output from a previous run, if we had decided to do an output of a WorldSnapshot object. Back to World.admit.

Bob: Wait a minute. If we start with a Plummer model, the class of the object that has been read in will be NBody, not WorldSnapshot. Wouldn't the elsif statement in World.admit fail?

Alice: No, it won't. That's the magic that we've built into acsio.rb, remember? In the first line, acs_read tries to recognize either a World object or a WorldSnapshot object. Now an NBody object certainly has nothing to do with the World class. But We have defined the WorldSnapshot class as a subclass of NBody. When acs_read sees this, it opens the gate for the Nbody object, and it then reads it in *as* a WorldSnapshot object.

Bob: Ah yes, how clever we were, when we wrote that! Almost too clever, I'm afraid. But it does all seem to work. Okay, so even in the case of a completely fresh Plummer model input file, the elsif statement will evaluate as true. Good. In that case we create a new World object w, which we return so that it can be evolved.

Alice: Yes, but before doing so, there is quite a bit of work done in the method setup_from_snapshot, more than in the World case.

Bob: The reason is that a World object is already all set to go, since it came from a previous integration with the same code world1.rb. If we start from a general NBody object, we'll have to do more book keeping. For example, we have to compute the initial energy. You can see the call to startup_and_report_energy, a method of the WorldEra class, in the middle of setup_from_snapshot. In contrast, if we start from a World object, we can safely assume that the initial energy value has already been stored there.

5.3 Output Choice

Alice: I get the picture now; it all comes back. And corresponding to the two ways of reading input, there are also two ways of writing output. In the case of input, we did not need any command line option, since we would recognize from the input offered what type of file we were dealing with. But for doing output, a decision has to be made. This we do with the option, well, . . . I've forgotten which option. Let's ask Ruby to list, first, all the options, and then to explain what the output switch really does:

```
|gravity> kali world1.rb -h
Individual Time Step, Individual Integration Scheme Code
-c  --step_size_control: Determines the time step size [default: 0.01]
-e  --era_length: Duration of an era [default: 0.01]
-m  --max_timestep_param: Maximum time step (units dt_era) [default: 1]
-d  --diagnostics_interval: Diagnostics output interval [default: 1]
-o  --output_interval: Snapshot output interval [default: 1]
-t  --time_period: Duration of the integration [default: 10]
-i  --init_out: Output the initial snapshot
-r  --world_output: World output format, instead of snapshot
-a  --shared_timesteps: All particles share the same time step
--verbosity: Screen Output Verbosity Level [default: 1]
--acs_verbosity: ACS Output Verbosity Level [default: 1]
--precision: Floating point precision [default: 16]
--indentation: Incremental indentation [default: 2]
-h  --help: Help facility
---help: Program description (the header part of --help)
|gravity> kali world1.rb --help -r
-r  --world_output: World output format, instead of snapshot

    If this flag is set to true, each output will take the form of a
    full world dump, instead of a snapshot (the default).  Reading in
    such an world again will allow a fully accurate restart of the
    integration,  since no information is lost in the process of writing
    out and reading in, in terms of world format.
```

Ah, yes, the `-r` option, or more descriptively, the `--world_output`.

Bob: The longer option is easier to recognize, while the shorter option is easier to remember to write.

Alice: I'm not sure about that last part. Anyway, we can use either one, and the help facility can always remind us of both. And in the code, the `world_output_flag` determines the behavior of the methods `output` and `init_output` in the `World` class.

Bob: With the `-r` flag switched on, the output is really simple: `output` does nothing else but invoke our standard output method `acs_write`, writing out `self`, the current state of the `World` object itself.

Without that flag, that is, in the default case, we first have to take a snapshot, using `@era.take_snapshot`, and then we can write out that snapshot in a similar way, with a call to `acs_write`.

It's time that we have a look at `WorldEra`, to see all that happens there, behind the scenes. The only part that we can see here is we create a new era in the first line of `setup_from_snapshot`, and then ask `World#evolve` to invoke `WorldEra#evolve`, through the call to `@era.evolve` in the second line of the `evolve` method here.

Alice: Perhaps we should first look at `WorldSnapshot`, before we descend through the lineage of `WorldEra`, `WorldLine`, and `WorldPoint`.

Chapter 6

The `WorldSnapshot` Class

6.1 Three-Dimensional Slices

Alice: Although our new N-body code is written from an inherently four-dimensional perspective, we need to take three-dimensional slices in order to deal with input and output. The `WorldSnapshot` class allows us to do that, by taking a snapshot of all world lines at a given point in time. Here is the class code:

```
class WorldSnapshot < NBody

  def get_acc_and_jerk(pos, vel)
    acc = jerk = pos*0                  # null vectors of the correct length
    @body.each do |b|
      r = b.pos - pos
      r2 = r*r
      r3 = r2*sqrt(r2)
      v = b.vel - vel
      acc += b.mass*r/r3
      jerk += b.mass*(v-3*(r*v/r2)*r)/r3
    end
    [acc, jerk]
  end

  def collision_time_scale(mass, pos, vel)
    time_scale_sq = VERY_LARGE_NUMBER             # square of time scale value
    @body.each do |b|
      r = b.pos - pos
      v = b.vel - vel
      r2 = r*r
```

```
      v2 = v*v + 1.0/VERY_LARGE_NUMBER          # always non-zero, for division
      estimate_sq = r2 / v2              # [distance]^2/[velocity]^2 = [time]^2
      if time_scale_sq > estimate_sq
        time_scale_sq = estimate_sq
      end
      a = (mass + b.mass)/r2
      estimate_sq = sqrt(r2)/a           # [distance]/[acceleration] = [time]^2
      if time_scale_sq > estimate_sq
        time_scale_sq = estimate_sq
      end
    end
    sqrt(time_scale_sq)                  # time scale value
  end

  def kinetic_energy
    e = 0
    @body.each{|b| e += b.kinetic_energy}
    e
  end

  def potential_energy
    e = 0
    @body.each{|b| e += b.potential_energy(@body)}
    e/2                                  # pairwise potentials were counted twice
  end

  def total_energy
    kinetic_energy + potential_energy
  end

  def write_diagnostics(initial_energy)
    e0 = initial_energy
    ek = kinetic_energy
    ep = potential_energy
    etot = ek + ep
    STDERR.print <<-END
    E_kin = #{sprintf("%.3g", ek)} ,\
     E_pot =  #{sprintf("%.3g", ep)} ,\
      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

end
```

6.2 No Surprises

Bob: No real surprises here. Everything looks very familiar, as if it could have been lifted straight from a previous code.

Alice: As it should, since previous codes were all written from a 3-D perspective. This is the only 3-D part in an otherwise 4-D code. The main difference really is the way that we *take* a snapshot, something that happens in methods that are part of the `WorldEra` Class. Once a `WorldSnapshot` object has been obtained, it behaves for all intents and purposes like a `NBody` object.

Bob: Can you remind me why we did not call it an `NBody` object then? I see that the `WorldSnapshot` class is a subclass of `NBody`. What's the difference?

Alice: Good question. Let's see. I believe it is mainly an alias, a new name to remind us that we are no longer dealing with a simple N-body system. Ah, yes, I remember now. By calling it a snapshot, we wanted to emphasize the fact that we are conducting any dynamics within this class.

To put it more formally, we are calculating only the right-hand side of the equations of motion, what is often called the force calculations, acceleration and jerk in the case of the Hermite scheme. The integration of the left-hand side, the actual particle pushing, is done in other classes, specifically in `WorldLine`, and orchestrated by `World` through the intermediate class `WorldEra`.

Bob: So the term `WorldSnaphot` merely is meant to stress the modularity of our approach, isn't it?

Alice: Yes, but 'merely' is an understatement. When writing large codes, modularity is more important than anything else.

Bob: I have heard you saying this before. Well, we'll see.

Chapter 7

The `WorldEra` Class

7.1 The Next Level Down

Bob: So far we have seen only the outer wrapping level, in the `World` class, where IO was scheduled and the command was given to the `WorldEra` class to do a bunch of work. The `World` class really behaves like a top executive.

Alice: Or like a secretary, letting people in and out.

Bob: Or a concierge or doorman then. We've got a rather socialistic code, if all these functions are combined in one! Well, let's see what the `WorldEra` class looks like.

```
class WorldEra

  attr_accessor  :start_time, :end_time, :worldline

  def initialize
    @worldline = []
  end

  def acc_and_jerk(wl, wp)
    take_snapshot_except(wl, wp.time).get_acc_and_jerk(wp.pos, wp.vel)
  end

  def timescale(wl, wp)
    take_snapshot_except(wl, wp.time).collision_time_scale(wp.mass,
                                                           wp.pos, wp.vel)
  end

  def startup_and_report_energy(dt_max)
```

```
  worldline.each do |wl|
    wl.startup(self, dt_max)
  end
  take_snapshot(@start_time).total_energy
end

def shortest_extrapolated_worldline
  t = VERY_LARGE_NUMBER
  wl = nil
  @worldline.each do |w|
    if t > w.worldpoint.last.next_time
      t = w.worldpoint.last.next_time
      wl = w
    end
  end
  wl
end

def shortest_interpolated_worldline
  t = VERY_LARGE_NUMBER
  wl = nil
  @worldline.each do |w|
    if t > w.worldpoint.last.time
      t = w.worldpoint.last.time
      wl = w
    end
  end
  wl
end

def evolve(dt_era, dt_max, shared_flag)
  nsteps = 0
  while shortest_interpolated_worldline.worldpoint.last.time < @end_time
    unless shared_flag
      shortest_extrapolated_worldline.grow(self, dt_max)
      nsteps += 1
    else
      t = shortest_extrapolated_worldline.worldpoint.last.next_time
      @worldline.each do |w|
        w.worldpoint.last.next_time = t
        w.grow(self, dt_era)
        nsteps += 1
      end
    end
  end
  [next_era(dt_era), nsteps]
```

```
  end

  def next_era(dt_era)
    e = WorldEra.new
    e.start_time = @end_time
    e.end_time = @end_time + dt_era
    @worldline.each do |wl|
      e.worldline.push(wl.next_worldline(e.start_time))
    end
    e
  end

  def take_snapshot(time)
    take_snapshot_except(nil, time)
  end

  def take_snapshot_except(wl, time)
    ws = WorldSnapshot.new
    ws.time = time
    @worldline.each do |w|
      s = w.take_snapshot_of_worldline(time)
      ws.body.push(s) unless w == wl
    end
    ws
  end

  def write_diagnostics(t, nsteps, initial_energy, init_flag = false)
    STDERR.print "at time t = #{sprintf("%g", t)} "
    STDERR.print "(from interpolation after #{nsteps} steps "
    if init_flag
      STDERR.print "to time #{sprintf("%g", @start_time)}):\n"
    else
      STDERR.print "to time #{sprintf("%g", @end_time)}):\n"
    end
    take_snapshot(t).write_diagnostics(initial_energy)
  end

  def setup_from_snapshot(ss, dt_param, dt_era)
    @start_time = ss.time
    @end_time = @start_time + dt_era
    ss.body.each do |b|
      wl = WorldLine.new
      wl.setup_from_single_worldpoint(b, dt_param, ss.time)
      @worldline.push(wl)
    end
  end
```

```
end
```

7.2 Choosing a Particle

Alice: Let's start with `evolve`. As we've done before, it's often easier to start in the middle than at the beginning.

Bob: In the generic case, `shared_flag` will be false. We have added the option `-a` to ask the code to run with shared time steps, to make it easier to compare the results with our previous code `nbody_sh1.rb`. But the default value of this flag is `false`, in which case the `unless` statement evaluates to being `true`. The next line then tells the shortest extrapolated world line to extend itself. After that, we do bookkeeping, by adding one to the variable that counts the time steps.

Alice: That line indeed reads like English. Let's reconstruct the meaning. We need to find the first particle that needs to be pushed along its orbit, from its current world point to the next world point. This means that we needt to find the particle with the earliest sales date, which is the smallest time `@next_time`, an instance variable of class `WorldPoint`. However, at this level we are still two levels removed from that class, and we certainly don't want to rely on the particular way that this latest sales date is implemented.

Bob: At this point, we'd better remind our students what this latest sales date means.

Alice: It is the latest time for which our extrapolation from the last world point is still valid. What can be extrapolated till this time are the positions and velocities, using an extrapolation method which is again hidden on this level, and implemented on the next level down, in class `WorldLine`. All we can do on this higher level is to ask a particle . . .

Bob: . . . I thought particles were two levels down!

Alice: You're right! Thanks. I have to learn to be more precise. All we can do on this higher level is to ask a *worldline* to extend itself. The way it extends itself is by asking the particle as defined at the last world point to take a step to the next world point. So indirectly we will give a command to the world point.

Now to be precise, yes, you are right, the *world point* is two levels down. But you can also say that a particle, with a particular identity, is represented in our code by a world line. A single world line corresponds to the *history* of a single particle. So in that sense, a *world line* in 4-D stands for a particles, just like a *body* stood for a particle in our 3-D way of coding.

Bob: I'm still wondering whether we've not introduced too many legalities and abstractions here, but I'll reserve judgment. So to sum up, we look for the

particle with the earliest latest sales date. Hmm, listening to myself, I have to admit that that does sound a bit contradictory.

Alice: But it is correct. Each particle has a latest sales date. And from among the ensemble of all particles, we choose the particle that has the earliest such date. But I agree, 'earliest latest' is confusing. So we could say instead that we are looking for a particle that has the earliest cutoff time for its extrapolation.

Bob: And if you look toward the future, the world line that has the earliest cutoff is shortest. Hence we ask for the shortest extrapolatable world line. I see. But why did we call it extrapolated?

Alice: I guess extrapolatable just sounds a bit funny, not altogether palatable. We can later change the wording, if we want. Anyway, *that* particle, once we've found it, is asked to extend itself, and we'll see on the `WorldLine` level just how it wants to do that.

Bob: Quite a long discussion, to explain a single line of code! And especially a line of code that already looks like plain English. It goes to show that, once you understand how a code works, it is easier to read the source itself than the explanation.

Alice: Isn't that true for anything?

7.3 The Case of Shared Time Steps

Bob: I guess. Let's move on to the `else` clause. We will enter the next part when the `shared_flag` is true, that is, when we want to use shared time steps. In that case, let's see, aha, we again find our 'earliest latest salesdate' particle, but this time we ask it a more complicated question. We ask it for the `next_time` of the last world point on its world line. Funny how you have to read Ruby backwards in order to translate it into an English sentence.

Alice: Looking at it now, I must say, I don't like the fact that we are specifically addressing the array `worlpoint[]` in the class `WorldLine`. While this gives us access to the last point in the array, we wind up with a object of class `WorldPoint` and then we ask this object for its `next_time`. This is precisely the sort of level crossing that we wanted to avoid.

Bob: Well, the most important thing is: the code works, as we have already tested. Let's not touch it for now. I'm already happy if I can get an overview of all that we've done so far.

Alice: Fair enough, but I do want to get back to this blemish, at some point. This is not modular programming!

Bob: For now, all I want to do is to understand what we wrote here. In the case of shared time steps, shouldn't we ask all particles for their prefered time step, and then take the shortest time step, so as to make nobody unhappy?

Alice: Well, *if* all particles would be synchronized, what you just said would be correct, but in general, they will not be synced. Each particle will have been updated to their own time `@time`, and each particle is thinking about taking a next step, in the future, to a time `@time_next`.

So in order to make nobody unhappy, we should avoid making a shared system step that would go beyond anybody's `@time_next`. This implies that our shared step should exactly land on the earliest value `@time_next` within the ensemble of all particles. And this is precisely what we are doing here: the particle with the shortest extrapolated world line has by definition the smallest `@time_next` value.

Bob: I see, yes, that must be right. So the only difference with the statement three lines higher is that now we obtain the actual time `@time_next`, and then we ask all particles to make a step toward this time.

Alice: What we really *should* do is to ask all world lines to extend themselves to this time. What we *actually* are doing here is to cross two levels and use the `next_time` method to set the variable `@time_next` within each particle by hand. To be cleaned up!

Bob: Okay, okay, later, as we already said. Here at the end of the shared time step case, we update the step counter by one. Shouldn't that be updating by N, for N particles? Ah, no, the update happens within the loop, and we pass through the loop N times. Fine.

7.4 Finishing the Loop

Alice: let's get back to the beginning of our `while` loop. We have not yet looked at the conditional statement there. The code tells us that we keep looping only while the last world point for the shortest interpolated world line has a time that is earlier than the time `@end_time` at which the era ends.

Bob: And now you're going to tell us that we're crossing levels again.

Alice: Don't worry, I'll shut up.

Bob: For now.

Alice: For now, yes. Shortest interpolated, what again did that mean? Ah, it must mean the time up till the last computed world point. Up to that point, positions and velocities can be interpolated between two world points; beyond that point, we have to use extrapolation.

Bob: So shortest interpolated means earliest interpolatable, yes?

Alice: I guess we decided to avoid that word, but if you want to use interpolatable, then we are asking for the earliest time at which a particle runs out of being interpolatable. Now *that* sounds like a truly awful sentence. What we are really doing is asking for the earliest time at which a particle runs out of

interpolation. Or simply, we ask for the earliest actually computed world point among those points that are at the end of a world line.

Bob: Aha, the earliest latest computed point!

Alice: Yes, I you like. Now when the time corresponding to this world point is at least as large as the end time `@end_time` of our era, we can stop. In this case we know for sure that all particles have passed the finish line. What we really need is a guarantee that every worldine is fully interpolatable from start to finish of an era, during the whole time span from `@begin_time` till `@end_time`.

Bob: I remember that we had a discussion about that. We first thought that it might be enough to let every particle make just one step to or beyond the finish. But then we realized that some particles would have to step quite a bit *beyond* the finish.

Alice: Yes. As long as there is even one particle left that has not yet stepped out of the duration of our era, we cannot stop. But it is possible that this last particle has a rather long time step, and hence a very late latest sales date. Given our algorithm, this particle will only step forward when the system time reached its latest sales date. If this date is way beyond the end of our era, all other particles may be forced to take many steps each, before this straggler particle is finally asked to step out of the era.

Bob: In itself, this is not such a bad situation, since all the work we do now, in the current era, is something we don't have to do in the next era. It is only at the end of a calculation, that we do a bit too much work. And in addition, each era will need more storage than strictly necessary. So for both reasons we decided to put an upper limit to the length of a time step for any particle. Ah yes, this is the parameter `dt_max` that is passed as a parameter to `evolve`.

Alice: In methods `World#setup_from_world` and `World#setup_from_snapshot`, the value of `dt_max` is set as being equal to the duration of the era, multiplied by a factor `dt_max_param`. By default, this last number is equal to one, but we can set it with the command line argument `-m` or `--max_timestep_param`.

Bob: So we can limit the growth of an era in two ways, by setting the length `dt_era` itself, which is the duration during which all world lines are interpolatable. In other words, this is a slice of spacetime between two fixed-time hypersurfaces. Each world line will stick out at both sides, back into the past and forward into the future. And we are guaranteed that each world line has at least one world point on its world line sticking out into the past, before the slice, and one world point sticking out into the future, beyond the slice.

In general, many world lines will have several points sticking out at one or two sides of the slice. All we can say is that the longest line sticking out at either side of the slice has a length that is at most a factor `dt_max_param` longer than the thickness of the slice.

Alice: Yes, that's the picture. Individual time step algorithms sure complicate matters!

Bob: And so does spacetime visualization. But I must admit, all this has given me a fresh way of looking at what I have been doing in the past, when I coded up individual time step methods. And I do think I have a more complete overview of the whole picture now. Whether or not it will turn out to be useful, has to be seen, but it is fun to see an old idea in a new light.

7.5 Some Concern

Alice: For the time being, I think the use of `dt_max` is okay, but I must say, I'm a bit concerned about the way we let the user specify it. It may seem natural to couple the value of `dt_max` directly to the value of the era length `dt_era`, making them equal by default, but there is a danger.

Bob: What type of danger?

Alice: Ideally, when you change the length of an era, by using the command line option `-e` or `--era_length`, you don't expect the orbits of the stars to change. But in fact, they will change, initially by a small amount, but because of the exponential instability of stellar dynamics, after a while the orbits will be completely different. That is a bad thing.

Bob: I see. Doubling the length of an era, without changing the option `-m` or `--max_timestep_param` would lead to an effective doubling of `dt_max`, which could lead to larger timesteps for some particles, and hence different orbits. The way to counter this would be to make that last option smaller, also by the same factor of two. Let me try it:

```
|gravity> kali mkplummer.rb -n 4 -s 1 | kali nbody_set_id.rb > tmp.in
==> Takes an N-body system, and gives each body a unique ID <==
value of @body_id for 1st body: n = 1
Floating point precision: precision = 16
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
==> Plummer's Model Builder <==
Number of particles: N = 4
pseudorandom number seed given: 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
             actual seed used: 1
|gravity> kali world1.rb -t 1 < tmp.in > tmp1.out
```

```
==> Individual Time Step, Individual Integration Scheme Code <==
Determines the time step size: dt_param = 0.01
Duration of an era: dt_era = 0.01
Maximum time step (units dt_era): dt_max_param = 1.0
Diagnostics output interval: dt_dia = 1.0
Snapshot output interval: dt_out = 1.0
Duration of the integration: t = 1.0
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
at time t = 0 (from interpolation after 0 steps to time 0):
    E_kin = 0.25 ,     E_pot =  -0.5 ,      E_tot = -0.25
       E_tot - E_init = 0
        (E_tot - E_init) / E_init = -0
at time t = 1 (from interpolation after 4291 steps to time 1):
    E_kin = 0.313 ,     E_pot =  -0.563 ,      E_tot = -0.25
       E_tot - E_init = -1.98e-10
        (E_tot - E_init) / E_init = 7.92e-10
|gravity> kali world1.rb -t 1 -e 0.02 < tmp.in > tmp2.out
==> Individual Time Step, Individual Integration Scheme Code <==
Determines the time step size: dt_param = 0.01
Duration of an era: dt_era = 0.02
Maximum time step (units dt_era): dt_max_param = 1.0
Diagnostics output interval: dt_dia = 1.0
Snapshot output interval: dt_out = 1.0
Duration of the integration: t = 1.0
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
at time t = 0 (from interpolation after 0 steps to time 0):
    E_kin = 0.25 ,     E_pot =  -0.5 ,      E_tot = -0.25
       E_tot - E_init = 0
        (E_tot - E_init) / E_init = -0
at time t = 1 (from interpolation after 4268 steps to time 1):
    E_kin = 0.313 ,     E_pot =  -0.563 ,      E_tot = -0.25
       E_tot - E_init = 1.6e-10
        (E_tot - E_init) / E_init = -6.4e-10
|gravity> kali world1.rb -t 1 -e 0.02 -m 0.5 < tmp.in > tmp3.out
==> Individual Time Step, Individual Integration Scheme Code <==
Determines the time step size: dt_param = 0.01
Duration of an era: dt_era = 0.02
Maximum time step (units dt_era): dt_max_param = 0.5
Diagnostics output interval: dt_dia = 1.0
Snapshot output interval: dt_out = 1.0
```

```
Duration of the integration: t = 1.0
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
at time t = 0 (from interpolation after 0 steps to time 0):
    E_kin = 0.25 ,     E_pot =  -0.5 ,      E_tot = -0.25
       E_tot - E_init = 0
        (E_tot - E_init) / E_init = -0
at time t = 1 (from interpolation after 4291 steps to time 1):
    E_kin = 0.313 ,     E_pot =  -0.563 ,      E_tot = -0.25
       E_tot - E_init = -1.98e-10
        (E_tot - E_init) / E_init = 7.92e-10
|gravity> cat tmp1.out tmp2.out | kali nbody_diff.rb
==> 6N-dimensional phase space distance between two N-body systems <==
Floating point precision: precision = 2
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
6N-dim. phase space dist. for two 4-body systems:   6.9922800657753722e-09
|gravity> cat tmp1.out tmp3.out | kali nbody_diff.rb
==> 6N-dimensional phase space distance between two N-body systems <==
Floating point precision: precision = 2
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
6N-dim. phase space dist. for two 4-body systems:   0.0000000000000000e+00
```

Alice: Yes, that works, but a typical user is not likely to think about this, and anyway, modularity requires us to decouple the notions of era length and maximum step size length, if the first one is supposed not to change the physics, while the second one does.

Bob: Perhaps. But it does work!

Alice: It works for a factor of two alright, but if you introduce a factor of three increase for the era length, and you want to compensate by shrinking the maximum timestep parameter by a factor three, you may not be able to cancel the two factors precisely without round-off error.

Bob: I agree that we could come up with a better version. However, as you said, for now the current choice of command line arguments will do.

7.6 Back to the Beginning

Alice: Let's move to the beginning of the `WorldEra` class, to see what other methods we have here. The method `initialize`, which is called when we order a `WorldEra.new` sets up an empty array `@worldline`, ready to accept a bunch of world lines.

The methods `acc_and_jerk` and `timescale` appear here only as a way to pass them on to a snapshot. When the request comes, from a lower level, to provide acceleration and jerk, or a time scale value, then first a snapshot is taken, and then the snapshot is asked to care care of the requests.

The method `startup_and_report_energy` is almost a tautology: it hands on the request to start things up down to the `WorldLine` level, and it takes a snapshot in order to ask that snapshot to provide the total energy value. This value is then returned to the calling function in `World`, one level higher. I must say, I still have to remind myself that in Ruby you don't have to write something like `return energy`: you can leave out the `return` part, and the value that is automatically returned is the value that the last method evaluated provides, in this case the method `total_energy`.

Bob: But once you get used to it, Ruby does provide a nicely compact notation. It allows you to pack a lot of information on one page, and still in such a way that it is pretty clear what is being done.

We now come to the next two methods, `shortest_extrapolated_worldline` and `shortest_interpolated_worldline`. They speak for themselves: they do just what you think they should do.

Alice: At least when you understand the concepts they are pointing too; but we've just gone over that. `evolve` we've analyzed already. The method `next_era` creates a new instance of the class `WorldEra`. This method is invoked, whenever we have evolved a complete era, and we are ready for another one. Now where does this method gets called again?

Bob: At the end of `evolve`. It is part of what is handed back to the `World` level, together with the value of the counter of the number of steps that have been taken.

Alice: But `next_era` seems to contruct, as the name implies, a next era, the next slice in time, after the current era. And it populates that era with those parts of the world lines that fit into the next era, using `WorldLine#next_worldline`. Now I'm puzzled. If `evolve` returns an almost empty new time slice, with only the beginning parts of each world line populated, haven't we lost most of what we just computed?

Bob: We have to go back to the `World` level, to decide that. Let me print out `World#evolve`:

```
def evolve(c)
```

```
    while @era.start_time < @t_end
      @new_era, dn = @era.evolve(c.dt_era, @dt_max, c.shared_flag)
      @nsteps += dn
      @time = @era.end_time
      while @t_dia <= @era.end_time and @t_dia <= @t_end
        @era.write_diagnostics(@t_dia, @nsteps, @initial_energy)
        @t_dia += c.dt_dia
      end
      while @t_out <= @era.end_time and @t_out <= @t_end
        output(c)
        @t_out += c.dt_out
      end
      @old_era = @era
      @era = @new_era
    end
  end
```

The object that is returned by `WorldEra#evolve` is assigned to `@new_era`, as it should be. But the era that has just been completely evolved is not lost: it is still referenced by the variable `@era`. You see, `WorldEra#evolve` is an instance method, and we invoke it within `World#evolve` as the method that belongs to `@era`, in the first line of the `while` loop. So `@era` is still available to us.

Alice: Ah, yes. I guess I'm thinking too much in a data flow style of programming. Ruby is object oriented, and even though it supports a data flow style too, here we are definitely dealing with objects, where we ask the objects to evolve themselves, but without changing their identity.

Bob: Or, in Ruby terms, without changing their `object_id`; their name can change. At the end of `World#evolve` above, you can see how we change the name of the object, pointed at by `@era`, to `@old_era`, while we call `@new_era` now `@era`, as it should be, before entering the next round.

Alice: So not only is the fully history of the time slice that we just computed preserved, also the history of the previous time slice remains available, through `@old_era`. Why did we do that?

Bob: I seem to remember that we thought it might come in handy to have some extra information at hand. It probably not essential, and if we ever run into problems with not having enough memory, this will be the first thing to drop.

Alice: And then, during the round after that, our object loses its last reference, when the pointer `@old_era` gets reassigned to the era after the one we have been following. As a result, soon after that, the Ruby garbage collector will do its job, and the memory places are recycled. Okay, I now see the whole picture.

7.7 Snapshots

Bob: The last few methods in `WorldEra` are all related to IO through the use of snapshots. At the bottom we see how we do input, starting from a snapshot, in `setup_from_snapshot`. The real work is done on the `WorldLine` level, in `setup_from_single_worldpoint`, where we add each next body to a brand new world line, created in the line above. The line below shows how this new world line is than added at the end of our array of world lines, `@worldline`.

Alice: I keep wondering whether we shouldn't have choosen the plural version of the name, to make it `@worldlines` instead. After all the array `@worldline[]` contains many world lines.

Bob: But when we talk about the 3rd world line, it is nice to be able to write `@worldline[2]`. It would make less sense to write `@worldlines[2]`. This is the same logic we used way back when when we gave the `NBody` class an array `body` for its bodies.

Alice: Yeah, I can see the logic, either way. Oh well, we started off this way, so we may as well continue.

Bob: Let us now turn to output, and the way it is mediated by snapshots. We have the function `take_snapshot(time)` that constructs a full snapshot at a given time. It translates into using the generic method `take_snapshot_except`, except that nothing is excepted, because we give the `nil` argument for the exception, if you see what I mean.

Alice: Hard to hear what you mean, exceptionally so, but yes, I see what you may have meant. And the reason for this overly complex redirection is that the function `take_snapshot_except` is the real workhorse that is used in preparation for a force calculation: it presents a single particle with the positions of all other particles, after which that single particle can compute the distances to all other particles, and based on that, accelerations and jerks.

Bob: And this ends our tour through `WorldEra`. Time to descend one more level, to `WorldLine`.

Chapter 8

The `WorldLine` Class

8.1 A Matter of Identity

Alice: We started on the top level, with the `World` class, which you could view as the generalization of our old `NBody` class. Now, in the `WorldLine` class, we finally encounter the generalization of our old `Body` class.

Bob: I thought that `WorldPoint` generalized, or better took over, our `Body` notion.

Alice: Yes and no. It is all a question of identity. What is important about a particular body, say a particular star, perhaps with particular planets revolving around it . . .

Bob: . . . and programmers living on those planets . . .

Alice: . . . which would make it a very particular planet indeed. Anyway, such a star is instantiated at different times through different world points, but it is the particular world line that belongs to that particular star. So in terms of identity, it is really the `WorldLine` class that takes over here from the `Body` class.

Bob: While the positions and velocities and all that important type of detail can only be found by digging one level deeper, to `WorldPoint`. And if you project everything down the time direction, on a space like hypersurface, both a given world line, as well as the world points on it, all project onto a single particle.

Alice: Yes, that's it! Well, here is the listing for the `WorldLine` class:

```
class WorldLine

  attr_accessor  :worldpoint
```

```
def initialize
  @worldpoint = []
end

def grow(era, dt_max)
  new_point = take_step(era, dt_max)
  @worldpoint.push(new_point)
end

def setup(time)
  @worldpoint[0].setup(time)
end

def startup(era, dt_max)
  wp = @worldpoint[0]
  acc, jerk = era.acc_and_jerk(self, wp)
  timescale = era.timescale(self, wp)
  wp.startup(acc, jerk, timescale, @dt_param, dt_max)
end

def take_step(era, dt_max)
  new_point = predict
  acc, jerk = era.acc_and_jerk(self, new_point)
  timescale = era.timescale(self, new_point)
  new_point.correct(@worldpoint.last, acc, jerk,
                            timescale, @dt_param, dt_max)
end

def predict
  wp = @worldpoint.last
  wp.extrapolate(wp.next_time)
end

def valid_extrapolation?(time)
  unless @worldpoint.last.time <= time and time <= @worldpoint.last.next_time
    raise "#{time} not in [#{@worldpoint.last.time}, #{@worldpoint.last.next_tim
  end
end

def valid_interpolation?(time)
  unless @worldpoint[0].time <= time and time <= @worldpoint.last.time
    raise "#{time} not in [#{@worldpoint[0].time}, #{@worldpoint.last.time}]"
  end
end
```

```
  def take_snapshot_of_worldline(time)
    if time >= @worldpoint.last.time
      valid_extrapolation?(time)
      wp = @worldpoint.last.extrapolate(time)
      wp.body_id = @body_id if defined? @body_id
      wp
    else
      valid_interpolation?(time)
      @worldpoint.each_index do |i|
        if @worldpoint[i].time > time
          wp = @worldpoint[i-1].interpolate(@worldpoint[i], time)
          wp.body_id = @body_id if defined? @body_id
          return wp
        end
      end
    end
  end

  def next_worldline(time)
    valid_interpolation?(time)
    i = @worldpoint.size
    loop do
      i -= 1
      if @worldpoint[i].time <= time
        wl = self.clone
        wl.worldpoint = @worldpoint[i...@worldpoint.size]
        return wl
      end
    end
  end

  def setup_from_single_worldpoint(b, dt_param, time)
    @worldpoint[0] = b.to_worldpoint
    @body_id = @worldpoint[0].body_id if @worldpoint[0].body_id
    @dt_param = dt_param
    setup(time)
  end

end
```

8.2 Evolving on a Third Level

Bob: I see no particular prefered entry point here. Why don't we just go down the list of methods.

Alice: Well, `extend` plays the role that `evolve` has played on the two higher levels; we could have called this method `evolve` as well: a world line evolves by extending itself. But it doesn't make any difference: `extend` happens to be the first method, following the initializer that assigns an empty array of world points.

Bob: All `extend` does is to take one step, and to add the resulting new world point to the growing array of world points. We should have written this as a oneliner method:

```
def extend(era, dt_max)
  @worldpoint.push(take_step(era, dt_max))
end
```

Alice: That would probably have been just a bit too terse for me; on the face of it, it's not clear that `take_step` returns a world point. But perhaps, when we get more familiar with this notation, and if we really stick to this notation, such a way of writing may well begin to look natural. For now, let's stick to two lines.

Bob: The next method, `startup`, does the first force calculation, and the determination of the time scale needed for finding the next time step size. All of this is done here for one particular particle, the one associated with our world line.

Then, in the method following that one, `take_step` does the real thing: like `startup`, it does a force calculation, but it does that calculation at a predicted position and velocity, and after that it uses the information given by the forces to do a corrector step, finishing off all the work. Presumably this means that everything is done as before, in `nbody_ind1.rb`, but now hidden in the internal workings of the `WorldPoint` class.

Alice: It must be, otherwise our code wouldn't have given the same results. Now `predict` does an extrapolation. But wait, we have to be careful. We have been talking about two different types of prediction.

8.3 Two Types of Prediction

Bob: Two types?

Alice: In order for one particle to step forwards, it has to predict its own position and velocity for its own desired time `@next_time`. Then, in order to calculate the forces that it will receive at this time from all other particles, it has to asked all other particles to predict their positions and velocities too. So there is the active prediction of one particle, for its own purposes, and the passive prediction of all other particles, obeying the wishes of the one particle that is, temporarily, in charge here.

Bob: Clearly, `predict` here is the active prediction, since it asks our particle to step forwards to its own `@next_time`.

Alice: But where does the passive prediction take place? Ah, that must be what is done in `take_snapshot_of_worldline`. That is the next method, after the two trivial, but important, checking methods `valid_extrapolation?` and `valid_interpolation?`. At least that seems like the most logical place. But I'd like to trace the actual flow of the logic. Let's take a step back, to `take_step`, so that we can follow exactly where the active and passive prediction takes place.

The active prediction happens directly in the first line. Now in the second line, we are already doing a force calculation, which implies that both and active and passive prediction has taken place already. So the passive prediction mechanism must be invoked as a side effect of the call `era.acc_and_jerk(self, new_point)`. Let's go back to `WorldEra`, to inspect that that call is doing:

```
def acc_and_jerk(wl, wp)
  take_snapshot_except(wl, wp.time).get_acc_and_jerk(wp.pos, wp.vel)
end
```

Aha: first a snapshot is taken of all particles, except of the particle associated with the worldline that calls this function.

Bob: And a snapshot is constructed by asking all particles to predict their positions and velocities at the time of that snapshot. That must be the passive prediction part: it happens for all particles, except the calling particle, that has already predicted its position and velocity in the active prediction call.

Alice: That must be right, but still, I'd like to see specifically how that is done. Let's go once more to `take_snapshot_except`:

```
def take_snapshot_except(wl, time)
  ws = WorldSnapshot.new
  ws.time = time
  @worldline.each do |w|
    s = w.take_snapshot_of_worldline(time)
    ws.body.push(s) unless w == wl
  end
  ws
end
```

Aha, each world line is asked to execute `take_snapshot_of_worldline`, which we already suspected to be the executioner of the passive predict step. And here we have the proof! Okay, I'm happy, we now have all the pieces on the table, and we can see the flow of the logic.

8.4 Extrapolation and Interpolation

Bob: While we have identified the role that `take_snapshot_of_worldline` plays, we haven't yet looked at how it does what it does. The first part, after the `if` statement, makes sense to me. If we ask a particle to predict itself, passively as you said, to a time that is past the time at which it computed its last world point, we have to do an extrapolation. If not, we can interpolate between two completed worldpoints, one before and one after the time at which we order our particle to be predicted.

The extrapolation part is clear: first find the lst world point, and then extrapolate beyond. As an English sentence: extrapolate beyond the last world point, which means in Rubyese: `worldpoint.last.extrapolate`. And the particle that is handed back has to be branded with the right `body_id` number.

Alice: I don't like that image.

Bob: What image?

Alice: Branded. I can see a poor calf in front of me, being branded with a hot iron.

Bob: Aren't you a bit too sensitive?

Alice: It may be the `body` part of the terminology that triggered my additional imagination. It's hard to brand a world point.

Bob: Well, let's say that the `body_id` is a *sticker* that is stuck on the particle. In any case, thus equipped the world point is returned. Now what I am puzzled about is the question of what happens in the `else` part of the method. There is a `each` loop that is traversed. Why?

Alice: If the time at which we want to take a snapshot is not in the extrapolated part of the world line, it must be in the interpolated part, and to be more precise, it must be in between two world points, as you just said. The question is: in between which two. This loop loops over all points, to find the right two.

Bob: Ah, of course. And that is why we use `each_index` and not just `each`. The point is to traverse the world line in an ordered way. So we start at the beginning of the array, at the oldest point, `@worldpoint[0]`, and then move forward to `@worldpoint[1]`, `@worldpoint[2]`, and so on, until we find a point with a time that is larger than the desired time. At that point, no pun intended, our previous point is still before the desired time. So the last two points in hand must straddle the desired time.

Alice: I think that pun was intended. But you've made your point, or points really: that's how it works. The point just before the desired time is `@worldpoint[i-1]`, and it is given the point after the desired time, `@worldpoint[i]` as the argument for its interpolation method.

Bob: And the rest happens as before, in the `if` part: the body receives a sticker with its name on it.

Alice: Thank you!

8.5 Wrapping Up

Bob: The method `next_worldline` is called when we want to create a new era, using the method `WorldEra#next_era`, as we've seen before. Here we see how it is done, by cloning the existing world line, and returning a new one, with only a subset of the previous string of world points.

Alice: `Array` of world points.

Bob: Strung together, yes. And this `clone` method produces only a shallow clone.

Alice: I remember that this took me a while to get used to. `clone` only copies the instance variables of an object, but it does not copy the objects that the instance variables may be pointing to. So already in the case of an array, `clone` provides a new reference to the same old array.

Bob: Indeed: the world points themselves are unaffected. The variable `@worldpoint` that belongs to `self`, the objects that is calling this method, and the variable `wl.worldpoint` are different. However, immediately after the cloning, `@worldpoint[0]` of `self` and `@worldpoint[0]` of `wl`, that can be accessed through a call to `wl.worldpoint`, are identical.

Then, after the cloning operation, pruning takes place in the next line. That much I remember, but how exactly did we do that?

Ah, we should go back to the loop at the top.

Alice: Let's go to the top of `next_worldline` altogether. We first insist that the time at which we want to create a next worldline is a time at which interpolation can take place. Aha, yes, this method gets invoked exactly at the end of an era.

Bob: Ah, the end of an era . . .

Alice: Sounds nostalgic, doesn't it? So at that time we are guaranteed that all worldline have at least one worldpoint, and often more, sticking out in time beyond this end of our era. Hence we should be able to interpolate at that time. If not, something is seriously wrong, and it is good to check this.

Next we determine what seems like the size of a world point . . .

Bob: . . . which should be zero if it is a point.

Alice: You see, there really is something to say for giving this array the plural name `worldpoints` instead. So `i` starts of being the length of the whole array of worldpoints, that is, the number of world points in our world line. And in the loop, `i` is used to reference the array, while decreasing one unit at the time. In other words, we are stepping through the array, starting at the end, and walking toward the beginning of the array.

Bob: Which means that the time associated with each world point decreases. And we keep going back in time, until we find a world point that has a time that is smaller or equal to the time `time` at which we call our method.

Alice: The end of an era.

Bob: At it is at *that* point that we clone our world line, and give it a new array of world points, with the array starting at index `i`.

Alice: You mean, the new array is populated with the objects that were at locations `i` and higher in the old array. Of course, the new array starts at `0`, and ends at an index that is `i` smaller in value that the previous ending index.

Bob: Yes, that is what I meant. And now I see the meaning: this new array is guaranteed to start with a world point that is at or before the time `time`, which is the end of the previous era . . .

Alice: . . . but at the same time the beginning of the new era. So the new era is guaranteed to have wordlines sticking out into the past, with at least one worldpoint either sticking out or sitting right on the boundary.

Bob: Yes. Amazing, isn't it, how complex something becomes when you have to put it into words. But yes, this is how it works.

Alice: And the final method `setup_from_single_worldpoint` does what it is supposed to do: it imports what might be a particle of class `Body` and converts it into a real `Worldpoint`. This is what we added at the end of the code, in an extension of the `Body` class, as follows:

```
class Body

  def to_worldpoint
    wp = WorldPoint.new
    wp.restore_contents(self)
  end

end
```

Bob: Ah yes, the method `restore_contents` is something we constructed within the file `acsio.rb`. It does a virtual output and input, to make it looks as if we were actually reading in a `Worldpoint` object, instead of a `Body`.

Alice: And after this class conversion, the world line assumes the same identity as the body we read in, by taking over the same value of its `@body_id`. And at the end, we call `setup`, which as we have seen passed the work on to the world point method with the same name. Done!

Chapter 9

The `WorldPoint` Class

9.1 Down at the Bottom

Bob: Now we come to the bottom of our hierarchy, where we find the `WorldPoint` class, which is a subclass of the `Body` class:

```
class WorldPoint < Body

  attr_accessor :acc, :jerk,
                :time, :next_time, :nsteps,
                :minstep, :maxstep,
                :body_id

  def setup(time)
    @time = @next_time = time
    @acc = @pos*0
    @jerk = @pos*0
    @nsteps = 0
    @minstep = VERY_LARGE_NUMBER
    @maxstep = 0
  end

  def startup(acc, jerk, timescale, dt_param, dt_max)
    @acc = acc
    @jerk = jerk
    dt = timescale * dt_param
    dt = dt_max if dt > dt_max
    @next_time = @time + dt
  end
```

```
def correct(old_point, acc, jerk, timescale, dt_param, dt_max)
  @acc = acc
  @jerk = jerk
  dt = timescale * dt_param
  dt = dt_max if dt > dt_max
  @next_time = @time + dt
  dt = @time - old_point.time
  @vel = old_point.vel + (1/2.0)*(old_point.acc + @acc)*dt +
                         (1/12.0)*(old_point.jerk - @jerk)*dt**2
  @pos = old_point.pos + (1/2.0)*(old_point.vel + @vel)*dt +
                         (1/12.0)*(old_point.acc - @acc)*dt**2
  admin(old_point.time)
  self
end

def admin(old_time)
  dt = @time - old_time
  @maxstep = dt if @maxstep < dt
  @minstep = dt if @minstep > dt
  @nsteps = @nsteps + 1
end

def extrapolate(t)
  if t > @next_time
    raise "t = " + t.to_s + " > @next_time = " + @next_time.to_s + "\n"
  end
  wp = WorldPoint.new
  wp.minstep = @minstep
  wp.maxstep = @maxstep
  wp.nsteps = @nsteps
  wp.mass = @mass
  wp.time = t
  dt = t - @time
  wp.pos = @pos + @vel*dt + (1/2.0)*@acc*dt**2 + (1/6.0)*@jerk*dt**3
  wp.vel = @vel + @acc*dt + (1/2.0)*@jerk*dt**2
  wp
end

def interpolate(other, t)
  wp = WorldPoint.new
  wp.minstep = @minstep
  wp.maxstep = @maxstep
  wp.nsteps = @nsteps
  wp.mass = @mass
  wp.time = t
  dt = other.time - @time
```

```
    snap = (-6*(@acc - other.acc) - 2*(2*@jerk + other.jerk)*dt)/dt**2
    crackle = (12*(@acc - other.acc) + 6*(@jerk + other.jerk)*dt)/dt**3
    dt = t - @time
    wp.pos = @pos + @vel*dt + (1/2.0)*@acc*dt**2 + (1/6.0)*@jerk*dt**3 +
             (1/24.0)*snap*dt**4 + (1/144.0)*crackle*dt**5
    wp.vel = @vel + @acc*dt + (1/2.0)*@jerk*dt**2 + (1/6.0)*snap*dt**3 +
             (1/24.0)*crackle*dt**4
    wp
  end

  def kinetic_energy
    0.5*@mass*@vel*@vel
  end

  def potential_energy(body_array)
    p = 0
    body_array.each do |b|
      unless b == self
        r = b.pos - @pos
        p += -@mass*b.mass/sqrt(r*r)
      end
    end
    p
  end

end
```

9.2 Setting Up and Starting Up

Alice: We have defined three phases of operation, a setup phase, a startup phase, and a normal phase in which we push particles forward. The difference between there three is finally becoming clear now.

Bob: At startup, we are revving the engines, so to speak: we are computing the initial force calculations, but we are not moving any particle yet. The engines are running in place.

Alice: In a way, we are simulating a previous step, by leaving the system in exactly the type of state it would have been in, had we arrived at the initial time from a previous integration. To the extent that a previous integration would have provided us with force calculations at the end of a previous step, we are doing those force calculations now.

Bob: Now why did we not just combine the `setup` and `startup` routines?

Alice: Good question. Hmm. On this level there certainly does not seem to be any clear reason not to combine these two. But we must have had *some* reason. Let's go back to the next level up, to `WordLine`, where these functions must be called from.

Ah, yes, there too we have two functions with the same name, `setup` and `startup`. Hmm again. At this point, too, it seems clear that we could have combined these functions. Well, let's keep moving up, to the `WorldEra` level.

Bob: Ho, before we do that, notice that there is another setup related method on the `WorldLine` level. At the very end, we have `setup_from_single_worldpoint`. In that method, the worldpoint itself is set up. Not *that* type of setting up surely has to be done for each particle, before we can even think of starting up the force calculation for each particle.

Alice: Perhaps we should have combined those two methods, `WorldLine#setup` and `WorldLine#setup_from_single_worldpoint`. Something to keep in mind for a future version. For now, let's leave this version alone as it is, since it works.

Still, I'd like to see what happens on higher levels. I'll just do a search for `setup`.

On the `WorldEra` level, the only thing that turns up is `setup_from_snapshot`. Ah, this is in contrast with what we just saw on the `WorldLine` level, where we set up a single word line from a single world point, as the starting point. Here we setup the whole system, starting from a whole snapshot.

Continuing the search for `setup`, here is the next function, now on the top level, `World#setup_from_world` and `World#setup_from_snapshot`.

Bob: The first one is almost trivial, since everything has already been set up in the previous run; all we have to do is to set a few appropriate variables and off we go. The second one, `setup_from_snapshot`, is the interesting one. And hey, look, on the fifth line, we are doing a form of startup, in `startup_and_report_energy`.

Alice: Now *that* is confusing. Definitely, in the next rewrite, I will insist on separating setting up and starting up on all levels!

Bob: But as you already said, not today. We still have to go through the `WorldPoint` class definition, remember! We've only looked at the first two methods.

9.3 Predicting and Correcting.

Alice: The rest should not contain too many surprises. The `correct` step looks just like what we already had in `nbody_ind1.rb`: all the coefficients are the same, as they should be, just the notation is somewhat different.

But wait a minute, if we have a corrector here, what happened with our predic-

tor?

Bob: You have forgotten already? On the `WorldLine` level, we took a single step as follows:

```
def take_step(era, dt_max)
  new_point = predict
  acc, jerk = era.acc_and_jerk(self, new_point)
  timescale = era.timescale(self, new_point)
  new_point.correct(@worldpoint.last, acc, jerk,
                            timescale, @dt_param, dt_max)
end
```

which indeed contains a predict step and a correct step. The correct step is a direct invocation of `WorldPoint#correct` which we are now looking at. And the predict step is given on the `WorldLine` level as:

```
def take_step(era, dt_max)
  new_point = predict
  acc, jerk = era.acc_and_jerk(self, new_point)
  timescale = era.timescale(self, new_point)
  new_point.correct(@worldpoint.last, acc, jerk,
                            timescale, @dt_param, dt_max)
end
```

This means that we should not look for a method `WorldPoint#predict` but a method `WorldPoint#extrapolate`.

Alice: Hmm, not exactly what I would have expected. And I don't want to rely on memory. The logic of the code should speak for itself!

Perhaps we should call `WorldPoint#extrapolaten` `WorldPoint#predict` instead. But no, that won't do either, since there are other places where we only want to exrapolate. There we specifically want to call `WorldPoint#extrapolate` and it would be confusing to have to call `WorldPoint#predict`. Perhaps the best thing to do is to introduce the name `WorldPoint#predict` as just a one line method calling `WorldPoint#extrapolate`, as a kind of alias.

Bob: Not today! I'm sure that once you get started, your taste for abstraction and modularity and what not will get you carried away. We'll leave that for the next pass.

Where were we? The method `admin` is trivial, just does some bookkeeping. And then we have `extrapolate`, which we just discussed. It does indeed exactly what `Body#predict_step` did in `nbody_ind1.rb`.

The next method is `interpolate`. Ah, this is new.

Alice: Yes. We decided that we'd better be as accurate as we could reasonably be with our interpolation. After all, the main reason to interpolate is to construct snapshots that we then hand to a diagnostics routine.

Bob: I thought the *main* reason to construct snapshots was to be able to do force calculations.

Alice: True, but for *those* snapshots, we have to extrapolate particles; excuse me, *predict* particle positions. This double meaning of predicting and extrapolating is making things sound complicated.

Bob: I see. So while you're growing world lines dynamically, you're always taking snapshots that are a bit ahead of every known world point. That makes sense. But by the time you look back to do all the diagnostics for a completed era, you use interpolation.

Alice: Exactly. And in order to get the most accurate energy estimate, we may as well use all the information we have, from both world points, before and after the time at which we want to interpolated.

Bob: And if we count parameters, we have computed acceleration and jerk at each of the two world point. These four pieces of information can be transformed into four quantities at one the points, for convenience, to allow us to construct a Taylor series there, without loss of information.

This means that at that point we have the acceleration and jerk that were already available, and we compute the next two derivatives, the snap and crackle, $d^4\mathbf{r}/dt^4$ and $d^5\mathbf{r}/dt^5$, from the two accelarations and jerks we have at hand. Got it!

9.4 Interpolation

Alice: I don't remember exactly how we derived the equations here. I'd feel more comfortable just to check them again.

Bob: I do remember that it was quite a bit of pen and paper work. I'm not eager to do this again, given that we know it works.

Alice: I agree, everything does seem to work, in the sense that we get good fourth-order scaling of errors, and that for small enough time steps we get an accuracy that is close what you can expect, using double precision. That is a pragmatic check, but I would like to check also on a more theoretical level.

This does not mean that we have to rederive everything here. All we have to do is to check that the equations given here for the position and velocity in between the two worldpoints actually reduce to the exact position and velocity of each worldpoint, in the limit of minimal and maximal interpolation during this interval.

Bob: Is that really enough? A straight line between the positions at the two

worldpoints would not be a good interpolation, yet it would agree with the positions at either end.

Alice: You are right, I was a little to glib. I should have added that we need to check whether the appropriate derivates at either end have the correct values as well.

Bob: How many derivatives does it take to be 'appropriate'?

Alice: Good question. Hmm. Let us derive the answer, from scratch. To start with, we are dealing with a fourth-order integration scheme, so we would like to see a fourth-order polynomial, an expression in powers of `dt` with `dt**4` as the highest power. Now what are the precise conditions for the coefficients . . .

Ah, there is a simpler way. You see, the expressions for the corrected position and velocities, given by the methods `correct`, three methods above `interpolate`, provide fourth-order expressions for the position and velocities, in terms of powers of `dt`. We may as well reinterpret those equations as providing us the intermediate values for position and velocity as well!

Bob: They look mightily second-order in `dt`. But yes, that's because they have been cleverly written that way. If you express everything in terms of a Taylor series at the beginning point, the true fourth-order nature becomes clear. This is a trick similar to the one that is employed for the leapfrog: that scheme looks first-order, but is actually second-order. This one looks second-order, but is actually fourth-order.

That much is fine. But how do you want to recycle these equations to use them for interpolation?

Alice: The corrector makes one big step from one world point to the next. What we need is a method to make a smaller step, starting at the same world point, but moving a smaller distance in time. We can use these equations, just by shrinking the time `dt`.

Bob: But how? These expressions rely on the fact that you know the values of the acceleration and jerk both at the beginning point and end point. If you know keep the beginning point the same, but you place the end point somewhere in the middle of the interval, you have no information any more about the acceleration and jerk at that point. These two quantities had been computed, directly from Newton's equations of motion and its derivative, only at the two given world points.

Alice: True, we have to reconstruct the values for acceleration and jerk that we would have measured there, had we decided to do so. And we only have to do that to fourth order in accuracy in `dt`. What I did when I wrote the interpolator, was to evaluate the snap and crackle at the beginning of the interval, from the measured acceleration and jerk values . . .

Bob: . . . and then to use that snap and crackle in turn in a Taylor series to reconstruct the needed acceleration and jerk at the end of the interval.

Alice: I could have done that, but that would not be necessary: once you have the Taylor series at the beginning of the interval, to sufficiently high order, you already have the interpolation formula you were looking for!

Bob: So you've put those expressions in there, when we were creating this code. I wonder why I couldn't remember them at all!

Alice: You were rebooting the server, and I had a copy on my laptop, so I thought I might as well make myself useful too.

Bob: Useful indeed. But wait a minute, I see a fifth order term at the end of the right hand side expression for the interpolated position! It is `(1/144.0)*crackle*dt**5`. What is that doing there?

Alice: I remember we had a good reason to add that one, but I agree, strictly speaking, it is not needed. Well, let's just accept the fact that we put it there, and see what happens when we check the equations.

Bob: But not only it is not needed, it is inconsistent! If you differentiate the expression for `wp.pos`, you get the one for `wp.vel`, term by term, except for the last term. The coefficient for the fifth-order term in `wp.pos` should have been `(1/120.0)`, shouldn't it, in order to lead to the proper last term in the velocity.

Alice: Yes, it should have been, if we had insisted to make the velocity to be the exact derivative of the position. But given that we only need the position to fourth order, this discrepancy does not necessarily spell trouble. But your objection is well taken; let's put in a note to remind ourselves to get back to it.

Bob: Fair enough.

9.5 Derivation

Alice: Let us start now by checking whether I did my home work correctly. If so, the method `interpolate` should reproduce the correct position and velocity values for the world point `other`, if we take `dt` to be exactly the difference in time `dt = other.time - @time` between the two world points.

Bob: But that is already the value we have chosen, in line 7! Where do we use the value `t`? Ah, I see, further down, in line 10, we change the value of `dt` to be `t - time`.

Alice: That's quite confusing. We'd better introduce a different notation for the first `dt`, perhaps call it τ, and use `dt` only for the last two lines.

Bob: That might be better, but as we said before, we're now analyzing the code, and let us postpone changes for later.

Alice: Anyway, to check whether the interpolator lands correctly on the end point, we actually equate these two forms of using `dt`. So let's get started. The values associated with the world point `self`, and given by instance variable names starting with a `@`, are the values at the beginning of the step. Let us

indicate those with a subscript 0. The values at the end of the step, associated with the world point `other`, I will give a subscript 1. We then get the following notation, if we use τ for the interval `dt`, and translate the four lines for `snap`, `crackle`, `wp.pos`, and `wp.vel`:

```
snap = (-6*(@acc - other.acc) - 2*(2*@jerk + other.jerk)*dt)/dt**2
crackle = (12*(@acc - other.acc) + 6*(@jerk + other.jerk)*dt)/dt**3
wp.pos = @pos + @vel*dt + (1/2.0)*@acc*dt**2 + (1/6.0)*@jerk*dt**3 +
         (1/24.0)*snap*dt**4 + (1/144.0)*crackle*dt**5
wp.vel = @vel + @acc*dt + (1/2.0)*@jerk*dt**2 + (1/6.0)*snap*dt**3 +
         (1/24.0)*crackle*dt**4
```

We then get:

$$\begin{aligned} \mathbf{s}_0 &= (-6\mathbf{a}_0 + 6\mathbf{a}_1)/\tau^2 \ + \ (-4\mathbf{j}_0 - 2\mathbf{j}_1)/\tau \\ \mathbf{c}_0 &= (12\mathbf{a}_0 - 12\mathbf{a}_1)/\tau^3 \ + \ (6\mathbf{j}_0 + 6\mathbf{j}_1)/\tau^2 \end{aligned} \tag{9.1}$$

and

$$\begin{aligned} \mathbf{r}_1 &= \mathbf{r}_0 + \mathbf{v}_0\tau + \tfrac{1}{2}\mathbf{a}_0\tau^2 + \tfrac{1}{6}\mathbf{j}_0\tau^3 + \tfrac{1}{24}\mathbf{s}_0\tau^4 + \tfrac{1}{144}\mathbf{c}_0\tau^5 \\ \mathbf{v}_1 &= \mathbf{v}_0 + \mathbf{a}_0\tau + \tfrac{1}{2}\mathbf{j}_0\tau^2 + \tfrac{1}{6}\mathbf{s}_0\tau^3 + \tfrac{1}{24}\mathbf{c}_0\tau^4 \end{aligned} \tag{9.2}$$

I will start by substituting the expressions in eq. (9.1) into the second line of eq. (9.2). This gives:

$$\begin{aligned} \mathbf{v}_1 &= \mathbf{v}_0 + \mathbf{a}_0\tau + \tfrac{1}{2}\mathbf{j}_0\tau^2 + \tfrac{1}{6}\left\{(-6\mathbf{a}_0 + 6\mathbf{a}_1)\tau \ + \ (-4\mathbf{j}_0 - 2\mathbf{j}_1)\tau^2\right\} \\ &\qquad + \tfrac{1}{24}\left\{(12\mathbf{a}_0 - 12\mathbf{a}_1)\tau \ + \ (6\mathbf{j}_0 + 6\mathbf{j}_1)\tau^2\right\} \\ &= \mathbf{v}_0 + \mathbf{a}_0\tau + \tfrac{1}{2}\mathbf{j}_0\tau^2 + \left\{(-\mathbf{a}_0 + \mathbf{a}_1)\tau \ + \ (-\tfrac{2}{3}\mathbf{j}_0 - \tfrac{1}{3}\mathbf{j}_1)\tau^2\right\} \\ &\qquad + \left\{(\tfrac{1}{2}\mathbf{a}_0 - \tfrac{1}{2}\mathbf{a}_1)\tau \ + \ (\tfrac{1}{4}\mathbf{j}_0 + \tfrac{1}{4}\mathbf{j}_1)\tau^2\right\} \\ &= \mathbf{v}_0 + \tfrac{1}{2}(\mathbf{a}_0 + \mathbf{a}_1)\tau \ + \ \tfrac{1}{12}(\mathbf{j}_0 - \mathbf{j}_1)\tau^2 \end{aligned} \tag{9.3}$$

And this is exactly what our corrector `correct` tells us that the new velocity should be:

```
@vel = old_point.vel + (1/2.0)*(old_point.acc + @acc)*dt +
                       (1/12.0)*(old_point.jerk - @jerk)*dt**2
```

So far so good! Time to look at the first line of eq. (9.2), again substituting eq. (9.1):

$$\begin{aligned}
\mathbf{r}_1 &= \mathbf{r}_0 + \mathbf{v}_0\tau + \tfrac{1}{2}\mathbf{a}_0\tau^2 + \tfrac{1}{6}\mathbf{j}_0\tau^3 + \tfrac{1}{24}\left\{(-6\mathbf{a}_0 + 6\mathbf{a}_1)\tau^2 \;+\; (-4\mathbf{j}_0 - 2\mathbf{j}_1)\tau^3\right\} \\
&\qquad + \tfrac{1}{144}\left\{(12\mathbf{a}_0 - 12\mathbf{a}_1)\tau^2 \;+\; (6\mathbf{j}_0 + 6\mathbf{j}_1)\tau^3\right\} \\
&= \mathbf{r}_0 + \mathbf{v}_0\tau + \tfrac{1}{2}\mathbf{a}_0\tau^2 + \tfrac{1}{6}\mathbf{j}_0\tau^3 + \left\{(-\tfrac{1}{4}\mathbf{a}_0 + \tfrac{1}{4}\mathbf{a}_1)\tau^2 \;+\; (-\tfrac{1}{6}\mathbf{j}_0 - \tfrac{1}{12}\mathbf{j}_1)\tau^3\right\} \\
&\qquad + \left\{(\tfrac{1}{12}\mathbf{a}_0 - \tfrac{1}{12}\mathbf{a}_1)\tau^2 \;+\; (\tfrac{1}{24}\mathbf{j}_0 + \tfrac{1}{24}\mathbf{j}_1)\tau^3\right\} \\
&= \mathbf{r}_0 + \mathbf{v}_0\tau + \tfrac{1}{6}(2\mathbf{a}_0 + \mathbf{a}_1)\tau^2 \;+\; \tfrac{1}{24}(\mathbf{j}_0 - \mathbf{j}_1)\tau^3 \qquad (9.4)
\end{aligned}$$

This expression cannot be directly compared yet with the equivalent expression in our corrector, since there we used the difference between the old and the new velocity.

```
@pos = old_point.pos + (1/2.0)*(old_point.vel + @vel)*dt +
                       (1/12.0)*(old_point.acc - @acc)*dt**2
```

It is probably easiest to translate that expression directly into our notation, after which we can substitute eq. (9.2):

$$\begin{aligned}
\mathbf{r}_1 &= \mathbf{r}_0 + \tfrac{1}{2}(\mathbf{v}_0 + \mathbf{v}_1)\tau + \tfrac{1}{12}(\mathbf{a}_0 - \mathbf{a}_1)\tau^2 \\
&= \mathbf{r}_0 + \tfrac{1}{2}\left(2\mathbf{v}_0 + \tfrac{1}{2}(\mathbf{a}_0 + \mathbf{a}_1)\tau \;+\; \tfrac{1}{12}(\mathbf{j}_0 - \mathbf{j}_1)\tau^2\right)\tau + \tfrac{1}{12}(\mathbf{a}_0 - \mathbf{a}_1)\tau^2 \\
&= \mathbf{r}_0 + \mathbf{v}_0\tau + \tfrac{1}{6}(2\mathbf{a}_0 + \mathbf{a}_1)\tau^2 \;+\; \tfrac{1}{24}(\mathbf{j}_0 - \mathbf{j}_1)\tau^3 \qquad (9.5)
\end{aligned}$$

This is indeed exactly the same result as we obtained in eq. (9.4), and this concludes the proof that we have done exactly the right thing, in postulating eqs. (9.1) and (9.2). These equations were in turn what we had put in our method `interpolate`, so this proofs that we wrote that one correctly.

9.6 Glitches

Bob: Apart from the inconsistency in the fact that the derivative of the position is no longer exactly what it should be.

Alice: You mean the fact that the last term in the first line of eq. (9.2) has a coefficient of 1/444, rather than 1/120?

Bob: Yes. You told me you would get back to that.

Alice: Well, we now can find the reason. It would be nice to keep the expressions for position and velocity consistent, in that you retrieve the velocity by differentiating the position with respect to time. At the same time, it would be nice to guarantee that the interpolation formula gives you the correct values for both position and velocity, at the beginning and at the end of the interpolated time interval.

Bob: And we now see that we cannot do both.

Alice: Exactly. Note, however, that we *can* do both on the level of a fourth-order approximation. This inconsistency shows up only at fifth order in τ.

To be specific, in the first line of eq. (9.2), we could have left out the last term altogether. In that case, we would be fourth-order consistent, and you would be granted your wish: to this order of accuracy, indeed the velocity would be the time derivative of the position. But we our interpolation formula would be off by exactly that last term, when we would ask for the position of the interpolation at the end of the interpolated interval.

Bob: So in the end, it is mostly a matter of taste, whether we mind to have fifth-order discontinuities in interpolated position, as a function of time, in a fourth-order scheme, or whether we mind having the velocity and the time derivative of the position being off by a fifth-order term.

Alice: Well summarized. And I guess that is the penalty of having any finite precision: if you are accurate to k^{th} order, you will make errors on the $(k+1)^{\text{th}}$ order, in all kinds of ways.

Bob: I'm happy with the current choice, to keep the interpolator fifth-order accurate in position. But it was a good exercise to check exactly what we did and why.

Chapter 10

`World` Input and Output

10.1 A Bad Thing

Alice: It's great to have such a well structured code now, and I'm especially happy that output issues are now much better decoupled from running the orbit integrator.

Bob: I thought you'd like that kind of modularity!

Alice: Indeed. And not only that, we are now guaranteed that orbit integration will not be affected at all by any type of diagnostics output. In contrast, in the earlier code `nbody_ind1.rb`, we had to synchronize all particles in order to do any type of output. That meant that a change of diagnostics affected the actual outcome of the code. Here is an example, starting again from our standard input choice:

```
|gravity> kali mkplummer.rb -n 4 -s 1 | kali nbody_set_id.rb > test.in
==> Plummer's Model Builder <==
Number of particles: N = 4
pseudorandom number seed given: 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
             actual seed used: 1
==> Takes an N-body system, and gives each body a unique ID <==
value of @body_id for 1st body: n = 1
Floating point precision: precision = 16
Screen Output Verbosity Level: verbosity = 1
```

```
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
```

We will run the code first with two energy reports, only at the beginning and the end:

```
|gravity> kali nbody_ind1.rb -t 1 -d 1 < test.in > test1.out
==> Individual Time Step Hermite Code <==
Parameter to determine time step size: dt_param = 0.01
Interval between diagnostics output: dt_dia = 1.0
Time interval between snapshot output: dt_out = 1.0
Duration of the integration: t = 1.0
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
at time t = 0, after 0 steps :
  E_kin = 0.25 , E_pot =  -0.5 , E_tot = -0.25
             E_tot - E_init = 0
  (E_tot - E_init) / E_init = -0
at time t = 1, after 4257 steps :
  E_kin = 0.313 , E_pot =  -0.563 , E_tot = -0.25
             E_tot - E_init = 7.24e-10
  (E_tot - E_init) / E_init = -2.9e-09
```

and then with three reports:

```
|gravity> kali nbody_ind1.rb -t 1 -d 0.5 < test.in > test2.out
==> Individual Time Step Hermite Code <==
Parameter to determine time step size: dt_param = 0.01
Interval between diagnostics output: dt_dia = 0.5
Time interval between snapshot output: dt_out = 1.0
Duration of the integration: t = 1.0
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
at time t = 0, after 0 steps :
  E_kin = 0.25 , E_pot =  -0.5 , E_tot = -0.25
             E_tot - E_init = 0
  (E_tot - E_init) / E_init = -0
at time t = 0.5, after 2021 steps :
```

```
  E_kin = 0.241 , E_pot =  -0.491 , E_tot = -0.25
            E_tot - E_init = 3.54e-10
  (E_tot - E_init) / E_init = -1.42e-09
at time t = 1, after 4261 steps :
  E_kin = 0.313 , E_pot =  -0.563 , E_tot = -0.25
            E_tot - E_init = -5.01e-10
  (E_tot - E_init) / E_init = 2e-09
```

Note, first of all, that the total number of steps is different, which is a dead giveaway that something has changed. And indeed, we can check directly that the particles have not landed at exactly the same place.

```
|gravity> cat test[12].out | kali nbody_diff.rb
==> 6N-dimensional phase space distance between two N-body systems <==
Floating point precision: precision = 2
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
6N-dim. phase space dist. for two 4-body systems:   1.1238501517441471e-08
```

This is definitely a bad thing. Unlike in quantum mechanics, a classical mechanics system should not be influenced by a measurement!

10.2 Modularity in Practice

Bob: And the claim is that this problem does not occur for our new code, since we evolve all particles one era at a time, and then do the diagnostics off-line, so to speak. Well, we'd better check:

```
|gravity> kali world1.rb -t 1 -d 1 < test.in > test1.out
==> Individual Time Step, Individual Integration Scheme Code <==
Determines the time step size: dt_param = 0.01
Duration of an era: dt_era = 0.01
Maximum time step (units dt_era): dt_max_param = 1.0
Diagnostics output interval: dt_dia = 1.0
Snapshot output interval: dt_out = 1.0
Duration of the integration: t = 1.0
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
```

```
Incremental indentation: add_indent = 2
at time t = 0 (from interpolation after 0 steps to time 0):
    E_kin = 0.25 ,     E_pot =  -0.5 ,      E_tot = -0.25
       E_tot - E_init = 0
        (E_tot - E_init) / E_init = -0
at time t = 1 (from interpolation after 4291 steps to time 1):
    E_kin = 0.313 ,     E_pot =  -0.563 ,      E_tot = -0.25
       E_tot - E_init = -1.98e-10
        (E_tot - E_init) / E_init = 7.92e-10
|gravity> kali world1.rb -t 1 -d 0.5 < test.in > test2.out
==> Individual Time Step, Individual Integration Scheme Code <==
Determines the time step size: dt_param = 0.01
Duration of an era: dt_era = 0.01
Maximum time step (units dt_era): dt_max_param = 1.0
Diagnostics output interval: dt_dia = 0.5
Snapshot output interval: dt_out = 1.0
Duration of the integration: t = 1.0
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
at time t = 0 (from interpolation after 0 steps to time 0):
    E_kin = 0.25 ,     E_pot =  -0.5 ,      E_tot = -0.25
       E_tot - E_init = 0
        (E_tot - E_init) / E_init = -0
at time t = 0.5 (from interpolation after 2033 steps to time 0.5):
    E_kin = 0.241 ,     E_pot =  -0.491 ,      E_tot = -0.25
       E_tot - E_init = -2.44e-10
        (E_tot - E_init) / E_init = 9.74e-10
at time t = 1 (from interpolation after 4291 steps to time 1):
    E_kin = 0.313 ,     E_pot =  -0.563 ,      E_tot = -0.25
       E_tot - E_init = -1.98e-10
        (E_tot - E_init) / E_init = 7.92e-10
```

Alice: At least the number of steps has not been changed.

Bob: A good sign already! Let's inspect the phase space distance:

```
|gravity> cat test[12].out | kali nbody_diff.rb
==> 6N-dimensional phase space distance between two N-body systems <==
Floating point precision: precision = 2
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
6N-dim. phase space dist. for two 4-body systems:   0.0000000000000000e+00
```

Alice: Perfect. Modularity in practice, not only in theory!

10.3 A Test Suite

Bob: Ah, but there is more! Not only can we safely put our a report about energy conservation, any time we want, but we can even do a full output, and then restart from that output, as often as we want, without affecting the trajectories of any of the particles in the slightest.

Alice: Again, that is the claim. We'd better test that, too! Of course, this idea only works for a full output, one that dumps the whole internal data structure, for which we had the command line option `-r`. Without that option, you get a synchronized snapshot, and changing the frequency of synchronization will lead to the same problem we saw above with the previous code.

Bob: Let's build a little test suite. We'll create an input file, and then ask for output in different ways.

To start with, let's compare a run that directly integrates for two time units with a run that restarts from a dump halfway. We can do that comparison in two ways: compare a dump for each type of run, direcly and in stages, and compare a snapshot output for each type of run, where in the latter case of course the intermediate output still has to be a dump.

Then, for good measure, let's integrate till three time units. We can then compare a run that goes there non-stop with a run that makes one stop, and with a run that makes two stops along the way. We'll make those two comparisons on the dump level.

And since we require that *everything* should be unchanged, I want to do more than measuring phase space distances: I'll do a `diff` on the whole output file, to check whether *any* variable, important or not, has *any* change whatsoever. Here is the test suite:

```
|gravity> kali mkplummer.rb -n 4 -s 1 | kali nbody_set_id.rb > tmp0
==> Takes an N-body system, and gives each body a unique ID <==
value of @body_id for 1st body: n = 1
Floating point precision: precision = 16
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
==> Plummer's Model Builder <==
Number of particles: N = 4
pseudorandom number seed given: 1
```

```
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
            actual seed used: 1
|gravity> kali world1.rb -t 1 -r < tmp0 | kali acstail.rb > tmp1w
==> Returns the last n chunks of ACS data <==
Number of last ACS data chunks returned: n = 1
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
==> Individual Time Step, Individual Integration Scheme Code <==
Determines the time step size: dt_param = 0.01
Duration of an era: dt_era = 0.01
Maximum time step (units dt_era): dt_max_param = 1.0
Diagnostics output interval: dt_dia = 1.0
Snapshot output interval: dt_out = 1.0
Duration of the integration: t = 1.0
World output format, instead of snapshot
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
at time t = 0 (from interpolation after 0 steps to time 0):
    E_kin = 0.25 ,     E_pot =  -0.5 ,      E_tot = -0.25
       E_tot - E_init = 0
        (E_tot - E_init) / E_init = -0
at time t = 1 (from interpolation after 4291 steps to time 1):
    E_kin = 0.313 ,     E_pot =  -0.563 ,      E_tot = -0.25
       E_tot - E_init = -1.98e-10
        (E_tot - E_init) / E_init = 7.92e-10
|gravity> kali world1.rb -t 1 -r < tmp1w | kali acstail.rb > tmp2ww
==> Returns the last n chunks of ACS data <==
Number of last ACS data chunks returned: n = 1
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
==> Individual Time Step, Individual Integration Scheme Code <==
Determines the time step size: dt_param = 0.01
Duration of an era: dt_era = 0.01
Maximum time step (units dt_era): dt_max_param = 1.0
Diagnostics output interval: dt_dia = 1.0
```

```
Snapshot output interval: dt_out = 1.0
Duration of the integration: t = 1.0
World output format, instead of snapshot
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
at time t = 1 (from interpolation after 4291 steps to time 0.99):
    E_kin = 0.313 ,     E_pot =  -0.563 ,      E_tot = -0.25
       E_tot - E_init = -1.98e-10
        (E_tot - E_init) / E_init = 7.92e-10
at time t = 2 (from interpolation after 6995 steps to time 2):
    E_kin = 0.171 ,     E_pot =  -0.421 ,      E_tot = -0.25
       E_tot - E_init = -6.5e-11
        (E_tot - E_init) / E_init = 2.6e-10
|gravity> kali world1.rb -t 2 -o 2 -r < tmp0 | kali acstail.rb > tmp2w
==> Returns the last n chunks of ACS data <==
Number of last ACS data chunks returned: n = 1
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
==> Individual Time Step, Individual Integration Scheme Code <==
Determines the time step size: dt_param = 0.01
Duration of an era: dt_era = 0.01
Maximum time step (units dt_era): dt_max_param = 1.0
Diagnostics output interval: dt_dia = 1.0
Snapshot output interval: dt_out = 2.0
Duration of the integration: t = 2.0
World output format, instead of snapshot
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
at time t = 0 (from interpolation after 0 steps to time 0):
    E_kin = 0.25 ,     E_pot =  -0.5 ,      E_tot = -0.25
       E_tot - E_init = 0
        (E_tot - E_init) / E_init = -0
at time t = 1 (from interpolation after 4291 steps to time 1):
    E_kin = 0.313 ,     E_pot =  -0.563 ,      E_tot = -0.25
       E_tot - E_init = -1.98e-10
        (E_tot - E_init) / E_init = 7.92e-10
at time t = 2 (from interpolation after 6995 steps to time 2):
    E_kin = 0.171 ,     E_pot =  -0.421 ,      E_tot = -0.25
       E_tot - E_init = -6.5e-11
        (E_tot - E_init) / E_init = 2.6e-10
```

```
|gravity> kali world1.rb -t 2 -o 2 < tmp0 | kali acstail.rb > tmp2s
==> Returns the last n chunks of ACS data <==
Number of last ACS data chunks returned: n = 1
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
==> Individual Time Step, Individual Integration Scheme Code <==
Determines the time step size: dt_param = 0.01
Duration of an era: dt_era = 0.01
Maximum time step (units dt_era): dt_max_param = 1.0
Diagnostics output interval: dt_dia = 1.0
Snapshot output interval: dt_out = 2.0
Duration of the integration: t = 2.0
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
at time t = 0 (from interpolation after 0 steps to time 0):
    E_kin = 0.25 ,     E_pot =  -0.5 ,      E_tot = -0.25
       E_tot - E_init = 0
        (E_tot - E_init) / E_init = -0
at time t = 1 (from interpolation after 4291 steps to time 1):
    E_kin = 0.313 ,     E_pot =  -0.563 ,      E_tot = -0.25
       E_tot - E_init = -1.98e-10
        (E_tot - E_init) / E_init = 7.92e-10
at time t = 2 (from interpolation after 6995 steps to time 2):
    E_kin = 0.171 ,     E_pot =  -0.421 ,      E_tot = -0.25
       E_tot - E_init = -6.5e-11
        (E_tot - E_init) / E_init = 2.6e-10
|gravity> kali world1.rb -t 1 < tmp1w | kali acstail.rb > tmp2ws
==> Returns the last n chunks of ACS data <==
Number of last ACS data chunks returned: n = 1
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
==> Individual Time Step, Individual Integration Scheme Code <==
Determines the time step size: dt_param = 0.01
Duration of an era: dt_era = 0.01
Maximum time step (units dt_era): dt_max_param = 1.0
Diagnostics output interval: dt_dia = 1.0
Snapshot output interval: dt_out = 1.0
Duration of the integration: t = 1.0
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
```

```
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
at time t = 1 (from interpolation after 4291 steps to time 0.99):
    E_kin = 0.313 ,     E_pot =  -0.563 ,      E_tot = -0.25
       E_tot - E_init = -1.98e-10
        (E_tot - E_init) / E_init = 7.92e-10
at time t = 2 (from interpolation after 6995 steps to time 2):
    E_kin = 0.171 ,     E_pot =  -0.421 ,      E_tot = -0.25
       E_tot - E_init = -6.5e-11
        (E_tot - E_init) / E_init = 2.6e-10
|gravity> kali world1.rb -t 1 -r < tmp2w | kali acstail.rb > tmp3ww
==> Returns the last n chunks of ACS data <==
Number of last ACS data chunks returned: n = 1
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
==> Individual Time Step, Individual Integration Scheme Code <==
Determines the time step size: dt_param = 0.01
Duration of an era: dt_era = 0.01
Maximum time step (units dt_era): dt_max_param = 1.0
Diagnostics output interval: dt_dia = 1.0
Snapshot output interval: dt_out = 1.0
Duration of the integration: t = 1.0
World output format, instead of snapshot
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
at time t = 2 (from interpolation after 6995 steps to time 1.99):
    E_kin = 0.171 ,     E_pot =  -0.421 ,      E_tot = -0.25
       E_tot - E_init = -6.5e-11
        (E_tot - E_init) / E_init = 2.6e-10
at time t = 3 (from interpolation after 11014 steps to time 3.01):
    E_kin = 0.506 ,     E_pot =  -0.756 ,      E_tot = -0.25
       E_tot - E_init = -2.81e-09
        (E_tot - E_init) / E_init = 1.13e-08
|gravity> kali world1.rb -t 3 -o 3 -r < tmp0 | kali acstail.rb > tmp3w
==> Returns the last n chunks of ACS data <==
Number of last ACS data chunks returned: n = 1
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
==> Individual Time Step, Individual Integration Scheme Code <==
Determines the time step size: dt_param = 0.01
```

```
Duration of an era: dt_era = 0.01
Maximum time step (units dt_era): dt_max_param = 1.0
Diagnostics output interval: dt_dia = 1.0
Snapshot output interval: dt_out = 3.0
Duration of the integration: t = 3.0
World output format, instead of snapshot
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
at time t = 0 (from interpolation after 0 steps to time 0):
    E_kin = 0.25 ,     E_pot =  -0.5 ,      E_tot = -0.25
       E_tot - E_init = 0
        (E_tot - E_init) / E_init = -0
at time t = 1 (from interpolation after 4291 steps to time 1):
    E_kin = 0.313 ,     E_pot =  -0.563 ,      E_tot = -0.25
       E_tot - E_init = -1.98e-10
        (E_tot - E_init) / E_init = 7.92e-10
at time t = 2 (from interpolation after 6995 steps to time 2):
    E_kin = 0.171 ,     E_pot =  -0.421 ,      E_tot = -0.25
       E_tot - E_init = -6.5e-11
        (E_tot - E_init) / E_init = 2.6e-10
at time t = 3 (from interpolation after 11014 steps to time 3.01):
    E_kin = 0.506 ,     E_pot =  -0.756 ,      E_tot = -0.25
       E_tot - E_init = -2.81e-09
        (E_tot - E_init) / E_init = 1.13e-08
|gravity> kali world1.rb -t 1 -r < tmp2ww | kali acstail.rb > tmp3www
==> Returns the last n chunks of ACS data <==
Number of last ACS data chunks returned: n = 1
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
==> Individual Time Step, Individual Integration Scheme Code <==
Determines the time step size: dt_param = 0.01
Duration of an era: dt_era = 0.01
Maximum time step (units dt_era): dt_max_param = 1.0
Diagnostics output interval: dt_dia = 1.0
Snapshot output interval: dt_out = 1.0
Duration of the integration: t = 1.0
World output format, instead of snapshot
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
at time t = 2 (from interpolation after 6995 steps to time 1.99):
```

```
    E_kin = 0.171 ,     E_pot =  -0.421 ,      E_tot = -0.25
       E_tot - E_init = -6.5e-11
        (E_tot - E_init) / E_init = 2.6e-10
at time t = 3 (from interpolation after 11014 steps to time 3.01):
    E_kin = 0.506 ,     E_pot =  -0.756 ,      E_tot = -0.25
       E_tot - E_init = -2.81e-09
        (E_tot - E_init) / E_init = 1.13e-08
|gravity> diff tmp2w tmp2ww | wc
      0       0       0
|gravity> diff tmp2s tmp2ws | wc
      0       0       0
|gravity> diff tmp3w tmp3ww | wc
      0       0       0
|gravity> diff tmp3w tmp3www | wc
      0       0       0
```

10.4 Using Snapshots Instead

Alice: Wonderful. That gives you confidence! We really seem to have things under control now. But wouldn't it be nice to check that there are still differences when you do *not* use a full dump at intermediate times, but use snapshots instead?

Bob: Sure. Easy to add. And since we now expect differences, let me do a phase space distance measurement at the end as well. Here you are:

```
|gravity> kali world1.rb -t 1 < tmp0 | kali acstail.rb > tmp1s
==> Returns the last n chunks of ACS data <==
Number of last ACS data chunks returned: n = 1
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
==> Individual Time Step, Individual Integration Scheme Code <==
Determines the time step size: dt_param = 0.01
Duration of an era: dt_era = 0.01
Maximum time step (units dt_era): dt_max_param = 1.0
Diagnostics output interval: dt_dia = 1.0
Snapshot output interval: dt_out = 1.0
Duration of the integration: t = 1.0
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
```

```
Incremental indentation: add_indent = 2
at time t = 0 (from interpolation after 0 steps to time 0):
    E_kin = 0.25 ,     E_pot =  -0.5 ,      E_tot = -0.25
       E_tot - E_init = 0
        (E_tot - E_init) / E_init = -0
at time t = 1 (from interpolation after 4291 steps to time 1):
    E_kin = 0.313 ,     E_pot =  -0.563 ,      E_tot = -0.25
       E_tot - E_init = -1.98e-10
        (E_tot - E_init) / E_init = 7.92e-10
|gravity> kali world1.rb -t 1 < tmp1s | kali acstail.rb > tmp2ss
==> Returns the last n chunks of ACS data <==
Number of last ACS data chunks returned: n = 1
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
==> Individual Time Step, Individual Integration Scheme Code <==
Determines the time step size: dt_param = 0.01
Duration of an era: dt_era = 0.01
Maximum time step (units dt_era): dt_max_param = 1.0
Diagnostics output interval: dt_dia = 1.0
Snapshot output interval: dt_out = 1.0
Duration of the integration: t = 1.0
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
at time t = 1 (from interpolation after 0 steps to time 1):
    E_kin = 0.313 ,     E_pot =  -0.563 ,      E_tot = -0.25
       E_tot - E_init = 0
        (E_tot - E_init) / E_init = -0
at time t = 2 (from interpolation after 2713 steps to time 2):
    E_kin = 0.171 ,     E_pot =  -0.421 ,      E_tot = -0.25
       E_tot - E_init = -2.8e-10
        (E_tot - E_init) / E_init = 1.12e-09
|gravity> diff tmp2ss tmp2s | wc
     76     146    2262
|gravity> cat tmp2s | wc
     74     128    2185
|gravity> cat tmp2ss tmp2s | kali nbody_diff.rb
==> 6N-dimensional phase space distance between two N-body systems <==
Floating point precision: precision = 2
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
```

```
6N-dim. phase space dist. for two 4-body systems:   2.2390038382354583e-08
```

Alice: As we expected, many lines are different, as you can see from the word-count `wc` after the `diff`, which is comparable in size to the file itself. And in phase space there is a definite distance between the two types of runs as well.

Chapter 11

Literature References

[to be provided]