The Art of Computational Science

The Kali Code

vol. 11

Shared Time Steps: Phase Space Diagnostics

Piet Hut and Jun Makino

July 11, 2005

Contents

Preface **5**

0.1 xxx 5

1 Shared Time Steps **7**

1.1 Estimating the Time Step Size 7

1.2 Implementation on the `Body` Level 8

1.3 Implementation on the `NBody` Level 9

1.4 Multiple Time Step Schemes 11

2 Distance Measures **15**

2.1 Phase Space 15

2.2 Numbering Bodies 16

2.3 A Test Count 17

2.4 Defensive Programming 19

2.5 Putting it together 21

2.6 Checking it out 22

3 A Puzzle **27**

3.1 Starting with Runge-Kutta 27

3.2 Energy Scaling 28

3.3 Phase Space Distance Scaling 29

3.4 The Shared Time Step Code 33

3.5 Back to Basics 34

3.6 Even more Basic 36

4 The Answer **39**
4.1 Inspection 39
4.2 Of Course! 41
4.3 The Right Thing 43
4.4 Setting up the Leapfrog 46
4.5 Testing the Leapfrog 48

5 Reproducibility Lost **49**
5.1 Making Sure 49
5.2 The Danger of Short Options 51
5.3 Know Your Output File 52
5.4 Synchronization 54
5.5 The Toll it Takes 56

6 Reproducibility Gained **61**
6.1 Asynchronicity 61
6.2 Overshooting 63
6.3 Living in a Logical World 65
6.4 An ACS `tail` Version 66
6.5 Testing `tail` 67

7 Efficiency **71**
7.1 Pure Thought 71
7.2 Concrete Results 72
7.3 More is Different 74

8 Code Listing **77**
8.1 The Whole Code 77

9 Literature References **87**

Preface

0.1 xxx

We thank xxx, xxx, and xxx for their comments on the manuscript.

Piet Hut and Jun Makino

Chapter 1

Shared Time Steps

1.1 Estimating the Time Step Size

Alice: Well, what is next? So far we've been using a constant value for the time step. As long as we use softening, that is fine in most cases, at least if we can make sure that particles all move with similar velocities.

Bob: Yes, we could rely on the fact that we had a natural time scale T, given by the time needed for a particle with a typical velocity to move over a distance equal to the softening length. To keep our integration accurate, all we had to do was to make sure that the time step we used was a small fraction f of T.

Alice: But even in that case, we could have run into trouble. If any particle were to move with a velocity larger than the typical velocity by a factor of more than $1/f$, such a particle could cross a distance more than the softening length within one time step. Since the gravitational potential, and hence the gravitational force, could chance significantly over such a distance, such a large time step would spell danger. Already for this reason, it would be better to use adaptive time steps, even if we were to use softening.

Bob: And for our main purpose, modeling dense stellar systems, we are not interested in softening. In our case, we are dealing with point particles, for all practical purposes. Only if stars approach each other within a distance comparable to the sum of their radii, do we have to take into account deviations from point-mass behavior. That deviation, if needed, will then be far more complex than adding a simple form of Plummer softening.

Alice: And if we are dealing with encounters between neutron stars and/or black holes, deviations from point-mass behavior will show up only at tiny distances, of order a hundred kilometers or so.

Bob: So we'll leave softening for those folks who want to study collisionless stellar dynamics, where by definition particle-particle interactions are not con-

sidered to be of interest. And yes, this means that we will have to introduce an adaptive way to determine the time step for each particle.

Alice: For starters, let us stick with a shared time step size, the same for all particles at any given time, but variable over time. What we need to do, is to determine an estimate for the time scale T over which we can expect significant changes to occur in the system. For the shared but variable time step we can then simply use the product of f and T, where f is a constant, with a small value that remains fixed during a run, and T is time dependent, and computed on the fly.

Bob: The most natural guess would be the time scale for collisions to occur. Even if particles are moving head-on toward each other, it will take of order relative distance divided by relative velocity to hit each other. And if they move in different directions, the time scale for significant changes in their relative position will still be given by this ratio.

Alice: But what if the particles happen to move at roughly the same velocity, in both magnitude and direction? In that case your collision time scale estimate will give much too large a number, in fact it will produce infinity if the relative velocity is exactly zero.

Bob: Good point. I guess we need to include another criterion as well. How about a free fall timescale? We can estimate the time it takes for two particles to meet each other, starting at rest. From dimensional analysis it is clear that this time scale must be something like relative distance squared divided by relative acceleration.

Alice: The square root of that quantity, you mean.

Bob: Yes, you should listen to what I mean, not to what I say!

Alice: If you say so; I mean, I'll try!

1.2 Implementation on the `Body` Level

Bob: There will be no ambiguity once we code it up. I'll copy the code `nbody_cst1.rb` into a new file `nbody_sh1.rb`. To start with, I'm going to get rid of all the references to softening. As we just discussed, there is no need to bother with that extra complication once we have an adaptive time step choice.

On the level of the `Body` class, I'll add the following method:

```
def collision_time_scale(body_array)
  time_scale_sq = VERY_LARGE_NUMBER
  body_array.each do |b|
    unless b == self
      r = b.pos - @pos
```

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

Alice: with the dimensional analysis arguments nicely commented in. very nice. Let me follow what you just wrote. You've chosen to work with the square of the time scale, sensibly called `time_scale_sq`, and at the very end you return the square root of that quantity. It is initialized as a very large number, and I presume that for each body that is not the body we are dealing with, you lower the timescale estimate, if that body will encounter our given body within a time shorter than the estimate so far.

As for the details, in the loop where you compare the body `self` with another body `b`, you first estimate the time it would take for the two particles to meet, if they were to move at roughly constant speed. If that estimate would lead to a shorter time scale than the time scale we've obtained so far, we replace the value of `time_scale_sq` by that of `estimate_sq`.

Similarly, when the two body will meet each other in a free fall time that is shorter than our time scale estimate so far, we make the same kind of replacement. Looks good!

1.3 Implementation on the NBody Level

Bob: This gives us an individual time scale, an estimate of the amount of time after which to expect significant changes, but on a per-particle basis. For some particles this will result in a rather long time scale, if they happen to move through the outskirts of an N-body system. Other particles, close to the center, will wind up with a much shorter time scale. In order to play it safe for all particles, we have no choice but to compute the minimum of all these time scale estimates, and use that for the shared time step for the whole system.

Alice: Yes, inefficient as it still is, and soon we'll do better with individual time steps. But at least we're already doing a lot better with shared time steps than we did with constant time steps.

Bob: I sure hope so, but we'd better test that, to make sure that is the case. Here is how we can implement the global time step finder, on the level of the `N-body` class:

```
def collision_time_scale
  time_scale = VERY_LARGE_NUMBER
  @body.each do |b|
    indiv_time_scale = b.collision_time_scale(@body)
    if time_scale > indiv_time_scale
      time_scale = indiv_time_scale
    end
  end
  time_scale
end
```

In addition, we have to make a few more changes. For one thing, the documentation string for the `clop` function has to be changed, and it is a good idea to do that right away, before we forget what it was exactly that we were doing.

Alice: You're really getting organized at your old age . . .

Bob: Just you wait to see how organized I'll be when I reach your age!

Alice: I know, I asked for it. I'll keep my mouth shut.

Bob: In addition, I 'll take out the little trick I had added to make sure that we didn't just miss a time boundary, because of floating point round-off error. In `nbody_cst1.rb` we had:

```
t_dia = @time + dt_dia - 0.5*dt
t_out = @time + dt_out - 0.5*dt
t_end = @time + dt_end - 0.5*dt
```

which in `nbody_sh1.rb` will have to revert to:

```
t_dia = @time + c.dt_dia
t_out = @time + c.dt_out
t_end = @time + c.dt_end
```

Alice: I see. In the constant time step case, you could afford flagging a halting condition half a step before it really had to happen, since there would be no danger that you would stop prematurely, given that the step size was fixed. In the case of variable time steps, you may wind up just barely before the time of an output, and in that case you still have to make an extra step.

Hmmm. This brings us to the question of diagnostics. In the variable time step scheme, you will always overshoot by a fraction of the last time step size, since it will be infinitely unlikely that you happend to land exactly on the intended halting time `t_end`.

Bob: With infinitely large, you mean something like the size of the largest number that can be represented in double precision floating point. Yes, I see what you mean. I guess we could make the last step smaller, adjusting it in such a way that it would exactly hit the required ending time.

Alice: But to be consistent, you would have to do the same thing for *any* time of output, for both particle output and diagnostics output, In other words, you would have to shrink the time step each time you are about to overshoot past a time `t_dia` and `t_out`, in addition to caring about not passing `t_end`.

Bob: We could, but I prefer to not worry about such niceties for now. I'm eager to get to individual time steps. Once we're there, we can try to clean up such type of details.

Alice: I agree, mostly because I see an extra complication lurking. If we would put time steps in a Procrustean bed, cutting off whatever sticks beyond an output time, this would give the following unfortunate side effect. If we ran a simulation from the same initial conditions, but with different intervals for diagnostics output, say, we would force the system to make slightly more time steps for the case for which we required more frequent output. This in turn would slightly modify the orbits of all particles, by modifying the numerical errors involved with integrations with finite step sizes.

Bob: Aha, and since N-body systems are exponentially unstable, even a very small difference will propagate quickly through the system, resulting in a different type of evolution. Indeed, it would be unfortunate if a difference in diagnostics output frequency would alter the orbits of the particles in the system. I agree, better to leave the rough edges, and let particles overshoot a bit. In that way, the integration errors will not be affected.

1.4 Multiple Time Step Schemes

Alice: Those are all the changes that we need to make? It seems almost too simple.

Bob: The real complications will occur once we move to individual time steps.

I can't wait! But let's be careful and first test our shared time step implementation.

Alice: Wait, there is something that still bothers me. Are you sure that all the many integration schemes that you have implemented will still work, across the transition from constant to shared time steps?

Bob: Why not?

Alice: Well, who knows why yes or why not. I want to be sure.

Bob: As long as all particles still share the same step size, at any given time, why would they care what will happen at the next step? They will happily step forwards in lock step at the current step.

Alice: Even for Yoshida's algorithm?

Bob: I would think so. Let's have a look at Yoshida's fourth-order integrator in `nbody_sh1.rb`

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

See, all particles step forward together, then backward together, and then forward again, all jointly. Everything is nicely self-contained, and really, this should work independently of any change in step size. The current step doesn't care what the previous step did or what the next step size will be.

Alice: Okay, Yoshida's algorithms are fine here, I agree. But what about the multistep algorithms? Here is what we have for the case of `nbody_cst1.rb`, for the simplest one, a second-order scheme:

```
def ms2
  if @nsteps == 0
    calc(" @prev_acc = acc(ba,eps) ")
    rk2
  else
    calc(" @old_acc = acc(ba,eps) ")
```

```
      calc(" @jdt = @old_acc - @prev_acc ")
      calc(" @pos += @vel*dt + @old_acc*0.5*dt*dt ")
      calc(" @vel += @old_acc*dt + @jdt*0.5*dt")
      calc(" @prev_acc = @old_acc ")
    end
  end
```

Bob: Aaaahhh, hmmmm, well, you are right, after all, to not trust me jumping to conclusions! This algorithm does *not* carry over in any direct way to shared time steps. Look at the definition of `@jdt`. It is implicitly assumed that the `dt` value used in the previous time step is the same as the `dt` value for the current time step.

Alice: A perfectly correct assumption for the case of constant time steps, which we have been working with so far. It never hurts to check, when you are relaxing conditions you had been relying on so far.

Bob: I admit. Thanks! Well, we have two options. Either we can just omit all multi-step schemes from `nbody_sh1.rb`, or we can try to repair the situation. It shouldn't be too hard to re-derive the expressions while taking into account the fact that each time step has a different size.

Alice: In the spirit of getting on with things, I vote for just leaving them out for now. As you said, by the time we get to individual time steps, we can add these methods again, and clean up the whole code.

Chapter 2

Distance Measures

2.1 Phase Space

Bob: Now that we have a shared time step code `nbody_sh1.rb`, I'd like to make sure that it really gives the same results as the previous code `nbody_cst1.rb` that used constant time steps. Having good energy accuracy is one thing, but it would be much better to check the actual particle positions.

Alice: How about computing the distance between the results of two runs, in 6N-dimensional phase space? A whole N-body system can be viewed as a single point in a space that spans all the degrees of freedom of the positions and velocities of all particles.

More generally, if we model a system in D dimensions, a single particle will have D components for its position as well as its velocity, or 2D in total. N particles will have 2DN such degrees of freedom. For planar N-body systems, we need a 4N-dimensional phase space, and for N-body systems in the real three-dimensional world we are dealing with a 6N-dimensional phase space.

Bob: In practice this means, I presume, that you use Pythagoras to compute the distance in such a large space. In other words you take the square root of the sum of the squares of all the differences, right?

Alice: Indeed.

Bob: So another way of looking at your suggestion is that you compute the average distance between corresponding particles in the usual 6-dimensional phase space. Apart from a normalization factor N, the result would be the same.

Alice: Yes, that's right. I like the more abstract picture, but your description boils down to the same thing.

Bob: I find it a lot easier to think about 2D dimensions than about 2ND

dimensions. In either case, the practical procedure is that we have to subtract two N-body systems, point by point, and then we have to compute the norm or 'size' of the resulting system, a type of absolute value.

Alice: Ah, so you *do* like abstract thinking, after all! How about defining a minus operator, that allows us to subtract two N-body systems, `nb1` and `nb2`, just by writing `nb1-nb2`? That would give us a nicely compact notation. And the 2DN-dimensional distance between the two systems could then be given by `(nb1-nb2).abs` if we define the correct absolute value operator `abs` for the `NBody` class.

Bob: Good idea! However, there is no guarantee that two N-body systems will show their particles in the output in the exact same order. In other words, what we need is a way to identify which particle in the one system corresponds to which particle in the other system.

Alice: I guess we have to number them, by giving each body a unique identifier.

2.2 Numbering Bodies

Bob: Given that every object in Ruby already has an identifier called `object_id`, a natural name for such an ID would be `body_id`. Now that is easy to implement. How about writing a nifty tool like this, in file `nbody_set_id.rb`:

```
#!/usr/local/bin/ruby -w

require "nbody.rb"

options_text= <<-END

  Description: Takes an N-body system, and gives each body a unique ID
  Long description:
    This program accepts an N-body system, and assigns a number to each
    body consecutively, as an instance variable @body_id which takes on
    integer values.

    (c) 2005, Piet Hut, Jun Makino; see ACS at www.artcompsi.org

    example:
              kali #{$0} -n 0


  Short name: -n
  Long name:            --starting_number
  Value type:           int
```

```
  Default value:        1
  Description:          value of @body_id for 1st body
  Variable name:        n
  Long description:
    This option allows the user to start numbering the particles at a
    value different from the default value (0, say, or 10, or whatever).


  END

c = parse_command_line(options_text)

class Body
  attr_accessor :body_id
end

nb = ACS_IO.acs_read(NBody)
i = c.n
nb.body.each do |b|
  b.body_id = i
  i += 1
end
nb.acs_write($stdout, false, c.precision)
```

2.3 A Test Count

You see, I even allowed for taste: the default is to start numbering at one, but you can start at zero as well, or at any other place you like. Let's test it with our Plummer Model:

```
|gravity> kali nbody_set_id.rb -h
Takes an N-body system, and gives each body a unique ID
-n  --starting_number: value of @body_id for 1st body [default: 1]
--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 mkplummer.rb -n 2 | kali nbody_set_id.rb --precision 5
==> Takes an N-body system, and gives each body a unique ID <==
```

```
value of @body_id for 1st body: n = 1
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 5
Incremental indentation: add_indent = 2
==> Plummer's Model Builder <==
Number of particles: N = 2
pseudorandom number seed given: 0
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: 1121228501
ACS
  NBody
    Array body
      Body body[0]
        Fixnum body_id
          1
        Float mass
             5.00000e-01
        Vector pos
            -2.04749e-01  -1.07460e-01  -9.50279e-02
        Vector vel
             4.00520e-01   5.46513e-01  -2.02256e-01
      Body body[1]
        Fixnum body_id
          2
        Float mass
             5.00000e-01
        Vector pos
             2.04749e-01   1.07460e-01   9.50279e-02
        Vector vel
            -4.00520e-01  -5.46513e-01   2.02256e-01
    Float time
         0.00000e+00
SCA

|gravity> kali mkplummer.rb -n 2 | kali nbody_set_id.rb --precision 5 -n 42
==> Takes an N-body system, and gives each body a unique ID <==
value of @body_id for 1st body: n = 42
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 5
```

```
Incremental indentation: add_indent = 2
==> Plummer's Model Builder <==
Number of particles: N = 2
pseudorandom number seed given: 0
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: 1121594575
ACS
  NBody
    Array body
      Body body[0]
        Fixnum body_id
          42
        Float mass
             5.00000e-01
        Vector pos
            -4.47331e-02   1.14152e-01  -2.17872e-01
        Vector vel
            -1.32748e-01   2.20825e-01   6.58494e-01
      Body body[1]
        Fixnum body_id
          43
        Float mass
             5.00000e-01
        Vector pos
             4.47331e-02  -1.14152e-01   2.17872e-01
        Vector vel
             1.32748e-01  -2.20825e-01  -6.58494e-01
    Float time
         0.00000e+00
SCA
```

2.4 Defensive Programming

Bob: Now, with identifiers in place, we can extend our `nbody.rb` file by adding a subtraction operator:

```
  def -(other)
```

```
    if other.class != self.class
      raise "other.class.name = #{other.class.name} != #{self.class.name}"
    end
    if (n = other.body.size) != body.size
      raise "other.body.size = #{other.body.size} != #{body.size}"
    end
    nb = NBody.new
    body.each_index do |i|
      if (id = body[i].body_id) == nil
        raise "body[#{i}].body_id == nil"
      end
      ob = other.body.find{|oi| oi.body_id == id}
      if ob == nil
        raise "body_id = #{body[i].body_id} not found in other N-body system"
      end
      nb.body[i] = Body.new
      nb.body[i].pos = body[i].pos - ob.pos
      nb.body[i].vel = body[i].vel - ob.vel
    end
    nb
  end
```

This includes proper testing of particle identities.

Alice: And I see that you are returning the newly constructed nbody system `nb` at the end of the method. Also, you're getting quite defensive in your programming!

Bob: This is a situation where defensive programming is called for. In general, I don't like to litter my code with many error checking lines all over the place, but in this particular case, it is quite likely that the difference operator will be invoked for incompatible N-body systems. So I'm checking whether the other object is really of `NBody` class, whether it has the same number of particles as the calling `NBody` object, and most importantly, I am then checking whether each particle in the calling system has a counter part in the other system, with the same `body_id`.

Alice: Applause, applause! As you know, I like to check things, as a matter of habit, in order to prevent future surprises that may be hard to debug. But I agree, you can overdo with anything, and there is a lot to say for keeping code small. In this case, though, the whole code is so small that it doesn't distract to have these extra safeguards built in.

2.5 Putting it together

Bob: With our new tools in hand, it now becomes really trivial to check for the actual size of the difference between two N-body systems, by defining the absolute value method, also within the file `nbody.rb`:

```
def abs
  a = 0
  body.each{ |b| a += b.pos*b.pos + b.vel*b.vel }
  sqrt a
end
```

Alice: Sometimes we may want to look only at the positions, restricting ourselves to configuration space, rather than to phase space.

Bob: That's easy. Here is a position-only version of `abs`:

```
def abs_pos
  a = 0
  body.each{ |b| a += b.pos*b.pos }
  sqrt a
end
```

Alice: Let me try to implement the module that gives the difference between two N-body systems, as the 6N-dimensional distance in phase space, or the 2DN-dimensional distance for the general case of D spatial dimensions. It's time to practice my Ruby skills again. Let's call the file `nbody_diff.rb`:

```
#!/usr/local/bin/ruby -w

require "nbody.rb"

options_text= <<-END

  Description: 6N-dimensional phase space distance between two N-body systems
  Long description:
    This program accepts two N-body systems, and computes the distance between
    them in the 2DN-dimensional phase space of all positions and velocities,
    if they are given in D spatial dimensions.  The values of the masses of
    the particles are not used, and each particle is given equal weight.  In
    order to compare corresponding particles in the two systems, the body_id
    instance variable of each body is used as an identifier.
```

```
    (c) 2005, Piet Hut, Jun Makino; see ACS at www.artcompsi.org


  Short name: -r
  Long name:            --positions_only
  Value type:           bool
  Description:          Using only positions, not velocities
  Variable name:        position_only_flag
  Long description:
    This option allows the user to compute the distance between two N-body
    systems using only particle information; velocity information is not used.


  END

c = parse_command_line(options_text)

class Body
  attr_accessor :body_id
end

nb1 = ACS_IO.acs_read(NBody)
nb2 = ACS_IO.acs_read(NBody)
nb = nb1 - nb2
d = nb.body[0].pos.size
n = nb.body.size
print "#{2*d}N-dim. phase space dist. for two #{n}-body systems:  "
if c.position_only_flag == false
  printf(" %.#{c.precision}e\n", nb.abs)
else
  printf(" %.#{c.precision}e\n", nb.abs_pos)
end
```

How about this?

2.6 Checking it out

Bob: Looks fine, as far as I can see. You put in options to allow for more precision, without burdening the reader with all those extra digits in the general case. And you also gave an option for positions-only comparisons.

Time to test it! We still have the initial conditions lying around for a circular binary, I believe. Ah yes, here it is, in `circular_binary.in`:

```
ACS
  NBody
    Array body
      Body body[0]
        Float mass
             4
        Vector pos
             1  0
        Vector vel
             0  1
      Body body[1]
        Float mass
             4
        Vector pos
             -1  0
        Vector vel
             0  -1
SCA
```

To start with, let's see whether two identical systems are assigned a distance of zero correctly:

```
|gravity> kali nbody_set_id.rb < circular_binary.in > tmp0.in
==> Takes an N-body system, and gives each body a unique ID <==
value of @body_id for 1st body: 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

|gravity> cat tmp0.in tmp0.in | kali nbody_diff.rb
==> 6N-dimensional phase space distance between two N-body systems <==
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
4N-dim. phase space dist. for two 2-body systems:   0.0000000000000000e+00
```

Alice: Good! Let's perturb our binary slightly. If we use the most popular Pythagorean triangle, we can immediately check whether our Cartesian distances in phase space come out correctly. Here is a file `pert_circ_binary.in`:

```
ACS
  NBody
    Array body
      Body body[0]
        Float mass
             4
        Vector pos
             1  0.03
        Vector vel
             0  1
      Body body[1]
        Float mass
             4
        Vector pos
             -1  0
        Vector vel
             0  -1.04
SCA
```

And here is our first real result:

```
|gravity> kali nbody_set_id.rb < pert_circ_binary.in > tmp1.in
==> Takes an N-body system, and gives each body a unique ID <==
value of @body_id for 1st body: 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

|gravity> cat tmp0.in tmp1.in | kali nbody_diff.rb
==> 6N-dimensional phase space distance between two N-body systems <==
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
4N-dim. phase space dist. for two 2-body systems:   5.0000000000000031e-02
```

Just what it should be! Now for positions only, and a higher precision:

```
|gravity> cat tmp0.in tmp1.in | kali nbody_diff.rb -r --precision 5
==> 6N-dimensional phase space distance between two N-body systems <==
Using only positions, not velocities
```

```
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 5
Incremental indentation: add_indent = 2
4N-dim. phase space dist. for two 2-body systems:   3.00000e-02
```

Perfect. I think we're ready to move on.

Chapter 3

A Puzzle

3.1 Starting with Runge-Kutta

Bob: We can now put our distance measurement tool to good use, by testing whether our shared time step integrator really does the right thing. We'll do a test run with the constant time step code first, and then compare the results with that from the shared time step code.

Alice: Let's take an unusual integrator. How about the good old fourth order Runge-Kutta scheme?

Bob: As you wish. And while we're at it, let's check whether the scheme is still fourth order, in constant as well as shared time step form. First I'll prepare initial conditions with the particles all properly numbered:

```
|gravity> kali mkplummer.rb -n 4 -s 1 | kali nbody_set_id.rb > test.in
==> Takes an N-body system, and gives each body a unique ID <==
value of @body_id for 1st body: 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
==> 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
```

Starting with the constant time step version, let's choose the `rk4` integrator, but what options did we use for algorithm choice? I always forget what names we gave our options.

Alice: Let's ask our clop function to provide them:

```
|gravity> kali nbody_cst1.rb -h
Constant Time Step Code
-g  --integration_method: Integration method [default: hermite]
-s  --softening_length: Softening length [default: 0.0]
-c  --step_size: Time step size [default: 0.001]
-d  --diagnostics_interval: Interval between diagnostics output [default: 1]
-o  --output_interval: Time interval between snapshot output [default: 1]
-t  --duration: Duration of the integration [default: 10]
-i  --init_out: Output the initial snapshot
--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)
```

3.2 Energy Scaling

Bob: Aha, now I remember. Okay, how about this?

```
|gravity> kali nbody_cst1.rb -g rk4 -t 1 < test.in > rk4cst.out
==> Constant Time Step Code <==
Integration method: method = rk4
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 = 0.0671 , E_pot =  -0.317 , E_tot = -0.25
            E_tot - E_init = -1.68e-08
  (E_tot - E_init) / E_init = 6.7e-08
```

Now let's make the time step two times smaller

```
|gravity> kali nbody_cst1.rb -g rk4 -t 1 -c 0.0005 < test.in > rk4cst_half.out
==> Constant Time Step Code <==
Integration method: method = rk4
Softening length: eps = 0.0
Time step size: dt = 0.0005
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 2000 steps :
  E_kin = 0.0671 , E_pot =  -0.317 , E_tot = -0.25
            E_tot - E_init = -5.24e-10
  (E_tot - E_init) / E_init = 2.1e-09
```

3.3 Phase Space Distance Scaling

Alice: Well, that doesn't look like a fourth-order behavior. The second run was far too accurate, as if it were a fifth-order algorithm. The reason may well be that running constant time steps in a system where particles are allowed to come arbitrarily close to each other is not a good thing. We really should have used softening.

Bob: That's easy to test. Here you are:

```
|gravity> kali nbody_cst1.rb -s 0.01 -g rk4 -t 1 -c 0.001 < test.in > /dev/null
==> Constant Time Step Code <==
Integration method: method = rk4
Softening length: eps = 0.01
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 = 0.0658 , E_pot =  -0.316 , E_tot = -0.25
            E_tot - E_init = -3.08e-08
  (E_tot - E_init) / E_init = 1.23e-07

|gravity> kali nbody_cst1.rb -s 0.01 -g rk4 -t 1 -c 0.0005 < test.in > /dev/null
==> Constant Time Step Code <==
Integration method: method = rk4
Softening length: eps = 0.01
Time step size: dt = 0.0005
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 2000 steps :
  E_kin = 0.0658 , E_pot =  -0.316 , E_tot = -0.25
            E_tot - E_init = -9.65e-10
  (E_tot - E_init) / E_init = 3.86e-09
```

It still behaves like a fifth-order scheme, with an improvement of about a factor $2^5 = 32$ instead of the expected $2^5 = 16$.

Alice: Hmm, that's surprising. In any case, we don't want to use softening in

our shared time step code, so we should find a reasonable run for our constant time step code without softening. How about making the total time integration ten times shorter?

Bob: I'm all for speeding things up. Here is the original set, now shortened:

```
|gravity> kali nbody_cst1.rb -g rk4 -t 0.1 -d 0.1 -o 0.1 < test.in > rk4cst.out
==> Constant Time Step Code <==
Integration method: method = rk4
Softening length: eps = 0.0
Time step size: dt = 0.001
Interval between diagnostics output: dt_dia = 0.1
Time interval between snapshot output: dt_out = 0.1
Duration of the integration: t = 0.1
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 = 0.1, after 100 steps :
  E_kin = 0.653 , E_pot =  -0.903 , E_tot = -0.25
            E_tot - E_init = 7.77e-10
  (E_tot - E_init) / E_init = -3.11e-09

|gravity> kali nbody_cst1.rb -g rk4 -c 0.0005 -t 0.1 -d 0.1 -o 0.1 < test.in > rk4cst_half.out
==> Constant Time Step Code <==
Integration method: method = rk4
Softening length: eps = 0.0
Time step size: dt = 0.0005
Interval between diagnostics output: dt_dia = 0.1
Time interval between snapshot output: dt_out = 0.1
Duration of the integration: t = 0.1
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 = 0.1, after 200 steps :
  E_kin = 0.653 , E_pot =  -0.903 , E_tot = -0.25
            E_tot - E_init = 4.87e-11
```

```
  (E_tot - E_init) / E_init = -1.95e-10
```

Alice: And now it scales indeed like fourth order. I don't like the fact that we don't have a good explanation for this phenomenon, but I guess the only thing that is really guaranteed in an integration scheme is that things scale at least as good as the order they are supposed to have. For now, let us test things with the shorter run.

Before we go any further, and use our shared time step integrator, let's first see whether in this case the distance in phase space also scales like the fourth power in the time step.

Bob: Ah, yes, good idea. But in order to do so, we have to know what the 'true' solution is. When we check energy conservation, we know that a truly accurate integration would give us an energy error of zero, but when we compare the two outputs we just got, we need to compare them to an even more accurate soluation. Well, we can run our code with a much smaller time step, and declare that the result should be close to 'true'. Or to speed things up a bit, we can take a higher-order integrator. Why not throw the sixth-oder Yoshida version in, to help us:

```
|gravity> kali nbody_cst1.rb -g yo6 -t 0.1 -d 0.1 -o 0.1 < test.in > yo6cst.out
==> Constant Time Step Code <==
Integration method: method = yo6
Softening length: eps = 0.0
Time step size: dt = 0.001
Interval between diagnostics output: dt_dia = 0.1
Time interval between snapshot output: dt_out = 0.1
Duration of the integration: t = 0.1
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 = 0.1, after 100 steps :
  E_kin = 0.653 , E_pot =  -0.903 , E_tot = -0.25
            E_tot - E_init = -6.56e-13
  (E_tot - E_init) / E_init = 2.62e-12
```

First we'll compare the first run with the 'true' run:

```
|gravity> cat rk4cst.out yo6cst.out | kali nbody_diff.rb
==> 6N-dimensional phase space distance between two N-body systems <==
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:   3.8774928642689844e-09
```

And then we'll take the second run, to compare it with the 'true' one:

```
|gravity> cat rk4cst_half.out yo6cst.out | kali nbody_diff.rb
==> 6N-dimensional phase space distance between two N-body systems <==
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:   2.4252715911265379e-10
```

Alice: Very nice indeed, almost exactly a factor 2^4 smaller.

3.4 The Shared Time Step Code

Bob: After all these preliminaries, we can finally test our shared time step code:

```
|gravity> kali nbody_sh1.rb -g rk4 -t 0.1 -d 0.1 -o 0.1 < test.in > rk4sh.out
==> Shared Time Step Code <==
Integration method: method = rk4
Parameter to determine time step size: dt_param = 0.01
Interval between diagnostics output: dt_dia = 0.1
Time interval between snapshot output: dt_out = 0.1
Duration of the integration: t = 0.1
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 = 0.100292, after 120 steps :
```

```
E_kin = 0.656 , E_pot =  -0.906 , E_tot = -0.25
          E_tot - E_init = 1.51e-10
(E_tot - E_init) / E_init = -6.05e-10
```

Not quite as accurate as the comparable run for constant step size, even though we used slightly more steps.

Alice: I guess that is the price we have to pay for constructing a more robust code.

Bob: Yes, that must be the case. In general, when you're developing a more complex algorithm, you have to pay an initial price of a reduce efficiency.

But we had no choice. In fact, we were lucky that we got such a good result with a constant time step code, without using any softening. If we would have repeated those runs with other Plummer realizations, sooner or later we would have found ourselves in a situation in which two particles have a close encounter at a distance less than the step size, leading to huge errors. That at least cannot happen in our shared time step code, where the adaptive time step determination would always shrink the time step to be much less than the encounter time.

Alice: This may be the first time I've heard you arguing eloquently for reducing efficiency!

Bob: Only when necessary! Now let's see whether we're getting equally close to the 'true' run in phase space:

```
|gravity> cat rk4sh.out yo6cst.out | kali nbody_diff.rb
==> 6N-dimensional phase space distance between two N-body systems <==
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:   8.6877211225495372e-03
```

Alice: What is that?!?

Bob: That sure doesn't look good. How can the distance be so large?

3.5 Back to Basics

Alice: Let's go back to basics, and do the same thing with the leapfrog integrator instead. And since we'll have lower accuracy, we may as well go to a comparison with time steps that are a factor of ten smaller; we won't have to be afraid to run into round-off errors:

```
|gravity> kali nbody_sh1.rb -g leapfrog -t 0.1 -d 0.1 -o 0.1 < test.in > leap_sh.out
==> Shared Time Step Code <==
Integration method: method = leapfrog
Parameter to determine time step size: dt_param = 0.01
Interval between diagnostics output: dt_dia = 0.1
Time interval between snapshot output: dt_out = 0.1
Duration of the integration: t = 0.1
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 = 0.100292, after 120 steps :
  E_kin = 0.656 , E_pot =  -0.906 , E_tot = -0.25
            E_tot - E_init = 9.19e-06
  (E_tot - E_init) / E_init = -3.67e-05

|gravity> kali nbody_sh1.rb -g leapfrog -c 0.001 -t 0.1 -d 0.1 -o 0.1 < test.in > leap_sh_ten.ou
==> Shared Time Step Code <==
Integration method: method = leapfrog
Parameter to determine time step size: dt_param = 0.001
Interval between diagnostics output: dt_dia = 0.1
Time interval between snapshot output: dt_out = 0.1
Duration of the integration: t = 0.1
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 = 0.1, after 1199 steps :
  E_kin = 0.653 , E_pot =  -0.903 , E_tot = -0.25
            E_tot - E_init = 9.07e-08
  (E_tot - E_init) / E_init = -3.63e-07
```

Bob: The energy scales like it should for a second-order integrator: a hundred times better accuracy for a ten times smaller time step.

Alice: And, really, the phase space distance should also become a hundred times better. Let's see whether it does:

```
|gravity> cat leap_sh.out yo6cst.out | kali nbody_diff.rb
==> 6N-dimensional phase space distance between two N-body systems <==
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:   8.7147418125614710e-03

|gravity> cat leap_sh_ten.out yo6cst.out | kali nbody_diff.rb
==> 6N-dimensional phase space distance between two N-body systems <==
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:   6.4209578482452148e-06
```

3.6 Even more Basic

Bob: No, it doesn't. Given the fact that energy conservation was scaling so perfectly like second-order, it is very hard to believe that the individual positions and velocities wouldn't scale in the same way. What is going on here?

Alice: Let's completely forget about higher-order integrators, and stick to the basics completely. If we do a third leapfrog integration, we can use the result of that run as our yard stick:

```
|gravity> kali nbody_sh1.rb -g leapfrog -c 0.0001 -t 0.1 -d 0.1 -o 0.1 < test.in > lea
==> Shared Time Step Code <==
Integration method: method = leapfrog
Parameter to determine time step size: dt_param = 0.0001
Interval between diagnostics output: dt_dia = 0.1
Time interval between snapshot output: dt_out = 0.1
Duration of the integration: t = 0.1
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 = 0.100002, after 11997 steps :
  E_kin = 0.653 , E_pot =  -0.903 , E_tot = -0.25
```

```
            E_tot - E_init = 9.06e-10
  (E_tot - E_init) / E_init = -3.62e-09

|gravity> cat leap_sh.out leap_sh_hundred.out | kali nbody_diff.rb
==> 6N-dimensional phase space distance between two N-body systems <==
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:   8.6693930283109049e-03

|gravity> cat leap_sh_ten.out leap_sh_hundred.out | kali nbody_diff.rb
==> 6N-dimensional phase space distance between two N-body systems <==
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:   3.8930612753114946e-05
```

Bob: That doesn't make much difference. Still no quadratic behavior.

Alice: And even if it were quadratic, I'm still worried about the large magnitude of the phase space distance for our fourth-order integrator. And in fact, these leapfrog phase space distances are also larger than I would have expected.

Bob: All this should tell us something. But what?

Alice: Beats me.

Bob: This is ennoying. Just when we were having fun!

Alice: How about a cup of tea first?

Bob: Feel free to get some tea, but I'll stay here. I just can't stand this. What could be going so terribly wrong here?

Chapter 4

The Answer

4.1 Inspection

Alice: Here is a cup of tea. I figured you'd need one!

Bob: Thanks! I've been scratching my head, but pure thought hasn't given me an answer yet.

Alice: Well, debugging is part of life, at least if your life is tied up with simulations. When in doubt, sort it out. Let's have a look at what is going on under the hood. How about just printing out some output files?

Bob: When in doubt, print it out, you mean. Okay, here are the first two leapfrog files:

```
|gravity> cat leap_sh.out
ACS
  NBody
    Array body
      Body body[0]
        Fixnum body_id
          1
        Float mass
             2.5000000000000000e-01
        Vector pos
            -1.3485433265148719e-01  -1.7846863689302612e+00   1.9423884773847429e+00
        Vector vel
            -9.3431441329143750e-01   9.0729056804334129e-01  -7.0321473505957210e-01
      Body body[1]
        Fixnum body_id
          2
```

```
        Float mass
             2.5000000000000000e-01
        Vector pos
             5.6537185039744553e-01   5.3929907883704606e+00  -5.5086798830929160e+00
        Vector vel
             1.4602083668477447e-01  -1.7273555866326543e-01   2.1676968279098491e-01
      Body body[2]
        Fixnum body_id
          3
        Float mass
             2.5000000000000000e-01
        Vector pos
            -2.3413793812618905e-01  -1.7760426985236157e+00   1.9883052266031183e+00
        Vector vel
             1.2704758045698876e+00  -8.8114874428175316e-01  -6.8945280017493504e-02
      Body body[3]
        Fixnum body_id
          4
        Float mass
             2.5000000000000000e-01
        Vector pos
            -1.9637957961976896e-01  -1.8322617209165852e+00   1.5779861791050518e+00
        Vector vel
            -4.8218222796322568e-01   1.4659373490167774e-01   5.5539033228608181e-01
    Float dt
         3.8100707056017835e-04
    Float e0
        -2.4999999999999994e-01
    Fixnum nsteps
      120
    Float time
         1.0029235235688644e-01
SCA

|gravity> cat leap_sh_ten.out
ACS
  NBody
    Array body
      Body body[0]
        Fixnum body_id
          1
        Float mass
             2.5000000000000000e-01
        Vector pos
            -1.3458306581606688e-01  -1.7849510425939628e+00   1.9425942533626255e+00
        Vector vel
```

```
            -9.2875251669016978e-01   9.0687573652006714e-01  -7.0522667804419537e-01
      Body body[1]
        Fixnum body_id
          2
        Float mass
             2.5000000000000000e-01
        Vector pos
             5.6532919160736717e-01   5.3930412514009145e+00  -5.5087432103200484e+00
        Vector vel
             1.4602098761528348e-01  -1.7273411879120573e-01   2.1676821342245239e-01
      Body body[2]
        Fixnum body_id
          3
        Float mass
             2.5000000000000000e-01
        Vector pos
            -2.3450745205870957e-01  -1.7757856949846311e+00   1.9883247443866785e+00
        Vector vel
             1.2649624829814954e+00  -8.8061065502478997e-01  -6.6001587327911934e-02
      Body body[3]
        Fixnum body_id
          4
        Float mass
             2.5000000000000000e-01
        Vector pos
            -1.9623867373259168e-01  -1.8323045138223193e+00   1.5778242125707438e+00
        Vector vel
            -4.8223095390661258e-01   1.4646903729592900e-01   5.5446005194965620e-01
    Float dt
         3.8049396181286760e-05
    Float e0
        -2.4999999999999994e-01
    Fixnum nsteps
      1199
    Float time
         1.0000021073044210e-01
SCA
```

4.2 Of Course!

Alice: And look, there is the answer! The final output times are *quite* different.

Bob: Indeed! And of course!! Remember our discussion when we wrote the shared timestep code? We realized that we would overshoot, but we decided

not to care. Well, a constant time step scheme can stop properly after one time unit, but a shared time step code is guaranteed to overshoot, unless we teach it to halt at exactly the required time.

Alice: And we decided not to do that, because we did not want to influence the integration. If you restart a code from a previous output, say at time `t=1`, it would be nice to get the same results at a later time, `t=2` for example, as if you had made a single run directly to `t=2`.

Bob: Yes, that was a good argument, but we can't have our cake and eat it. At least for testing purposes, we should be able to ask the code to stop at *exactly* the time we want it to stop.

Alice: Perhaps we *can* have our cake and eat it, if we build in an extra option. Running the code with the option `--exact_time`, if we want to call it that way, should produce all diagnostics at the exact time that we order them. For our current purposes, that would be ideal. The default should be to have only approximate obedience to time requests, in order to allow accurate restarts. So this extra option should be a boolean flag.

Bob: That's easy to implement. How about this:

```
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
```

I have added an extra `if` statement at the top of the `while` loop, which causes the time step to shorten whenever it threatens to overshoot the next output time `t_out`. Note that I subtracted a very small amount from `t_out` in the last `if` statement, just in case round-off errors would make `@time` ever so slightly smaller than the target time `t_out`. In this way, even with a small roundoff error, `@time` should still be larger than `t_out - 1.0/VERY_LARGE_NUMBER`.

And to make it complete, we should implement the option. Here is the full new list:

```
|gravity> kali nbody_sh1.rb -h
Shared Time Step Code
-g  --integration_method: Integration method [default: hermite]
-c  --step_size_control: Parameter to determine time step size [default: 0.01]
-d  --diagnostics_interval: Interval between diagnostics output [default: 1]
-o  --output_interval: Time interval between snapshot output [default: 1]
-t  --duration: Duration of the integration [default: 10]
--exact_time: Force all outputs to occur at the exact times
-i  --init_out: Output the initial snapshot
--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)
```

4.3 The Right Thing

Alice: Let's inspect an output file, to make sure it halts correctly.

Bob: Let's go back to our original Runge-Kutta tests. Here you are:

```
|gravity> kali nbody_sh1.rb -g rk4 --exact_time -t 0.1 -d 0.1 -o 0.1 < test.in > rk4sh.out
==> Shared Time Step Code <==
Integration method: method = rk4
Parameter to determine time step size: dt_param = 0.01
Interval between diagnostics output: dt_dia = 0.1
Time interval between snapshot output: dt_out = 0.1
Duration of the integration: t = 0.1
```

```
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 = 0.1, after 120 steps :
  E_kin = 0.653 , E_pot =  -0.903 , E_tot = -0.25
             E_tot - E_init = 1.51e-10
  (E_tot - E_init) / E_init = -6.03e-10

|gravity> cat rk4sh.out
ACS
  NBody
    Array body
      Body body[0]
        Vector a0
            -1.8878401707429514e+01   1.5045399208144028e+00   6.7181937871389596e+00
        Vector a1
            -1.8902383619991330e+01   1.4934117988123523e+00   6.7421954611113950e+00
        Vector a2
            -1.8926393301847646e+01   1.4822267755578977e+00   6.7662713965217129e+00
        Fixnum body_id
          1
        Float mass
             2.5000000000000000e-01
        Vector old_pos
            -1.3450034339447617e-01  -1.7850318892110921e+00   1.9426571576309333e+00
        Vector pos
            -1.3458287877818725e-01  -1.7849512304644410e+00   1.9425944023829409e+00
        Vector vel
            -9.2874843192054990e-01   9.0687549615491125e-01  -7.0522822241453442e-01
      Body body[1]
        Vector a0
            -5.1658665719092796e-04  -4.9285962390673427e-03   5.0295623846555006e-03
        Vector a1
            -5.1659512158489407e-04  -4.9286114507233574e-03   5.0295772659213617e-03
        Vector a2
            -5.1660358657626079e-04  -4.9286266622756234e-03   5.0295921475101649e-03
        Fixnum body_id
          2
        Float mass
             2.5000000000000000e-01
```

```
    Vector old_pos
         5.6531617257905042e-01   5.3930566521127625e+00  -5.5087625370495452e+00
    Vector pos
         5.6532916083626628e-01   5.3930412878011911e+00  -5.5087432559996499e+00
    Vector vel
         1.4602098772415720e-01  -1.7273411775258923e-01   2.1676821236255386e-01
  Body body[2]
    Vector a0
         1.8714917434208008e+01  -1.9252842248874613e+00  -9.9018004047685757e+00
    Vector a1
         1.8738635850576095e+01  -1.9143164516612177e+00  -9.9266128888697818e+00
    Vector a2
         1.8762381982130638e+01  -1.9032919056069417e+00  -9.9514999345088526e+00
    Fixnum body_id
      3
    Float mass
         2.5000000000000000e-01
    Vector old_pos
        -2.3462015147249890e-01  -1.7757071919479690e+00   1.9883305875917985e+00
    Vector pos
        -2.3450771027838960e-01  -1.7757855127984576e+00   1.9883247563900657e+00
    Vector vel
         1.2649584368307951e+00  -8.8061032500286318e-01  -6.5999367413200699e-02
  Body body[3]
    Vector a0
         1.6400085987869584e-01   4.2567290031212596e-01   3.1785770552449608e+00
    Vector a1
         1.6426436453681953e-01   4.2583326429958879e-01   3.1793878504924646e+00
    Vector a2
         1.6452792330358335e-01   4.2599375671131956e-01   3.1801989458396305e+00
    Fixnum body_id
      4
    Float mass
         2.5000000000000000e-01
    Vector old_pos
        -1.9619567771207611e-01  -1.8323175709536976e+00   1.5777747918268183e+00
    Vector pos
        -1.9623857177969017e-01  -1.8323045445382884e+00   1.5778240972266480e+00
    Vector vel
        -4.8223099263440355e-01   1.4646894660054141e-01   5.5445937746518059e-01
Float dt
     8.8947865472421839e-05
Float e0
    -2.4999999999999994e-01
Fixnum nsteps
  120
```

```
    Float time
         1.0000000000000001e-01
SCA
```

Alice: Good! Indeed at time 1. Let's repeat our comparison with the constant time step run.

Bob: That should come out a whole lot better now! Here we go:

```
|gravity> cat rk4sh.out rk4cst.out | kali nbody_diff.rb
==> 6N-dimensional phase space distance between two N-body systems <==
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:   3.4106773152746560e-09
```

Alice: And right you are. That's more like it. Wonderful!

4.4 Setting up the Leapfrog

Bob: Let's repeat our leapfrog test suite, just to make sure that everything behaves the way it should. It is hard to use a sixth-order integrator, since before you know it you are up to the round-off barrier.

Alice: I agree. Always good to make an extra test.

Bob: Here are the three output files, now finished at the exact time:

```
|gravity> kali nbody_sh1.rb -g leapfrog --exact_time -t 0.1 -d 0.1 -o 0.1 < test.in >
==> Shared Time Step Code <==
Integration method: method = leapfrog
Parameter to determine time step size: dt_param = 0.01
Interval between diagnostics output: dt_dia = 0.1
Time interval between snapshot output: dt_out = 0.1
Duration of the integration: t = 0.1
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 = 0.1, after 120 steps :
  E_kin = 0.653 , E_pot =  -0.903 , E_tot = -0.25
            E_tot - E_init = 9.17e-06
  (E_tot - E_init) / E_init = -3.67e-05

|gravity> kali nbody_sh1.rb -g leapfrog --exact_time -c 0.001 -t 0.1 -d 0.1 -o 0.1 < test.in > l
==> Shared Time Step Code <==
Integration method: method = leapfrog
Parameter to determine time step size: dt_param = 0.001
Interval between diagnostics output: dt_dia = 0.1
Time interval between snapshot output: dt_out = 0.1
Duration of the integration: t = 0.1
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 = 0.1, after 1199 steps :
  E_kin = 0.653 , E_pot =  -0.903 , E_tot = -0.25
            E_tot - E_init = 9.07e-08
  (E_tot - E_init) / E_init = -3.63e-07

|gravity> kali nbody_sh1.rb -g leapfrog --exact_time -c 0.0001 -t 0.1 -d 0.1 -o 0.1 < test.in >
==> Shared Time Step Code <==
Integration method: method = leapfrog
Parameter to determine time step size: dt_param = 0.0001
Interval between diagnostics output: dt_dia = 0.1
Time interval between snapshot output: dt_out = 0.1
Duration of the integration: t = 0.1
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 = 0.1, after 11997 steps :
```

```
E_kin = 0.653 , E_pot =  -0.903 , E_tot = -0.25
          E_tot - E_init = 9.06e-10
(E_tot - E_init) / E_init = -3.62e-09
```

4.5 Testing the Leapfrog

Alice: I bet this will come out correctly quadratic.

Bob: I think so to, but seeing is believing:

```
|gravity> cat leap_sh.out leap_sh_hundred.out | kali nbody_diff.rb
==> 6N-dimensional phase space distance between two N-body systems <==
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:   2.4184021559828722e-05

|gravity> cat leap_sh_ten.out leap_sh_hundred.out | kali nbody_diff.rb
==> 6N-dimensional phase space distance between two N-body systems <==
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:   2.3758207012759387e-07
```

Alice: We see, we believe, we are delighted!

Bob: You may say so! I am really relieved. I was so frustrated, when we found that linear behavior! But now all is well.

Chapter 5

Reproducibility Lost

5.1 Making Sure

Alice: Now that everything seems to work, I would like to get back to this question of reproducibility. We have argued that we can integrate a system for a while, get an output, and then restart a second integration from that ouput to a finishing time, while getting the exact same result as when we would have done a run from the beginning directly up to this finishing time.

Bob: Unless we set the `--exact_time` flag, in which case this no longer holds.

Alice: I think you're right, but I'd like to make sure. First, I would like to check whether we get exact conservation if we don't set the `--exact_time` flag. And then I'd like to see how much difference it makes, if we set that flag.

Bob: I'm game. It will certainly give us more confidence in our codes, if they do what they should be doing. But let's start at the beginning, with the constant time step code. Let's run it twice for one time unit, piping the result of the first one into the second one, and once for two time units. We can then compare the results with our phase space measuring rod.

We may as well take the same case we tested before, starting from the file `test.in`, containing a Plummer model with five particles:

```
|gravity> kali nbody_cst1.rb -t 1 < test.in > test01.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 = 0.0671 , E_pot =  -0.317 , E_tot = -0.25
            E_tot - E_init = -2.74e-08
  (E_tot - E_init) / E_init = 1.1e-07
```

Now I'll run it for another time unit, till time 2:

```
|gravity> kali nbody_cst1.rb -t 2 < test01.out > test12.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 = 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 = 1, after 0 steps :
  E_kin = 0.0671 , E_pot =  -0.317 , E_tot = -0.25
            E_tot - E_init = 0
  (E_tot - E_init) / E_init = -0
at time t = 2, after 1000 steps :
  E_kin = 0.574 , E_pot =  -0.824 , E_tot = -0.25
            E_tot - E_init = -1.05e-09
  (E_tot - E_init) / E_init = 4.18e-09
at time t = 3, after 2000 steps :
  E_kin = 0.172 , E_pot =  -0.422 , E_tot = -0.25
            E_tot - E_init = -1.54e-07
  (E_tot - E_init) / E_init = 6.16e-07
```

Hey, that is strange, it ran all the way to time 3, even though I gave it the option `-t 2`.

5.2 The Danger of Short Options

Alice: That shows the danger of using short options. I bet we defined the argument for this option to be the time *difference*, not the target time of the integration. Let's check:

```
|gravity> kali nbody_cst1.rb --help -t
-t  --duration: Duration of the integration [default: 10]

    This option sets the duration t of the integration, the time period
    after which the integration will halt.  If the initial snapshot is
    marked to be at time t_init, the integration will halt at time
    t_final = t_init + t.
```

Bob: I really like our help facility. Helpful for sure! Okay, I'll try again:

```
|gravity> kali nbody_cst1.rb -t 1 < test01.out > test12.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 = 1, after 0 steps :
  E_kin = 0.0671 , E_pot =  -0.317 , E_tot = -0.25
            E_tot - E_init = 0
  (E_tot - E_init) / E_init = -0
at time t = 2, after 1000 steps :
  E_kin = 0.574 , E_pot =  -0.824 , E_tot = -0.25
            E_tot - E_init = -1.05e-09
  (E_tot - E_init) / E_init = 4.18e-09
```

And now I'll run the whole show right through from the beginning:

```
|gravity> kali nbody_cst1.rb -t 2 < test.in > test02.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 = 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, 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 = 0.0671 , E_pot =  -0.317 , E_tot = -0.25
            E_tot - E_init = -2.74e-08
  (E_tot - E_init) / E_init = 1.1e-07
at time t = 2, after 2000 steps :
  E_kin = 0.574 , E_pot =  -0.824 , E_tot = -0.25
            E_tot - E_init = -2.84e-08
  (E_tot - E_init) / E_init = 1.14e-07
```

Now *if* everything behaves as I expect it will, we should get zero distance in phase space:

```
|gravity> cat test12.out test02.out | kali nbody_diff.rb
==> 6N-dimensional phase space distance between two N-body systems <==
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:   2.7152429767103721e+00
```

5.3 Know Your Output File

Alice: Not quite.

Bob: That's an understatement! The difference is huge. What happened?

Alice: No idea. But as we did before, we may have to inspect the files again. First, let's see what we have here:

```
|gravity> ll test??.out
ll: .
```

Bob: Ah, the file `test02.out` is twice as big as the others. Of course! By default, the integrator gives one full output per time unit. This means that the first file, the one we are comparing with by default, will be the result of an integration to time 1, not 2. That explains!

Alice: If it does, we should at least get a null result in a comparison with the first shorter run:

```
|gravity> cat test01.out test02.out | kali nbody_diff.rb
==> 6N-dimensional phase space distance between two N-body systems <==
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
```

Bob: And yes, that's obviously the correct explanation. Well, let me repeat the longer run, but this time with only one output at the end.

```
|gravity> kali nbody_cst1.rb -t 2 -o 2 < test.in > test02.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 = 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, 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 = 0.0671 , E_pot =  -0.317 , E_tot = -0.25
            E_tot - E_init = -2.74e-08
  (E_tot - E_init) / E_init = 1.1e-07
at time t = 2, after 2000 steps :
  E_kin = 0.574 , E_pot =  -0.824 , E_tot = -0.25
            E_tot - E_init = -2.84e-08
  (E_tot - E_init) / E_init = 1.14e-07
```

Wanna bet we'll get zero now?

Alice: I won't hold my breath. Let's check:

```
|gravity> cat test12.out test02.out | kali nbody_diff.rb
==> 6N-dimensional phase space distance between two N-body systems <==
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
```

Bob: You could have held your breath, and you wouldn't have suffered! Good, so we have reproducibility, at least for the constant time step scheme.

5.4 Synchronization

Alice: Let's do a similar thing for shared time steps, first with our `exact_time` option.

```
|gravity> kali nbody_sh1.rb -t 1 --exact_time < test.in > test01e.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 1263 steps :
  E_kin = 0.0671 , E_pot =  -0.317 , E_tot = -0.25
            E_tot - E_init = 1.18e-10
  (E_tot - E_init) / E_init = -4.72e-10

|gravity> kali nbody_sh1.rb -t 1 --exact_time < test01e.out > test12e.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 = 1, after 0 steps :
  E_kin = 0.0671 , E_pot =  -0.317 , E_tot = -0.25
            E_tot - E_init = 0
  (E_tot - E_init) / E_init = -0
at time t = 2, after 664 steps :
  E_kin = 0.574 , E_pot =  -0.824 , E_tot = -0.25
            E_tot - E_init = -2.38e-10
  (E_tot - E_init) / E_init = 9.51e-10

|gravity> kali nbody_sh1.rb -t 2 -o 2 --exact_time < test.in > test02e.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 = 2.0
Duration of the integration: t = 2.0
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 = 1.00345, after 1263 steps :
```

```
  E_kin = 0.0669 , E_pot =  -0.317 , E_tot = -0.25
            E_tot - E_init = 1.18e-10
  (E_tot - E_init) / E_init = -4.72e-10
at time t = 2, after 1927 steps :
  E_kin = 0.574 , E_pot =  -0.824 , E_tot = -0.25
            E_tot - E_init = -1.2e-10
  (E_tot - E_init) / E_init = 4.81e-10
```

5.5 The Toll it Takes

Bob: According to our predictions, the composition of the first two shorter runs should *not* give exactly the same result as the longer runs. Let's check:

```
|gravity> cat test12e.out test02e.out | kali nbody_diff.rb
==> 6N-dimensional phase space distance between two N-body systems <==
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:   4.2584576794453078e-11
```

Indeed, the difference is slight, but clearly noticeable.

Alice: Let's make sure that it's not a matter of having a different ending time:

```
|gravity> cat test12e.out test02e.out
ACS
  NBody
    Array body
      Body body[0]
        Fixnum body_id
          1
        Float mass
             2.5000000000000000e-01
        Vector old_acc
             1.9141062117061396e+00  -3.2573093565241975e+00   3.8783125536583976e+00
        Vector old_jerk
             3.1858863406978042e+00   3.8803068627154129e+00  -2.1926133679272155e+01
        Vector old_pos
            -3.3967972504594651e-01  -1.5678578297817809e+00   1.5544912589310083e+00
        Vector old_vel
```

```
      -8.8755570774917669e-01   5.3779931590199204e-01  -9.8728817993186183e-02
  Vector pos
      -3.4006047651984028e-01  -1.5676273129928968e+00   1.5544492426649668e+00
  Vector vel
      -8.8673390268023422e-01   5.3640167863474320e-01  -9.7066311997470137e-02
Body body[1]
  Fixnum body_id
    2
  Float mass
       2.5000000000000000e-01
  Vector old_acc
      -9.4445551001747343e-04  -5.6661216435256951e-03   5.7011954299326069e-03
  Vector old_jerk
      -3.0591122301515069e-04  -4.2931434315419530e-04   3.8951449102944094e-04
  Vector old_pos
       8.4153791390669419e-01   5.0556139411766106e+00  -5.0875185270532448e+00
  Vector old_vel
       1.4465748770260967e-01  -1.8276289362348408e-01   2.2693581536010152e-01
  Vector pos
       8.4159999900385285e-01   5.0555355011070553e+00  -5.0874211285150253e+00
  Vector vel
       1.4465708232594582e-01  -1.8276532549154978e-01   2.2693826227773484e-01
Body body[2]
  Fixnum body_id
    3
  Float mass
       2.5000000000000000e-01
  Vector old_acc
       7.8946856317134992e+00   3.8948301282852893e+00  -6.3541962347663006e+00
  Vector old_jerk
       1.6959464415028154e+02   1.0236698875701537e+01  -1.6823830645128800e+02
  Vector old_pos
      -3.1984044622330471e-01  -1.7641882200946466e+00   1.8043322643506285e+00
  Vector old_vel
       1.4083286336532503e+00   4.8897425053028498e-01  -4.1381618803834959e-02
  Vector pos
      -3.1923527983867950e-01  -1.7639779996124831e+00   1.8043139164205535e+00
  Vector vel
       1.4117326443301170e+00   4.9064678843954895e-01  -4.4124373770548259e-02
Body body[3]
  Fixnum body_id
    4
  Float mass
       2.5000000000000000e-01
  Vector old_acc
      -9.8078473879096215e+00  -6.3185465011756592e-01   2.4701824856779706e+00
```

```
        Vector old_jerk
            -1.7278022457975632e+02  -1.4116576424073795e+01   1.9016405061606912e+02
        Vector old_pos
            -1.8201774299499396e-01  -1.7235678920437867e+00   1.7286950023554932e+00
        Vector old_vel
            -6.6543041365856614e-01  -8.4401067406582542e-01  -8.6825379600230040e-02
        Vector pos
            -1.8230424300290635e-01  -1.7239301892458181e+00   1.7286579680129452e+00
        Vector vel
            -6.6965582402749735e-01  -8.4428314283967332e-01  -8.5747577546922166e-02
    Float dt
         4.2918749063036721e-04
    Float e0
        -2.4999999988195465e-01
    Fixnum nsteps
      664
    Float time
         2.0000000000000000e+00
SCA
ACS
  NBody
    Array body
      Body body[0]
        Fixnum body_id
          1
        Float mass
             2.5000000000000000e-01
        Vector old_acc
             1.9152526469459683e+00  -3.2559114990438873e+00   3.8704733534041362e+00
        Vector old_jerk
             3.2184721236941041e+00   3.9286525632628613e+00  -2.1867014071704482e+01
        Vector old_pos
            -3.3999735729238312e-01  -1.5676655002894941e+00   1.5544561612806473e+00
        Vector old_vel
            -8.8687023130320863e-01   5.3663341218017679e-01  -9.7341742316430044e-02
        Vector pos
            -3.4006047651801874e-01  -1.5676273129948204e+00   1.5544492426653671e+00
        Vector vel
            -8.8673390268747820e-01   5.3640167864254595e-01  -9.7066312004947572e-02
      Body body[1]
        Fixnum body_id
          2
        Float mass
             2.5000000000000000e-01
        Vector old_acc
            -9.4456502371123466e-04  -5.6662753217542015e-03   5.7013348732266374e-03
```

```
      Vector old_jerk
          -3.0587778675345218e-04  -4.2919621483461767e-04   3.8947383942998108e-04
      Vector old_pos
           8.4158970285608214e-01   5.0555485096390669e+00  -5.0874372811099340e+00
      Vector old_vel
           1.4465714955729472e-01  -1.8276492218636212e-01   2.2693785647724160e-01
      Vector pos
           8.4159999900385485e-01   5.0555355011070597e+00  -5.0874211285150244e+00
      Vector vel
           1.4465708232594507e-01  -1.8276532549154981e-01   2.2693826227773531e-01
    Body body[2]
      Fixnum body_id
        3
      Float mass
           2.5000000000000000e-01
      Vector old_acc
           7.9558430009012033e+00   3.8984108534192980e+00  -6.4150043838142787e+00
      Vector old_jerk
           1.7206257891329074e+02   9.7630621787578669e+00  -1.7147448536182904e+02
      Vector old_pos
          -3.1933574147265315e-01  -1.7640129121252832e+00   1.8043170407566027e+00
      Vector old_vel
           1.4111659411621365e+00   4.9036928960756015e-01  -4.3667343043685594e-02
      Vector pos
          -3.1923527984207184e-01  -1.7639779996138707e+00   1.8043139164201678e+00
      Vector vel
           1.4117326443096312e+00   4.9064678843334053e-01  -4.4124373754361901e-02
    Body body[3]
      Fixnum body_id
        4
      Float mass
           2.5000000000000000e-01
      Vector old_acc
          -9.8701510828234600e+00  -6.3683307905365694e-01   2.5388296955369158e+00
      Vector old_jerk
          -1.7528074515919809e+02  -1.3691285545805894e+01   1.9334110995969408e+02
      Vector old_pos
          -1.8225660444873118e-01  -1.7238700979694570e+00   1.7286640776558739e+00
      Vector old_vel
          -6.6895285946724281e-01  -8.4423778085905188e-01  -8.5928772154858848e-02
      Vector pos
          -1.8230424300145320e-01  -1.7239301892436258e+00   1.7286579680126053e+00
      Vector vel
          -6.6965582399911816e-01  -8.4428314284201378e-01  -8.5747577556158722e-02
  Float dt
       7.1176227354419197e-05
```

```
     Float e0
         -2.4999999999999994e-01
     Fixnum nsteps
       1927
     Float time
          2.0000000000000000e+00
 SCA
```

Bob: No, they both end nicely at time 2. So the difference is really a difference between the orbits. This is the toll that synchronisation takes!

Chapter 6

Reproducibility Gained

6.1 Asynchronicity

Alice: We have seen that constant time steps allow you to stop and restart, and to get exactly the same results as you would have gotten, had you done one single integration from being to end. But when we tried to do the same thing with adaptive, shared time steps, we found that we could not reproduce the same orbits.

Bob: And we knew why. In fact, we had predicted that we couldn't.

Alice: Yes. Now the question is: can we regain our reproducibility?

Bob: How?

Alice: What did *not* work was to force an output at exactly the right time. That had the side effect of forcing the last step to be smaller than it would have otherwise been.

However, there is an alternative. How about letting the system overshoot, so that it can do an output a time it is happy with. When we then restart the system, the integration should happen just like it would have otherwise.

Bob: I see what you mean. Well, let's test it!

Alice: Here is what we did before, but this time without the `--exact_time` option:

```
|gravity> kali nbody_sh1.rb -t 1 < test.in > test01n.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
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 = 1.00345, after 1263 steps :
  E_kin = 0.0669 , E_pot =  -0.317 , E_tot = -0.25
            E_tot - E_init = 1.18e-10
  (E_tot - E_init) / E_init = -4.72e-10

|gravity> kali nbody_sh1.rb -t 1 < test01n.out > test12n.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
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 = 1.00345, after 0 steps :
  E_kin = 0.0669 , E_pot =  -0.317 , E_tot = -0.25
            E_tot - E_init = 0
  (E_tot - E_init) / E_init = -0
at time t = 2.00373, after 669 steps :
  E_kin = 0.591 , E_pot =  -0.841 , E_tot = -0.25
            E_tot - E_init = -2.45e-10
  (E_tot - E_init) / E_init = 9.81e-10

|gravity> kali nbody_sh1.rb -t 2 -o 2 < test.in > test02n.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 = 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, 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 = 1.00345, after 1263 steps :
  E_kin = 0.0669 , E_pot =  -0.317 , E_tot = -0.25
            E_tot - E_init = 1.18e-10
  (E_tot - E_init) / E_init = -4.72e-10
at time t = 2.00058, after 1927 steps :
  E_kin = 0.577 , E_pot =  -0.827 , E_tot = -0.25
            E_tot - E_init = -1.22e-10
  (E_tot - E_init) / E_init = 4.86e-10
```

Bob: So you think we'll get a smaller distance now?

Alice: As close to zero as we can!

Bob: Well, let's hope for the best:

```
|gravity> cat test12n.out test02n.out | kali nbody_diff.rb
==> 6N-dimensional phase space distance between two N-body systems <==
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:   5.2669426997299477e-02
```

6.2 Overshooting

Alice: Hmmm, that's not good at all. What did we do to deserve that!

Bob: If we look at the diagnostics output, we see that we needed the same number of steps to reach time 1, exactly 1263 in each case. Then in the continued run, we took another 669 steps. This gives us a total number of 1932 steps. Aha! That is *not* the number of steps that the second run took.

Alice: Good catch! The second run stopped too early. And yes, of course, we could have noticed that directly, had we looked at the final time: the second run indeed shows an earlier ending time than the first one.

Bob: And that makes sense, too: we ask each run to go fro one or two time units, or slightly more if they overshoot. Now asking a code to overshoot twice in a row is likely to produce a larger overshoot than doing it only once.

Alice: Again, you must be right. Going from time 0 to 1, with an overshoot, and then adding one more time unit plus a new overshoot is *not* the same as going from 0 to 2 immediately, with whatever overshoot that gives.

Bob: Well, shall we try to let the long run proceed a little further, to match up with the previous one?

Alice: You can certainly try. So this means adding the difference of the two ending times to the desired duration; or in fact slightly less, since we'll produce an overshoot anyway.

Bob: Okay, how about this:

```
|gravity> kali nbody_sh1.rb -t 2.0037 -o 2.0037 < test.in > test02n.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 = 2.0037
Duration of the integration: t = 2.0037
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 = 1.00345, after 1263 steps :
  E_kin = 0.0669 , E_pot =  -0.317 , E_tot = -0.25
            E_tot - E_init = 1.18e-10
  (E_tot - E_init) / E_init = -4.72e-10
at time t = 2.00058, after 1927 steps :
  E_kin = 0.577 , E_pot =  -0.827 , E_tot = -0.25
            E_tot - E_init = -1.22e-10
  (E_tot - E_init) / E_init = 4.86e-10

|gravity> cat test12n.out test02n.out | kali nbody_diff.rb
==> 6N-dimensional phase space distance between two N-body systems <==
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
```

6.3 Living in a Logical World

Alice: The right distance, this time, namely zero. Perfect!

Bob: It literally couldn't have been better.

Alice: Wait, wait: we still have only 1927 steps, and we still stop at too early a time. What is going on? According to the diagnostics, nothing has changed from the previous run.

Bob: Now *that* is strange. Obviously, the phase space distance is zero. So something has changed. Yet I see your point. This is a real paradox.

Alice: What could possibly be . . . .

Bob: . . . Ahaha! Of course! We changed the ending time and output time, by a slight amount, but we did not change the diagnostics output time!

Alice: Of course, of course! Ah, that is a relief. Good! Mystery solved, hopefully. You'd better check, though!

Bob: Okay, here we go again:

```
|gravity> kali nbody_sh1.rb -t 2.0037 -d 2.0037 -o 2.0037 < test.in > test02n.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 = 2.0037
Time interval between snapshot output: dt_out = 2.0037
Duration of the integration: t = 2.0037
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 = 2.00373, after 1932 steps :
  E_kin = 0.591 , E_pot =  -0.841 , E_tot = -0.25
            E_tot - E_init = -1.27e-10
  (E_tot - E_init) / E_init = 5.1e-10

|gravity> cat test12n.out test02n.out | kali nbody_diff.rb
==> 6N-dimensional phase space distance between two N-body systems <==
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: All is well.

Bob: How easy to get confused!

Alice: And how nice to be reassured, afterwards, that the world is still logical, after all.

6.4 An ACS `tail` Version

Bob: You know, there is a better solution for the problem we encountered, when we had to deal with an output file that had more than one snapshot in it.

Alice: You mean in the case in which we ran a code for two time steps, but we had forgotten that we get by default an output at every time step, so that when we looked at the start of the file, we found the output for time 1, rather than for time 2?

Bob: Yes, that was the problem. I'm sure we'll run into this again, often, and what we really need is a tool that picks up the *last* ACS object from a file, not the first one. Normally, if you open a file and start reading, you would read in the first ACS file first, but instead, we need to do something like what in Unix the command `tail` does, showing the end of a file.

Alice: The Unix command `tail` has a default of showing you ten lines, but you can ask it to show more or fewer lines, with the option `-n` as in:

```
|gravity> tail -n 3 test02n.out
    Float time
         2.0037277867566679e+00
SCA
```

which shows the last three lines.

Bob: Okay, let us call the file `acstail.rb`. How about this?

```
#!/usr/local/bin/ruby -w

require "acs"

options_text= <<-END

  Description: Returns the last n chunks of ACS data
```

```
  Long description:
    This program accepts ACS data, in the form of a series of chunks, each
    chunk starting with "ACS" on the first line, and ending with "SCA" on
    the last line.  Like the Unix program ‘tail’, it returns the last n chunks.

    (c) 2004, Piet Hut, Jun Makino; see ACS at www.artcompsi.org

    example:
               kali #{$0} -n 5


  Short name:           -n
  Long name:            --number
  Value type:           int
  Default value:        1
  Description:          Number of last ACS data chunks returned
  Variable name:        n
  Long description:
    This option allows the user to choose the number of ACS chunks that
    will be returned by this program, after a series of ACS chunks have been
    read in.


  END

clop = parse_command_line(options_text)

chunk = []
a = ACS_IO.acs_read
while a
  chunk.push(a)
  chunk.shift if chunk.length > clop.n
  a = ACS_IO.acs_read
end
while chunk[0]
  chunk.shift.acs_write
end
```

6.5 Testing `tail`

Alice: Looks fine to me. How about doing our previous experiment, without adjusting the output frequency, and using `acstail.rb` instead?

Bob: Good idea! Here you go. Let's be systematic. We know that this works:

```
|gravity> kali nbody_cst1.rb -t 2 -o 2 < test.in > test02.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 = 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, 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 = 0.0671 , E_pot =  -0.317 , E_tot = -0.25
            E_tot - E_init = -2.74e-08
  (E_tot - E_init) / E_init = 1.1e-07
at time t = 2, after 2000 steps :
  E_kin = 0.574 , E_pot =  -0.824 , E_tot = -0.25
            E_tot - E_init = -2.84e-08
  (E_tot - E_init) / E_init = 1.14e-07

|gravity> cat test12.out test02.out | kali nbody_diff.rb
==> 6N-dimensional phase space distance between two N-body systems <==
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
```

and we also know that this doesn't work:

```
|gravity> kali nbody_cst1.rb -t 2 < test.in > test02.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 = 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, 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 = 0.0671 , E_pot =  -0.317 , E_tot = -0.25
            E_tot - E_init = -2.74e-08
  (E_tot - E_init) / E_init = 1.1e-07
at time t = 2, after 2000 steps :
  E_kin = 0.574 , E_pot =  -0.824 , E_tot = -0.25
            E_tot - E_init = -2.84e-08
  (E_tot - E_init) / E_init = 1.14e-07

|gravity> cat test12.out test02.out | kali nbody_diff.rb
==> 6N-dimensional phase space distance between two N-body systems <==
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:   2.7152429767103721e+00
```

And if all is well, our new tool should be able to repair the last problem, as follows:

```
|gravity> kali nbody_cst1.rb -t 2 < test.in | kali acstail.rb > test02.out
==> 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
==> 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 = 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, 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 = 0.0671 , E_pot =  -0.317 , E_tot = -0.25
            E_tot - E_init = -2.74e-08
  (E_tot - E_init) / E_init = 1.1e-07
at time t = 2, after 2000 steps :
  E_kin = 0.574 , E_pot =  -0.824 , E_tot = -0.25
            E_tot - E_init = -2.84e-08
  (E_tot - E_init) / E_init = 1.14e-07

|gravity> cat test12.out test02.out | kali nbody_diff.rb
==> 6N-dimensional phase space distance between two N-body systems <==
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: Well done!

Chapter 7

Efficiency

7.1 Pure Thought

Bob: There is one thing we haven't checked yet. The whole motivation for going to variable time steps was to improve accuracy, for the same amount of computer time. Or to put it differently, to save computer time, if the goal is to reach a certain accuracy.

Alice: Good idea. Let's check that.

Bob: It is pretty clear, on general grounds, that an adaptive algorithm must be more accurate, but I like to see by how much.

Alice: I don't think it is always clear, since there are algorithms that do a lot better for constant time steps than for variable time steps. The leapfrog is an example of an algorithm that is almost unreasonably accurate in energy conservation, basically because it is time symmetric. But that symmetry property is lost when you allow the time steps to become variable, unless you make sure that your time step criterion is again time symmetric, something we haven't done here.

Bob: That may be, but in general, adaptive algorithms should do better, as a rule of thumb.

Alice: Well, yes, in the limit of many particles and relatively large time steps you must be right, for sure. And most importantly, an adaptive time step scheme is safer, since particles are guaranteed not to suddenly run into each other. If you don't have softening, and you choose a fixed time step, you can never guarantee that you don't accidentally step on another particle. I mean, you could reach another particle in one time step, which would result in a huge energy error.

Bob: With all this pure thought, let's do a hard-nosed test. I think we've

done something like this already, but let's run it again, from the same initial conditions:

```
|gravity> kali mkplummer.rb -n 4 -s 1 | kali nbody_set_id.rb > test
==> Takes an N-body system, and gives each body a unique ID <==
value of @body_id for 1st body: 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
==> 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
```

7.2 Concrete Results

Alice: And we may as well start with the same type of accuracy requirements as we did before.

Bob: Here is the shared time step version:

```
|gravity> kali nbody_sh1.rb -t 1 -c 0.01 --exact_time -o 2 < test.in
==> 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 = 2.0
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 = 1.00345, after 1263 steps :
  E_kin = 0.0669 , E_pot =  -0.317 , E_tot = -0.25
            E_tot - E_init = 1.18e-10
  (E_tot - E_init) / E_init = -4.72e-10
```

And here is the constant time step version:

```
|gravity> kali nbody_cst1.rb -c 0.001 -t 1 -o 2 < test.in
==> 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 = 2.0
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 = 0.0671 , E_pot =  -0.317 , E_tot = -0.25
            E_tot - E_init = -2.74e-08
  (E_tot - E_init) / E_init = 1.1e-07
```

Alice: Well, the constant time step version seems to do a somewhat better job! I suggest you increase the timestep a bit, to bring out the difference more clearly.

Bob: Okay, I'll make the timestep 0.00125 time units:

```
|gravity> kali nbody_cst1.rb -c 0.00125 -t 1 -o 2 < test.in
==> Constant Time Step Code <==
Integration method: method = hermite
Softening length: eps = 0.0
Time step size: dt = 0.00125
Interval between diagnostics output: dt_dia = 1.0
Time interval between snapshot output: dt_out = 2.0
```

```
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 800 steps :
  E_kin = 0.0671 , E_pot =  -0.317 , E_tot = -0.25
            E_tot - E_init = -8.58e-08
  (E_tot - E_init) / E_init = 3.43e-07
```

Alice: With nearly the same number of time steps, the constant time step is scheme is more accurate by almost a factor two.

Bob: Hmmm. So much for all that hard work, to make the time step adaptive!

7.3 More is Different

Alice: Well, we are dealing only with five particles, and we are running at relatively high accuracy. I sugest we increase the number of particles and decrease the accuracy. At some point the adaptive time step scheme should win out from the constant time step scheme.

Bob: Okay, five times more particles:

```
|gravity> kali mkplummer.rb -n 25 -s 1 | kali nbody_set_id.rb > test2.in
==> Takes an N-body system, and gives each body a unique ID <==
value of @body_id for 1st body: 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
==> Plummer's Model Builder <==
Number of particles: N = 25
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
```

And a ten times longer time step. The shared time step code should be able to handle that gracefully.

```
|gravity> kali nbody_sh1.rb -t 1 -c 0.1 --exact_time -o 2 < test2.in
==> Shared Time Step Code <==
Integration method: method = hermite
Parameter to determine time step size: dt_param = 0.1
Interval between diagnostics output: dt_dia = 1.0
Time interval between snapshot output: dt_out = 2.0
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 = 1.00341, after 404 steps :
  E_kin = 0.294 , E_pot =  -0.544 , E_tot = -0.25
            E_tot - E_init = -1.7e-06
  (E_tot - E_init) / E_init = 6.8e-06
```

Alice: Indeed. I'm not so sure about the constant time step scheme.

Bob: Perhaps less gracefully, we have to see:

```
|gravity> kali nbody_cst1.rb -c 0.01 -t 1 -o 2 < test2.in
==> Constant Time Step Code <==
Integration method: method = hermite
Softening length: eps = 0.0
Time step size: dt = 0.01
Interval between diagnostics output: dt_dia = 1.0
Time interval between snapshot output: dt_out = 2.0
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 100 steps :
  E_kin = 0.455 , E_pot =  -0.517 , E_tot = -0.0619
            E_tot - E_init = 0.188
  (E_tot - E_init) / E_init = -0.753
```

Indeed, quite a bit less. Let's bring the number of time steps in line with that of the shared time step code:

```
|gravity> kali nbody_cst1.rb -c 0.005 -t 1 -o 2 < test2.in
==> Constant Time Step Code <==
Integration method: method = hermite
Softening length: eps = 0.0
Time step size: dt = 0.005
Interval between diagnostics output: dt_dia = 1.0
Time interval between snapshot output: dt_out = 2.0
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 200 steps :
  E_kin = 0.414 , E_pot =  -0.502 , E_tot = -0.0879
            E_tot - E_init = 0.162
  (E_tot - E_init) / E_init = -0.649
```

Good. The shared time step code is a lot better now, by orders of magnitude.

Alice: So all the hard work *did* pay off, after all!

Chapter 8

Code Listing

8.1 The Whole Code

Alice: Now that we are happy with our shared time step code, it would be nice to get a view of the whole layout.

Bob: I'll make a printout of the file `nbody_sh1.rb`. Here it is:

```
#!/usr/local/bin/ruby -w

require "nbody.rb"

class Body

  def calc(body_array, time_step, s)
    ba  = body_array
    dt = time_step
    eval(s)
  end

  def collision_time_scale(body_array)
    time_scale_sq = VERY_LARGE_NUMBER
    body_array.each do |b|
      unless b == self
        r = b.pos - @pos
        v = b.vel - @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 acc(body_array)
  a = @pos*0                            # null vector of the correct length
  body_array.each do |b|
    unless b == self
      r = b.pos - @pos
      r2 = r*r
      r3 = r2*sqrt(r2)
      a += r*(b.mass/r3)
    end
  end
  a
end

def jerk(body_array)
  j = @pos*0                            # null vector of the correct length
  body_array.each do |b|
    unless b == self
      r = b.pos - @pos
      r2 = r*r
      r3 = r2*sqrt(r2)
      v = b.vel - @vel
      j += (v-r*(3*(r*v)/r2))*(b.mass/r3)
    end
  end
  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

class NBody

  def collision_time_scale
    time_scale = VERY_LARGE_NUMBER
    @body.each do |b|
      indiv_time_scale = b.collision_time_scale(@body)
      if time_scale > indiv_time_scale
        time_scale = indiv_time_scale
      end
    end
    time_scale
  end

  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

def calc(s)
  @body.each{|b| b.calc(@body, @dt, s)}
end

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

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

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

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

def yo2
  leapfrog
end

def yo4
  d = [1.351207191959657, -1.702414383919315]
```

```
    old_dt = @dt
    @dt = old_dt * d[0]
    leapfrog
    @dt = old_dt * d[1]
    leapfrog
    @dt = old_dt * d[0]
    leapfrog
    @dt = old_dt
  end

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

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

  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

  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

options_text = <<-END

  Description: Shared Time Step Code
  Long description:
    This program evolves an N-body code with a fourth-order Hermite Scheme,
    or various other schemes such as forward Euler, leapfrog, or Runge-Kutta,
    using variable time steps, shared by all particles, where the size of
    of the time step is determined adaptively.
```

```
  (c) 2005, Piet Hut and Jun Makino; see ACS at www.artcompsi.org

  example:
  kali mkplummer.rb -n 4 -s 1 | kali #{$0} -t 1 > /dev/null


Short name:           -g
Long name:            --integration_method
Value type:           string
Default value:        hermite
Variable name:        method
Description:          Integration method
Long description:
  This option receives a string, containing the name of the integration
  method that will be used.  Example: "-g hermite" .


Short name:           -c
Long name:            --step_size_control
Value type:           float
Default value:        0.01
Variable name:        dt_param
Description:          Parameter to determine time step size
Long description:
  This option sets the step size control parameter dt_param << 1.  Before
  each new time step, we first calculate the time scale t_scale on which
  changes are expected to happen, such as close encounters or significant
  changes in velocity.  The new shared time step is then given as the
  product t_scale * dt_param << t_scale.


Short name:           -d
Long name:            --diagnostics_interval
Value type:           float
Default value:        1
Variable name:        dt_dia
Description:          Interval between diagnostics output
Long description:
  The time interval between successive diagnostics output.
  The diagnostics include the kinetic and potential energy,
  and the absolute and relative drift of total energy, since
  the beginning of the integration.
      These diagnostics appear on the standard error stream.
```

```
Short name:           -o
Long name:            --output_interval
Value type:           float
Default value:        1
Variable name:        dt_out
Description:          Time interval between snapshot output
Long description:
  This option sets the time interval between output of a snapshot
  of the whole N-body system, which which will appear on the
  standard output channel.

  The snapshot contains the mass, position, and velocity values
  for all particles in an N-body system, in ACS format.


Short name:           -t
Long name:            --duration
Value type:           float
Default value:        10
Variable name:        dt_end
Print name:           t
Description:          Duration of the integration
Long description:
  This option sets the duration t of the integration, the time period
  after which the integration will halt.  If the initial snapshot is
  marked to be at time t_init, the integration will halt at time
  t_final = t_init + t.


Long name:            --exact_time
Value type:           bool
Variable name:        exact_time_flag
Description:          Force all outputs to occur at the exact times
Long description:
  If this flag is set to true, all forms of output will happen at the
  exact times specified.  The variable shared time step will be shortened
  whenever the system would overshoot an output time, in order to guarantee
  output to occur at the right time.  Note that in this case, by changing
  the output times, the trajectories of the particles will be changed too,
  as a side effect.


Short name:           -i
Long name:            --init_out
Value type:           bool
Variable name:        init_out_flag
```

```
  Description:          Output the initial snapshot
  Long description:
    If this flag is set to true, the initial snapshot will be output
    on the standard output channel, before integration is started.


  END

clop = parse_command_line(options_text)

nb = ACS_IO.acs_read(NBody)
nb.evolve(clop)
```

Chapter 9

Literature References

[to be provided]