The Art of Computational Science

The Kali Code

vol. 18

N-Body Experiments:

Orchestrating and Analyzing Multiple Runs

Piet Hut and Jun Makino

September 14, 2007

Contents

Preface

0.1 xxx

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

Chapter 1

Ruby Running Ruby

1.1 A Concrete Research Problem

Alice: We've come a long way, starting with constant time steps, then introducing adaptive time steps that were shared by all particles, and finally switching to individual time steps. And we now have a powerful and very modular version `world3.rb` that contains a large choice of different integrators.

Bob: That is all nice and fine, but I'm getting ready to do some real science. We have spent so much time on tool building, I would like to see some actual experiments, leading to some new results.

Alice: Do you mean something like studying core collapse in star clusters?

Bob: At some point, sure, but that would require quite a few other tools that we don't have yet. At the very least, we would have to introduce regularization methods for binary stars. We'll get to that point, before too long, but frankly, right now I don't want to dive into yet another round of tool building. Let's do something with what we have.

Alice: Sounds reasonable. But we do have to work with our current limitations. Which means: small N values, and no simulations which involve long binary star integrations. That doesn't leave us that much choice.

Bob: Putting it that way already suggests an interesting project. Although we can't integrate binaries very accurately, for very long, at least we can run each simulation until the first hard binary forms. We can study how long it takes to form such a binary, on average.

Alice: That's a good idea! We can define the time for first binary formation, as the typical time it takes to form the first binary. Such a time will of course depend on the initial conditions, but we may as well choose a Plummer model, to start from. After all, there is a tradition in star cluster research, to base

theoretical investigations on Plummer's model.

Bob: So we can define $T_{fb}(N)$ as the typical time of first binary formation, as a function of N, starting from a Plummer model. I like that!

Alice: to make it precise, we'd better define what we mean with 'typical': do we mean the average, or the median, or some other measure?

Bob: I would say, let's start with the average, but it's a good idea to record the median as well. In fact, we should record the outcome for each individual experiment, so that we can later try different definitions.

Alice: So be it! How shall we get started?

1.2 Toward Automatizing

Bob: The first step is to automatize the procedure of running a simulation. If we want to establish the shape of the curve $T_{fb}(N)$, we need to do many runs for each N value. We are talking about a type of Monte Carlo experiment. In order to determine $T_{fb}(N_0)$, for a particular value N_0, to within an accuracy of, say, ten percent, we need to carry out a hundred simulations, something I don't plan to do by hand!

Alice: Yes, that's the drawback of a Monte Carlo approach, that the accuracy of an average over a sample increases only as the square root of the number of elements in that sample. But we don't have any choice, if we want to measure typical behaviors in a situation where each run will give a rather different outcome. How do you envision setting up an automatic Ruby running tool?

Bob: The simplest way would be to just write a shell script. That is how I have always orchestrated runs whenever I was involved in a simulation project.

Alice: That is possible, but I must say, I'm not too thrilled with the idea of having to use shell scripts. Some of the commands are rather arcane, and then there are the subtle differences between a C shell and a Borne shell and all those other UNIX shells that are in use. Can't we use Ruby itself to run Ruby programs?

Bob: That should be possible, in principle. Let me have a look at the manual. I remember seeing a way to execute shell commands from within Ruby. Ah, here it is. Well, that's simple! You just enclose the UNIX command within backquotes. Let's try it out. I'll open a file `test0.rb`. What commands shall we start with?

Alice: How about measuring the convergence of an algorithm? We can run an integration with three values for the accuracy parameter, and measure the phase space distance between the first and second, and between the second and third run. That will give us a sense of the rate of convergence. We've done that before, by hand. Let's now do it in a Ruby script.

Bob: Okay, give me a few minutes, I'll try something out.

1.3 A First Test Script

Alice: Did you succeed?

Bob: Yes, here it is. I'm using the shared time step code `nbody_sh1.rb`. This now does the job:

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

require "acs.rb"

options_text= <<-END

  Description: Testing Ruby's way of running Ruby programs
  Long description:
    This program prints out the results of a comparison between three
    different accuracy parameter choices, in otherwise similar integrations.

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

    example:

        kali mkplummer.rb -n 8 | kali world4.rb -t 1000 -x 0.1 |
        kali #{$0} -n 4 -s 42 -g leapfrog -c 0.01


  Short name:           -n
  Long name:            --n_particles
  Value type:           int
  Default value:        4
  Variable name:        n
  Print name:           N
  Description:          Number of particles
  Long description:
    Number of particles for the Plummer model realization.


  Short name:           -s
  Long name:            --seed
  Value type:           int
  Default value:        0
  Description:          pseudorandom number seed given
  Print name:
```

```
  Variable name:        seed
  Long description:
    Seed that will provided to generate the Plummer model realization.
    If no seed is provided, the default value 0 will be used.  This
    implies that two subsequent invocations of this program will generate
    different results, since the Plummer model builder translates a value
    0 in a seed value equal to a new pseudorandom number.


  Short name:           -g
  Long name:            --integration_method
  Value type:           string
  Default value:        hermite
  Variable name:        integration_method
  Description:          Choice of integration method
  Long description:
    This option chooses the integration method.  The user is expected to
    provide a string with the name of the method, for example "leapfrog",
    "hermite".


  Short name:           -c
  Long name:            --step_size_control
  Value type:           float
  Default value:        0.1
  Variable name:        dt_param
  Description:          Determines the time step size
  Long description:
    This option sets the step size control parameter dt_param for the first
    run.  Subsequent runs are performed with a step size control parameter
    that are ten and one hundred times smaller, respectively.


  END

clop = parse_command_line(options_text)

‘kali mkplummer.rb -n #{clop.n} -s #{clop.seed} > tmp.in‘
m = clop.integration_method
t = clop.dt_param
‘kali nbody_sh1.rb --exact_time -g #{m} -t 1 -c #{t} < tmp.in > tmp1.out‘
‘kali nbody_sh1.rb --exact_time -g #{m} -t 1 -c #{0.1*t} < tmp.in > tmp2.out‘
‘kali nbody_sh1.rb --exact_time -g #{m} -t 1 -c #{0.01*t} < tmp.in > tmp3.out‘
print ‘cat tmp1.out tmp2.out | kali nbody_diff.rb‘
print ‘cat tmp2.out tmp3.out | kali nbody_diff.rb‘
```

Alice: Wow, how neat and careful, and with four well described options. I'm impressed!

Bob: Well, now that we have built all this infrastructure, we may as well use it. It will make it far easier to check later on what it was we have been doing today. And besides, writing those options is easy: I just copied them out of the original codes `mkplummer.rb` and `nbody_sh1.rb`, and adjusted the wording slightly, to fit in with the current purpose.

1.4 Analyzing the Script

Alice: Yes, once you have the right tools in place, life *does* get a lot easier. Let me see whether I can parse what you did. The first line after the usual command line processing indeed is written between backquotes, so the content between those backquotes will be sent to the shell. But why the hash marks?

Bob: Those force an evaluation of what is between the parentheses. If you would leave those out, and if you would just write:

```
‘kali mkplummer.rb -n clop.n‘
```

our test code would send this line, literally as it is, to the shell. Believe me, this is what I did first, of course, and here is what happened:

```
|gravity> kali mkplummer.rb -n clop.n
==> Plummer's Model Builder <==
Number of particles: N = 0
pseudorandom number seed given: 0
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: 91945760400241246664994100956106796295
ACS
  NBody
    Array body
    String story 2
                   actual seed used: 91945760400241246664994100956106796295

SCA
```

Alice: Ah, `mkplummer.rb` simply ignored `clop.n` and gave you a Plummer model with zero particles. Not very interesting!

Bob: I don't think it ignored `clop.n`. If it had done so, it would produce the following result:

```
|gravity> kali mkplummer.rb
==> Plummer's Model Builder <==
Number of particles: N = 1
pseudorandom number seed given: 0
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: 175618412688736592425767831433370386426
ACS
  NBody
    Array body
      Body body[0]
        Fixnum body_id
          0
        Float mass
             1.0000000000000000e+00
        Vector pos
             0.0000000000000000e+00   0.0000000000000000e+00   0.0000000000000000e
        Vector vel
             0.0000000000000000e+00   0.0000000000000000e+00   0.0000000000000000e
    String story 2
                  actual seed used: 175618412688736592425767831433370386426

SCA
```

Alice: I see: one particle this time. A bit better, but still not what we wanted.

Bob: It seems that `-n clop.n` was converted to `-n 0`.

Alice: In any case, yes, I now see the need for the evaluations around each option, in the shell command. In the next two lines you introduce abbrevations in order to keep the following command lines to a reasonable length.

Bob: I know you don't like to exceed 80 columns. Oh, those Fortran types!

Alice: Just a matter of keeping lines from folding in my standard editor window. Okay, you then run the integrator for smaller and smaller time step size parameters, decreasing the value in steps of ten. And finally you determine phase space differences between pairs of output snapshots.

Bob: The tricky thing here was to put `print` in front of the last two lines. At first, I left that out, and I did not get any output at all! It took me a while to realize that giving a shell command in itself is not enough. It would be carried out, of course, and the result would come back from the shell to the

program `test0.rb`. But then the result would only be known internally within `test0.rb`. In order for us to see the result, we have to specifically print it out.

Alice: Tricky! But of course, once you explain it, it makes sense. It is all a matter of meta levels. There is the level of running the Ruby programs `nbody_sh1.rb` and `nbody_diff.rb` and so on, and there is the meta level of running `test0.rb` that in turn is running those other Ruby programs. And in order to get the results from the lower-level programs out, they have to be handed up through the higher-level program.

Bob: Exactly. And before long, we will probably running programs that run programs that run programs, adding more meta levels.

1.5 Running the Script

Alice: Let me try out your nifty meta program. How about starting with a leapfrog, with 5 particles, say, and a specific seed.

```
|gravity> kali test0.rb -n 5 -s 42 -g leapfrog
==> Testing Ruby's way of running Ruby programs <==
Number of particles: N = 5
pseudorandom number seed given: 42
Choice of integration method: integration_method = leapfrog
Determines the time step size: dt_param = 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
==> Plummer's Model Builder <==
Number of particles: N = 5
pseudorandom number seed given: 42
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: 42
==> Shared Time Step Code <==
Integration method: method = leapfrog
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 = 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 102 steps :
  E_kin = 0.16 , E_pot =  -0.41 , E_tot = -0.25
            E_tot - E_init = 0.000177
  (E_tot - E_init) / E_init = -0.000707
==> Shared Time Step Code <==
Integration method: method = leapfrog
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 1013 steps :
  E_kin = 0.161 , E_pot =  -0.411 , E_tot = -0.25
            E_tot - E_init = -9.33e-07
  (E_tot - E_init) / E_init = 3.73e-06
==> Shared Time Step Code <==
Integration method: method = leapfrog
Parameter to determine time step size: dt_param = 0.001
Interval between diagnostics output: dt_dia = 1.0
Time interval between snapshot output: dt_out = 1.0
Duration of the integration: t = 1.0
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 10123 steps :
  E_kin = 0.161 , E_pot =  -0.411 , E_tot = -0.25
```

```
            E_tot - E_init = -1.21e-08
  (E_tot - E_init) / E_init = 4.82e-08
==> 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-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 5-body systems:   6.3502943417827655e-03
6N-dim. phase space dist. for two 5-body systems:   2.4511908242770752e-05
```

Bob: As it should be: the leapfrog is clearly a second order integration scheme.

1.6 Too Much of a Good Thing

Alice: I must admit, we are now getting too much diagnostics for my taste.

Bob: I thought you liked diagnostics.

Alice: For a single run, yes, but enough is enough! For now, let me just look at the last bit of it all. How about testing forward Euler?

```
|gravity> kali test0.rb -n 5 -s 42 -g forward |& tail
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
==> 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 5-body systems:   1.0244617023395275e+00
6N-dim. phase space dist. for two 5-body systems:   3.5468255580390051e-01
```

Bob: Close, but not exactly first order. Could it be that the default starting step size was a bit too large for forward Euler? That's the drawback of leaving out all that diagnostics information.

Alice: Let's filter the information in a different way. How about this:

```
|gravity> kali test0.rb -n 5 -s 42 -g forward |& grep "/ E_init"
  (E_tot - E_init) / E_init = -0
  (E_tot - E_init) / E_init = -0.521
  (E_tot - E_init) / E_init = -0
  (E_tot - E_init) / E_init = -0.0831
  (E_tot - E_init) / E_init = -0
  (E_tot - E_init) / E_init = -0.0089
```

Bob: I was right: the first relative energy error is terribly large.

Alice: But then things converge quickly. Let me rerun the whole thing with a smaller starting value for the time step criterion:

```
|gravity> kali test0.rb -n 5 -s 42 -g forward -c 0.01 |& tail
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
==> 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 5-body systems:   3.5468255580390051e-01
6N-dim. phase space dist. for two 5-body systems:   5.0125664307044661e-02
```

Bob: That's more like it! Nice first-order behavior. Let's try a fourth-order method. How about Yoshida's symplectic algorithm?

Alice: Here you go:

```
|gravity> kali test0.rb -n 5 -s 42 -g yo4 |& tail
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
==> 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 5-body systems:   6.6026147159027994e-04
6N-dim. phase space dist. for two 5-body systems:   6.6219308240790365e-09
```

Bob: Approximately fourth order, indeed.

Chapter 2

A Story Mechanism

2.1 Accessing Diagnostics

Alice: We've made a big step forward, by being able to run other Ruby programs from within a Ruby program. But the example we started with leaves a lot to be desired. Right now, there is far too much information coming out. Using a `tail` or `grep` operation is not a very elegant way to do data reduction. We should find a way to suppress the information that is not needed.

And what is more, it would be nice to present the information that is needed in a more compact way. For example, in our example of algorithm testing, it would be nice to get one number out: the approximate order of the algorithm.

Bob: All of that can be taken care of, but let us do that with a more interesting example script. I'd like to see some real physics coming out! We talked about determining the time of first binary formation. Let's implement that. It should be easy to extend the diagnostics of our integrator, to let it report not only energy change, but also changes in binary configurations.

Alice: One problem I see is this. It is all nice and well to print some information in human-readable form on the standard error stream, as we have done so far, but how are we going to parse that output?

Bob: As we have seen, Ruby is good at parsing strings. In fact, it is a lot better in doing this than `grep` and `tail` and the like are. I suggest we just use the human-readable output, and interpret that with some clever Ruby pattern recognition moves.

Alice: I don't see how that would work, in general. As we have seen, when we run a Ruby script that runs Ruby programs, the diagnostics information still appears on the standard error stream, which is not directly available for analysis within the script.

Bob: Well, we could capture it, by adding " `>& tmp` " at the end of each shell command. We could then open the file `tmp` after each command and inspect the messages left therein. But I admit, that may not be the best solution.

Alice: That is putting it mildly; it would be a terrible solution! It would neither be elegant nor flexible. We have to find a better way.

Bob: At this point, I don't care much about elegance. I want to see some results. But I agree that using a temporary file is not a very good solution. Another drawback is that the information gets overwritten each time. For some applicaitons it would be nice to leave a trail of information that could be inspected if the need would arise, either for debugging or for further statistical analysis.

I have encountered this problem before. After spending a lot of time writing a complex program to do some hairy simulation, I find myself quickly doing a lot of different runs. And for each run, there is a data file containing the particle positions and velocities and so on, but there is also a file containing the runtime diagnostics. And after a few weeks of production runs, it is not alway easy to keep the particle data and the diagnostics data together. I've found myself updating the name of the particle output file but not the name of the diagnostics output file, and so on.

2.2 Capturing Diagnostics

Alice: Keeping related data in two different places is in itself a bad idea.

Bob: I agree, in principle. In practice, though, sticking them together is not so easy. We may be flexible and modular and all that, as you keep stressing, but look, our particle output always takes the form of an ACS data structure, appearing on the standard output stream, while our diagnostics output is just free-form, as a string of English sentences, appearing on the standard error stream. I can't see hot to glue those together.

Alice: Gluing is not the right metaphor; how about encapsulating?

Bob: Sounds nice, but how?

Alice: Of the two forms of output, the particle data are currently most structured, based on our ACS data format, as you just mentioned. How about encapsulating the diagnostics information, as a form of short narative, and adding that to the particle data? We can just call it a story. Each snapshot could get an extra instance variable `@story` in the form of a single long string that contains all the sentences that are reported on the standard error stream.

Bob: Hmmm, that's an idea. So this would imply an extension of the snapshot data format. Since `WorldSnapshot` is a subclass of `Nbody`, perhaps it would be better to introduce `@story` as an instance variable of `NBody` ?

Alice: But why stop there? I can imagine that we may want to report specific information about specific particles.

Bob: I see, you want to allow a story for `WorldPoint` as well, or for `Body`. That makes sense: a particle may want to report that it had an unusually strong encounter, or even a physical collision, when we implement that possibility. How about other classes, such as `WorldLine` ?

Alice: I suggest that we give every class the option to include a story, as a way to report diagnostics. In that way, each class can keep its own diary or chronicles.

Bob: But you still want to get the information out on the standard error stream, don't you?

Alice: That depends. The usual information about overall error behavior we probably want to keep on the error channel. However, even that depends on the application. When we start running 100 runs to determine first binary formation times, we may not want to see any diagnostics from any of the runs, unless we have an indication that something went wrong somewhere.

Here is a bold idea, if I may say so myself. Let us simply give the story option to *each and every* object.

Bob: Hmmm, that may be overkill, but then again, it would be nice to implement it once and for all, and then to forget about it. From then on it would be ready to use in any situation. I like the idea. Shall we implement it?

Here is the file `acsio.rb`. And here is `module ACS_IO` that is included in every object. Well, let me add `@story` as a recognized instance variable for the top class `Object` by adding the information to `module ACS_IO`. This won't take too long.

2.3 xxx

Something about getting precision and offset automatically set

Something about keeping clop and acsio independent, so that you can run one without having to know about the other

using proc and lambda

adding `verbosity` and `acs_verbosity`.

Chapter 3

Finding Binaries

3.1 A `Binary` Class

Alice: A job well done! Now every ruby program that starts with `require "acs"` will automatically receive options for specifying precision, indentation, and verbosity levels for reporting diagnostics on the standard error channel as well as in the stories that appear on the standard output channel.

Bob: I'm glad we have all that in place. Now, *finally* let's get back to our task: letting some real physics come out! We've been talking about determining the time of first binary formation.

In order to do so, we have to recognize binaries when they are formed. Well, I added a quick-and-dirty binary finding part to the diagnostics of our individual time step integrator. I started with `world3.rb`, but I'm calling it `world4.rb` now, with that addition.

At the top of `world4.rb`, I'm including a file, `binary.rb`, that contains a new class `Binary`. Here it is:

```
require "nbody.rb"

class Binary

  def initialize(body1, body2)
    @m1 = body1.mass
    @m2 = body2.mass
    @mass = @m1 + @m2
    @reduced_mass = ( @m1 * @m2 ) / ( @mass )
    @pos = (@m1*body1.pos + @m2*body2.pos)/@mass
    @vel = (@m1*body1.vel + @m2*body2.vel)/@mass
```

```
    @rel_pos = body2.pos - body1.pos
    @rel_vel = body2.vel - body1.vel
  end

  def rel_kinetic_energy
    0.5 * @reduced_mass * @rel_vel * @rel_vel
  end

  def rel_potential_energy
    -( @m1 * @m2 / sqrt( @rel_pos * @rel_pos ) )
  end

  def rel_energy
    rel_kinetic_energy + rel_potential_energy
  end

  def angular_momentum_squared
    r_cross_v = @rel_pos.cross(@rel_vel)
    @reduced_mass**2 * r_cross_v * r_cross_v
  end

  def semi_major_axis
    -( @m1 * @m2 ) / ( 2 * rel_energy )
  end

  def eccentricity
    e_sq = 1 - angular_momentum_squared /
                ( @reduced_mass * @m1 * @m2 * semi_major_axis )
    e_sq = 0.0 if e_sq < 0.0  # to avoid round-off to slightly negative numbers
    sqrt(e_sq)
  end

  def period
    2*PI/sqrt( @mass / semi_major_axis**3 )
  end
end
```

3.2 Verbosity

Alice: I see. You create a new instance of the `Binary` class by giving it two bodies. The initializer then immediately determines the relative position and velocity as well as the total mass and the reduced mass. From that point on, you can ask the binary what its eccentricity or semi-major axis is, or its period, or

some other piece of information. In each case, the answer is provided through a call to a method within the `Binary` class, where the method computes the answer on the fly.

Bob: Indeed. I could have done a more complete job, by adding information about all six orbital elements, preferably in various coordinate systems. For now, though, the main information we are interested in is the hardness of the binary, its internal energy.

Alice: Which you call `rel_energy` to indicate that it is the energy in the relative motion of the two stars, independent of the energy in the center-of-mass motion of the binary. And you provide the semi-major axis because that is directly related to the internal energy.

Bob: Yes, the semi-major axis a is inversely proportional to the binary binding energy `rel_energy`, for given values of the two masses. And once I had computed a I thought I might as well also compute e, the eccentricity of the orbit, which has a simple relation with the relative angular momentum, as you can see in the `binary.rb` listing.

Alice: Now how did you use this class to report binary diagnostics within the integrator?

Bob: In `world4.rb`o, in the class `WorldEra`, I have added the following method:

```
def binary_diagnostics(t)
  v = 1
  acs_log(v, take_snapshot(t).binary_diagnostics)
end
```

Alice: So this method takes a snapshot at the requested time, asks the snapshot to return the binary diagnostics, and then reports that information with our new `acs_log` function.

Bob: I could of course have given the value 1 directly to the first argument of `acs_log`, but I prefered to keep it a free variable, `v`, which is easier to notice and to chance later, if so desired. For the time being, it will have the value 1, which means a verbosity level of 1.

Alice: Let me see whether I got all this correctly now. If the user invokes the integrator with

```
kali world4.rb --verbosity 0 --acs_verbosity 2
```

no binary diagnostics will appear on the standard error stream, but it will be written in the story of `WorldEra`. Similarly, running the code as

```
kali world4.rb --verbosity 1 --acs_verbosity 0
```

will show binary diagnostics will on the standard error stream, but will not write anything in the stories that appear on the standard output stream. In general,

```
kali world4.rb --verbosity v1 --acs_verbosity v2
```

will only show diagnostics information on the screen when $v1 \geq v$ and will only add the information to the `WorldEra` story when $v2 \geq v$, where v is the first argument of `acs_log`.

Bob: That is indeed correct. And I must say, I'm really glad to have such a versatile and general tool for direction information where it is needed!

3.3 The `WorldSnapshot` Level

Alice: Let me look how you have implemented the real work, in the `binary_diagnostics` method within the `WorldSnapshot` class:

```
def binary_diagnostics
  v = 1
  c = Clop.option
  prec = c.binary_diag_precision
  s = ""
  @body.each do |bi|
    @body.each do |bj|
      if bj.body_id > bi.body_id
        b = Binary.new(bi, bj)
        if b.rel_energy < 0 and b.semi_major_axis <= c.max_semi_major_axis
          s += "  [#{bi.body_id}, #{bj.body_id}] :  "
          s += sprintf("a = %.#{prec}e ; ", b.semi_major_axis)
          s += sprintf("e = %.#{prec}e ; ", b.eccentricity)
          s += sprintf("T = %.#{prec}e\n", b.period)
        end
      end
    end
  end
  s
end
```

As before, you introduce the variable v right at the start and set it equal to 1, which is the normal verbosity level. And then in the third line you introduce a measure of precision, `prec`.

Bob: Yes. I could have used the normal precision specified in our standard ACS precision, which by default is 16 digits long for floating point variables.

However, it would be destracting to list the semimajor axis of a binary in full double precision. Therefore I introduced a special command line option to describe the binary diagnostics precision. Here, you can find it among the ever growing list of options for `world4.rb` :

```
|gravity> kali world4.rb -h
Individual Time Step, Individual Integration Scheme Code
-g  --integration_method: Choice of integration method [default: hermite]
-c  --step_size_control: Determines the time step size [default: 0.01]
-f  --init_timescale_factor: Initial timescale factor [default: 0.01]
-e  --era_length: Duration of an era [default: 0.0078125]
-m  --max_timestep_param: Maximum time step (units dt_era) [default: 1]
-d  --diagnostics_interval: Diagnostics output interval [default: 1]
-o  --output_interval: Snapshot output interval [default: 1]
-y  --pruned_dump: Prune Factor [default: 0]
-t  --time_period: Duration of the integration [default: 10]
-u  --cpu_time_max: Max cputime diagnost. interval [default: 60]
-i  --init_out: Output the initial snapshot
-r  --world_output: World output format, instead of snapshot
-a  --shared_timesteps: All particles share the same time step
-x  --max_semi_major_axis: Maximum semi-major axis [default: 1.0e+30]
--binary_diag_precision: Binary Diagnostics Precision [default: 4]
--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)
```

Alice: That makes sense. Reading further in `binary_diagnostics`, I see that you enter a double loop over all the bodies in a snapshot, and for each possible pair you check to see whether that pair of bodies is currently forming a binary.

Bob: This is of course rather wastful of computer time, and not very efficient. If we would be dealing with tens of thousands of particles, we may want to think of a more clever scheme. But for now, this seems like the simplest way of doing things.

Alice: I agree. Premature optimization is the root of all evil, as we already said! So for each pair you first check whether it is bound. Only if the relative energy is negative do you continue and check the semi-major axis. And only if that value is smaller than the maximum value allowed, do you print diagnostics information.

Bob: Well, `sprint` diagnostics information, you mean; the information is printed on a string, and the string is passed back to the calling function.

3.4 The `World` Level

Alice: And that calling function, also named `binary_diagnostics` within the `WorldEra` class, can use that string both for printing on the standard error stream and for adding it to the total ACS wrapped output on the standard output stream. Got ya! And in the list of options above I already saw how the user can set the maximum semi-major axis value.

Great! It all seems to be quite transparent. Last question: how and where does the `WorldEra#binary_diagnostics` method get invoked? Most likely on the `World` level, and close to the other diagnostics. But I don't see the word binary in the listing of clas `World`.

Bob: Almost correct. You are right in that the calls indeed follow the calls to the `WorldEra#write_diagnostics` methods, literally so, but they occur in the module `Output`. But then the module `Output` gets mixed into the class `World`, so a language lawyer might argue that the calls occur at the `World` level. Hera are the two places where it happens, first for normal diagnostics output:

```
def diagnostics(t_target, dt_dia)
  dia_output = false
  while @t_dia <= t_target
    @era.write_diagnostics(@t_dia, @initial_energy)
    @era.binary_diagnostics(@t_dia)
    @t_dia += dt_dia
    dia_output = true
  end
  dia_output
end
```

and then for unscheduled diagnostics output, when a CPU limit is reached before a whole era is finished:

```
def unscheduled_diagnostics(dt_dia)
  t = @era.wordline_with_minimum_interpolation.worldpoint.last.time
  unless diagnostics(t, dt_dia)
    @era.write_diagnostics(t, @initial_energy, true)
    @era.binary_diagnostics(t)
  end
end
```

3.5 Binary Diagnostics in Action

Alice: No need for lawyers here; I think I was close enough. Now I'd like to see your binary diagnostics in actions. How shall I begin?

Bob: Easy enough, why not just do what the programs suggest you do, when you use the `---help` option.

Alice: I keep forgetting that we have such a nice way to do the hand-holding, for users and developers alike! Here we are:

```
|gravity> kali world4.rb ---help

    This program evolves an N-body code with a fourth-order Hermite Scheme,
    using individual time steps.  Note that the program can be forced to let
    all particles share the same (variable) time step with the option -a.

    This is a test version, for the ACS data format

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

    example:
    kali mkplummer.rb -n 4 -s 1 | kali world4.rb -t 1
```

So you suggest I literally do that? Let me try:

```
|gravity> kali mkplummer.rb -n 4 -s 1 | kali world4.rb -t 1
==> Plummer's Model Builder <==
Number of particles: N = 4
pseudorandom number seed given: 1
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
==> Individual Time Step, Individual Integration Scheme Code <==
Choice of integration method: integration_method = hermite
Determines the time step size: dt_param = 0.01
Initial timescale factor: init_timescale_factor = 0.01
Duration of an era: dt_era = 0.0078125
Maximum time step (units dt_era): dt_max_param = 1.0
Diagnostics output interval: dt_dia = 1.0
Snapshot output interval: dt_out = 1.0
Prune Factor: prune_factor = 0
```

```
Duration of the integration: t = 1.0
Max cputime diagnost. interval: cpu_time_max = 60
Maximum semi-major axis: max_semi_major_axis = 1.0e+30
Binary Diagnostics Precision: binary_diag_precision = 4
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
t = 0 (after 0, 0, 0 steps <,=,> t)
          E_kin = 0.25 ,     E_pot =  -0.5 ,      E_tot = -0.25
          E_tot - E_init = 0
          (E_tot - E_init) / E_init = -0
  [0, 1] :  a = 2.1938e+00 ; e = 6.9252e-01 ; T = 2.8872e+01
  [1, 2] :  a = 1.8007e-01 ; e = 8.8222e-01 ; T = 6.7900e-01
t = 1 (after 4329, 0, 6 steps <,=,> t)
          E_kin = 0.313 ,     E_pot =  -0.563 ,      E_tot = -0.25
          E_tot - E_init = 1.64e-10
          (E_tot - E_init) / E_init = -6.58e-10
  [0, 1] :  a = 4.9021e-01 ; e = 8.4660e-01 ; T = 3.0498e+00
  [1, 2] :  a = 1.8175e-01 ; e = 9.4611e-01 ; T = 6.8852e-01
ACS
  NBody
    Array body
      Body body[0]
        Vector acc
             4.3680007628230150e-01  -7.7920066607005689e-01  -3.4447512499978329e
        Array acsmixins
          Module acsmixins[0]
            Integrator_hermite
        Fixnum body_id
          0
        Float dt_param
             1.0000000000000000e-02
        Vector jerk
             3.2148293860481963e-01   1.5455084837998969e+00   5.0309934483466201e
        Float mass
             2.5000000000000000e-01
        Float maxstep
             5.3298829935937153e-03
        Float minstep
             2.8993571645200446e-05
        Float next_time
             1.0026061296123183e+00
        Fixnum nsteps
          332
        Vector pos
```

```
      7.9772604070178846e-03   8.2919239954665480e-01   8.6949862920921966e-02
  Float time
      1.0000000000000000e+00
  Vector vel
     -6.0338461103000307e-01  -2.5365567416813917e-01  -4.2283983072465980e-01
Body body[1]
  Vector acc
      2.9407213696345624e+00   4.3737103517008107e+00   2.8696484413452512e+00
  Array acsmixins
    Module acsmixins[0]
      Integrator_hermite
  Fixnum body_id
    1
  Float dt_param
      1.0000000000000000e-02
  Vector jerk
     -3.0679123445456916e+01  -6.0941785672989397e+01  -4.1544144235617914e+01
  Float mass
      2.5000000000000000e-01
  Float maxstep
      2.8967962731903940e-03
  Float minstep
      1.0126966959234096e-05
  Float next_time
      1.0007045202601113e+00
  Fixnum nsteps
    1933
  Vector pos
      3.4274367211420625e-01   1.1210653405826151e-01   6.9410494803967479e-03
  Float time
      1.0000000000000000e+00
  Vector vel
     -1.7471717282768337e-01  -4.6103585065601543e-01  -3.2046114392320907e-01
Body body[2]
  Vector acc
     -3.4927708485553790e+00  -3.7533254304484691e+00  -2.8624660563443318e+00
  Array acsmixins
    Module acsmixins[0]
      Integrator_hermite
  Fixnum body_id
    2
  Float dt_param
      1.0000000000000000e-02
  Vector jerk
      3.0274773693388109e+01   5.9364365363757656e+01   4.1104361307013662e+01
  Float mass
```

```
          2.5000000000000000e-01
      Float maxstep
          2.8967962777788347e-03
      Float minstep
          1.0126966976886642e-05
      Float next_time
          1.0007045842393165e+00
      Fixnum nsteps
        1933
      Vector pos
          4.5401351630391895e-01   2.5523386444086688e-01   1.0661490776678197e
      Float time
          1.0000000000000000e+00
      Vector vel
          9.5626266521217429e-01   2.6335235690235648e-01   2.0752875999010723e
    Body body[3]
      Vector acc
          1.1435586922134454e-01   1.5721237097257557e-01   2.7242540252499236e
      Array acsmixins
        Module acsmixins[0]
          Integrator_hermite
      Fixnum body_id
        3
      Float dt_param
          1.0000000000000000e-02
      Vector jerk
          8.8699613819200801e-02   3.4954588596844444e-02  -6.0072687819262158e
      Float mass
          2.5000000000000000e-01
      Float maxstep
          7.8125000000000278e-03
      Float minstep
          1.6386823913682258e-04
      Float next_time
          1.0002615373733836e+00
      Fixnum nsteps
        135
      Vector pos
         -8.0473444786322912e-01  -1.1965327941527619e+00  -2.0050581825457850e
      Float time
          1.0000000000000000e+00
      Vector vel
         -1.7816088524364576e-01   4.5133917639638632e-01   5.3577221678447662e
String story 2
  t = 0 (after 0, 0, 0 steps <,=,> t)
            E_kin = 0.25 ,     E_pot =  -0.5 ,      E_tot = -0.25
```

```
                E_tot - E_init = 0
                (E_tot - E_init) / E_init = -0
        [0, 1] :  a = 2.1938e+00 ; e = 6.9252e-01 ; T = 2.8872e+01
        [1, 2] :  a = 1.8007e-01 ; e = 8.8222e-01 ; T = 6.7900e-01
      t = 1 (after 4329, 0, 6 steps <,=,> t)
                E_kin = 0.313 ,     E_pot =  -0.563 ,      E_tot = -0.25
                E_tot - E_init = 1.64e-10
                (E_tot - E_init) / E_init = -6.58e-10
        [0, 1] :  a = 4.9021e-01 ; e = 8.4660e-01 ; T = 3.0498e+00
        [1, 2] :  a = 1.8175e-01 ; e = 9.4611e-01 ; T = 6.8852e-01

    Float time
         1.0000000000000000e+00
SCA
```

3.6 Taming the Flow of Output

Bob: I bet that's more than you wanted!

Alice: Yeah, I have to learn to tame our programs. But it is nice to see the same diagnostics appearing on the screen and within the story that now appears within the ACS wrapped particle output.

Let me exercise your new options. I will ask only for binaries that have a semimajor axis smaller than 0.5, and I want to know the results only to the first two significant digits. And I certainly don't want all that output. I could redirect it to `/dev/null`, but let me instead set the time of next output to, say, 1000, to make sure that we don't see it within our run. Here goes:

```
|gravity> kali mkplummer.rb -n 4 -s 1 > tmp.in
==> Plummer's Model Builder <==
Number of particles: N = 4
pseudorandom number seed given: 1
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
             actual seed used: 1
|gravity> kali world4.rb -t 1 -o 1000 -x 0.5 --binary_diag_precision 2 < tmp.in
==> Individual Time Step, Individual Integration Scheme Code <==
Choice of integration method: integration_method = hermite
Determines the time step size: dt_param = 0.01
Initial timescale factor: init_timescale_factor = 0.01
Duration of an era: dt_era = 0.0078125
```

```
Maximum time step (units dt_era): dt_max_param = 1.0
Diagnostics output interval: dt_dia = 1.0
Snapshot output interval: dt_out = 1000.0
Prune Factor: prune_factor = 0
Duration of the integration: t = 1.0
Max cputime diagnost. interval: cpu_time_max = 60
Maximum semi-major axis: max_semi_major_axis = 0.5
Binary Diagnostics Precision: binary_diag_precision = 2
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
t = 0 (after 0, 0, 0 steps <,=,> t)
          E_kin = 0.25 ,     E_pot =  -0.5 ,      E_tot = -0.25
          E_tot - E_init = 0
          (E_tot - E_init) / E_init = -0
  [1, 2] :  a = 1.80e-01 ; e = 8.82e-01 ; T = 6.79e-01
t = 1 (after 4329, 0, 6 steps <,=,> t)
          E_kin = 0.313 ,     E_pot =  -0.563 ,      E_tot = -0.25
          E_tot - E_init = 1.64e-10
          (E_tot - E_init) / E_init = -6.58e-10
  [0, 1] :  a = 4.90e-01 ; e = 8.47e-01 ; T = 3.05e+00
  [1, 2] :  a = 1.82e-01 ; e = 9.46e-01 ; T = 6.89e-01
```

Bob: Just what you ordered!

Chapter 4

Measuring Binary Formation Times

4.1 Getting Blunt

Bob: Enough with all these preparations! My fingers are itching to do some lab experiments.

Alice: Fair enough. We have already decided to find the typical time of first binary formation, starting from a Plummer model. We have learned how to let Ruby run a Ruby program, and we have taught our `world4.rb` work horse how to identify binaries. Time to put it all together.

Bob: You bet. And this time I will make sure that we don't get drawn into niceties and generalities. Let's be blunt and get results first.

Alice: No more mister Nice Guy, you mean?

Bob: I'm happy to be nice again in a little while, but for now, let's get moving. First question: how to do a single run, and determine the time of first binary formation.

Alice: I suppose we can let `world4.rb` run for a while, then analyse the output, restart `world4.rb` again if no binary of the right hardness has been formed, and so on.

Bob: Too nice. Too complicated. Later on, fine, but let's make a shortcut. How about letting `world4.rb` run and run until the first binary appears, and then just kill the program?

Alice: You are getting blunt, aren't you!

Bob: I sure am. Let me try something. How about this, as a lovely short test program? I'll call it `test1.rb`:

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

require "acs.rb"

options_text= <<-END

  Description: Find the first reported binary, print its story and exit
  Long description:
    This program accepts a stream of Nbody snapshots, and checks the story of
    each one until it finds the first reported binary.  It then prints that
    story on the standard output, and exits.

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

    example:

        kali mkplummer.rb -n 4 -s 1 | kali world4.rb -t 1000 -x 0.25 | kali #{$0}


  END

clop = parse_command_line(options_text)

while nb = ACS_IO.acs_read
  raise "class #{nb.class} is not NBody" unless nb.class == NBody
  nb.story.each_line do |line|
    if line =~ / :  a = /
      print nb.story
      exit
    end
  end
end
```

Alice: A lovely program to kill our integrator? Funny choice of phrase.

Bob: But that's what it does. It reads in whatever our integrator spits out, and as soon as it finds a diagnostics line reporting a binary being formed, it exits.

Alice: And by doing so, it breaks the pipe between the integrator and the test program, rudely stopping the integrator in its tracks. This is about the most unelegant way of programming I've every seen!

Bob: So be it. Let me show you in more detail, perhaps you will then appreciate the simplicity, if not the elegance, behind the madness. See, the particle output from `world4.rb` gets piped into `test1.rb` which then reachs each story line by line. If the story contains a line that contains a semicolon followed by two

spaces and then `a =`, it assumes that it has found a report of a binary that has been formed.

The first time it finds such a binary, it halts execution and forces the whole pipeline to end its operation. So it is up to the user to give the correct value to the option in `world4.rb` that governs the maximum semi-major axis that triggers binary reporting.

In other words, if you want to detect when the first binary is formed that has a semi-major axis that is smaller than `0.25`, you should invoke the integrator as `world4.rb -x 0.25`. Wider binaries will then go unreported, and only binaries of interest will trigger the collapse of the pipeline.

4.2 Lots of Output

Alice: Soon I'll insist to clean up this criminal way of killing integrators. But for now, let's see whether it all works.

Bob: It does, and I'll show you. See, I even gave it a suggestion how to get started:

```
|gravity> kali world4.rb ---help

    This program evolves an N-body code with a fourth-order Hermite Scheme,
    using individual time steps.  Note that the program can be forced to let
    all particles share the same (variable) time step with the option -a.

    This is a test version, for the ACS data format

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

    example:
    kali mkplummer.rb -n 4 -s 1 | kali world4.rb -t 1
```

And here we go:

```
|gravity> kali mkplummer.rb -n 4 -s 1 | kali world4.rb -t 1000 -x 0.25 | kali test1.rb
==> Find the first reported binary, print its story and exit <==
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
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
==> Individual Time Step, Individual Integration Scheme Code <==
Choice of integration method: integration_method = hermite
Determines the time step size: dt_param = 0.01
Initial timescale factor: init_timescale_factor = 0.01
Duration of an era: dt_era = 0.0078125
Maximum time step (units dt_era): dt_max_param = 1.0
Diagnostics output interval: dt_dia = 1.0
Snapshot output interval: dt_out = 1.0
Prune Factor: prune_factor = 0
Duration of the integration: t = 1000.0
Max cputime diagnost. interval: cpu_time_max = 60
Maximum semi-major axis: max_semi_major_axis = 0.25
Binary Diagnostics Precision: binary_diag_precision = 4
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
t = 0 (after 0, 0, 0 steps <,=,> t)
          E_kin = 0.25 ,     E_pot =  -0.5 ,      E_tot = -0.25
          E_tot - E_init = 0
          (E_tot - E_init) / E_init = -0
  [1, 2] :  a = 1.8007e-01 ; e = 8.8222e-01 ; T = 6.7900e-01
t = 1 (after 4329, 0, 6 steps <,=,> t)
          E_kin = 0.313 ,     E_pot =  -0.563 ,      E_tot = -0.25
          E_tot - E_init = 1.64e-10
          (E_tot - E_init) / E_init = -6.58e-10
  [1, 2] :  a = 1.8175e-01 ; e = 9.4611e-01 ; T = 6.8852e-01
t = 2 (after 7067, 0, 6 steps <,=,> t)
          E_kin = 0.171 ,     E_pot =  -0.421 ,      E_tot = -0.25
          E_tot - E_init = -9.53e-10
          (E_tot - E_init) / E_init = 3.81e-09
  [1, 2] :  a = 1.8109e-01 ; e = 9.2257e-01 ; T = 6.8479e-01
t = 0 (after 0, 0, 0 steps <,=,> t)
          E_kin = 0.25 ,     E_pot =  -0.5 ,      E_tot = -0.25
          E_tot - E_init = 0
          (E_tot - E_init) / E_init = -0
  [1, 2] :  a = 1.8007e-01 ; e = 8.8222e-01 ; T = 6.7900e-01
t = 1 (after 4329, 0, 6 steps <,=,> t)
          E_kin = 0.313 ,     E_pot =  -0.563 ,      E_tot = -0.25
```

```
          E_tot - E_init = 1.64e-10
          (E_tot - E_init) / E_init = -6.58e-10
  [1, 2] :  a = 1.8175e-01 ; e = 9.4611e-01 ; T = 6.8852e-01
```

Alice: It works all right! That much I have to credit you for. But I can't say I like it.

Bob: We can do a mopping up operation later. Let's move on.

4.3 Time of Binary Formation

Alice: Okay. So you have managed to wrestle the story with the binary formation information out of the integrator. What we really need is not the story, but the time of first binary formation, given the specified limit on the size of the binary. How do you propose to do that?

Bob: Simple, just ask for it, that's the beauty of our ACS data format. Here it is, as `test2.rb`

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

require "acs.rb"

options_text= <<-END

  Description: Find when the first binary froms, print that time and exit
  Long description:
    This program accepts a stream of Nbody snapshots, and checks the story of
    each one until it finds the first reported binary.  It then prints the
    time at which that binary was first found, on the standard output, and
    exits.

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

    example:

        kali mkplummer.rb -n 4 -s 1 | kali world4.rb -t 1000 -x 0.25 | kali #{$0}


  END

class NBody
  attr_reader :time
end
```

```
clop = parse_command_line(options_text)

while nb = ACS_IO.acs_read
  raise "class #{nb.class} is not NBody" unless nb.class == NBody
  nb.story.each_line do |line|
    if line =~ / :  a = /
      print nb.time, "\n"
      exit
    end
  end
end
```

Alice: You see the payoff that we got for *not* being blunt, and taking the time to set up a well crafted ACS IO mechanism.

Bob: Noted. Let's see whether this works, but let me spare you the lots and lots of output that we got in the previous run. I'll just report the end of the story. Taking the same initial conditions, here we go once again:

```
|gravity> kali mkplummer.rb -n 4 -s 1 > tmp.in
==> Plummer's Model Builder <==
Number of particles: N = 4
pseudorandom number seed given: 1
Screen Output Verbosity Level: verbosity = 1
ACS Output Verbosity Level: acs_verbosity = 1
Floating point precision: precision = 16
Incremental indentation: add_indent = 2
             actual seed used: 1
|gravity> (kali world4.rb -t 1000 -x 0.25 < tmp.in | kali test2.rb > tmp.out) >& t
```

Alice: Quite a relief, to see such a silent run! What did you catch in the standard output?

Bob: Hopefully the time of first binary formation:

```
|gravity> cat tmp.out
1.0
```

Alice: Congratulations! Of course, it is not yet the *actual* time of binary formation. It is the first time the binary has been spotted in a story, as part of the particle output. So if we do this type of output once every time unit, a binary forming at, say, time 3.14 will be noticed only at time 4.0.

Bob: Correct. And there is no reason not to do an output more often, as soon as we find a good way to suppress all the unnecessary visible output.

Alice: did the standard error channel give the write diagnostics?

Bob: Presumably, but let's check:

```
|gravity> tail tmp.err
t = 1 (after 4329, 0, 6 steps <,=,> t)
          E_kin = 0.313 ,     E_pot =  -0.563 ,      E_tot = -0.25
          E_tot - E_init = 1.64e-10
          (E_tot - E_init) / E_init = -6.58e-10
  [1, 2] :  a = 1.8175e-01 ; e = 9.4611e-01 ; T = 6.8852e-01
t = 2 (after 7067, 0, 6 steps <,=,> t)
          E_kin = 0.171 ,     E_pot =  -0.421 ,      E_tot = -0.25
          E_tot - E_init = -9.53e-10
          (E_tot - E_init) / E_init = 3.81e-09
  [1, 2] :  a = 1.8109e-01 ; e = 9.2257e-01 ; T = 6.8479e-01
```

Looks like it!

4.4 A Test with One Run

Alice: What is next? I presume you're going to generate these times of first binary formation user a higher-level Ruby program?

Bob: Exactly. And we know how to do that, as we have demonstrated with our test program `test0.rb`. We can apply the same procedure. Let me call the new program `test3.rb`.

Just to stay on the safe side, let me ask `test3.rb` to orchestrate only a single run, for now, using `test2.rb` to produce the time in which the first binary forms in that run. Once we know how to do that, we can ask it to perform a whole series of runs.

How about this:

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

require "acs.rb"

options_text= <<-END

  Description: Gathers statistics for binary formation times
  Long description:
    This program runs a number of simulations, starting with a Plummer model,
```

```
  and reports some statistics concerning the times of first binary formation.
  NOTE: TEST VERSION: prints only the result from one run

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

  example:

      kali #{$0} -n 4 -s 1 -x 0.25


Short name:           -n
Long name:            --n_particles
Value type:           int
Default value:        1
Variable name:        n
Print name:           N
Description:          Number of particles
Long description:
  Number of particles in a realization of Plummer's Model.

  Each particles is drawn at random from the Plummer distribution,
  and therefore there are no correlations between the particles.

  Standard Units are used in which G = M = 1 and E = -1/4, where
    G is the gravitational constant
    M is the total mass of the N-body system
    E is the total energy of the N-body system


Short name:           -s
Long name:            --seed
Value type:           int
Default value:        0
Description:          pseudorandom number seed given
Print name:
Variable name:      seed
Long description:
  Seed for the pseudorandom number generator.  If a seed is given with
  value zero, a preudorandom number is chosen as the value of the seed.
  The seed value used is echoed separately from the seed value given,
  to allow the possibility to repeat the creation of an N-body realization.

    Example:

      |gravity> kali mkplummer1.rb -n 42 -s 0
      . . .
```

```
        pseudorandom number seed given  : 0
                    actual seed used    : 1087616341
        . . .
        |gravity> kali mkplummer1.rb -n 42 -s 1087616341
        . . .
        pseudorandom number seed given  : 1087616341
                    actual seed used    : 1087616341
        . . .


  Short name:           -x
  Long name:            --max_semi_major_axis
  Value type:           float
  Default value:        #{VERY_LARGE_NUMBER}
  Description:          Maximum value of semi major axis
  Variable name:        max_semi_major_axis
  Long description:
    This option allows the user to limit the number of binaries detected
    by discarding binaries with a semi-major axis larger than the specified
    number.  This is useful in situation such as the initial state for a cold
    collapse situation, where every star is formally bound to every other star.


  END

c = parse_command_line(options_text)

print `kali mkplummer.rb -n #{c.n} -s #{c.seed} --verbosity 0 | kali world4.rb \
-t 1000 -x #{c.max_semi_major_axis} --verbosity 0 | kali test2.rb --verbosity 0`
```

Alice: Yes, that should do just the type of thing we've seen before. And by setting the normal verbosity level to 0 you surpress the screen output. Good! That will clean up things. And since the `acs_verbosity` will retain its default value of 1, the binary diagnostics will still be passed on to the story in the particle output, so that test2.rb can use it to produce the final number.

Bob: Let's see whether it indeed does what it should do, once more with the same numbers as before:

```
|gravity> test3.rb -n 4 -s 1 -x 0.25
test3.rb: .
```

Alice: Wonderful. Our first automatically generated lab result number.

Bob: No stopping us now!

nil nil nil nil nil

Chapter 5

Literature References

[to be provided]