The Art of Computational Science

The Maya Open Lab Development Series,

volume 1

The 2-Body Problem:

Forward Euler, in Ruby

Piet Hut and Jun Makino

September 14, 2007

Contents

Preface **7**

0.1 N-Body Simulations 7
0.2 Ruby 7
0.3 Speed 8
0.4 Programming Issues 8
0.5 Acknowledgments 9

1 Setting the Stage **11**

1.1 Ruby 11
1.2 A Body Class 13
1.3 The irb Interpreter 15

2 Introducing the Players **19**

2.1 Creating a Body 19
2.2 Initializing a Body 20
2.3 Assigning New Values 22
2.4 Syntactic Sugar 25

3 Pretty Printing **29**

3.1 Tinkering with the Output 29
3.2 A Body-to-String Converter 31
3.3 Taming the Arrays 33
3.4 Executing Ruby Files 35

4 Input and Output **39**

4.1 Double Precision Input/Output . . . 39
4.2 A Simple Version . . . 40
4.3 Trying It Out . . . 42
4.4 A Surprise . . . 44

5 The Two-Body Problem 47

5.1 Equations of Motion . . . 47
5.2 Relative Motion . . . 48
5.3 Modularity . . . 50
5.4 The First Integrator . . . 51
5.5 Writing Clean Code . . . 53
5.6 Where the Work is Done . . . 55

6 Orbit Integration 57

6.1 Two Dimensions . . . 57
6.2 Forward . . . 58
6.3 Error Scaling . . . 59
6.4 A Full Orbit . . . 60
6.5 Taking Time . . . 62
6.6 Convergence . . . 63

7 Debugging 65

7.1 A Vector Class . . . 65
7.2 Debugging . . . 67
7.3 An Extra Safety Check . . . 69
7.4 And Yet It Is . . . 70
7.5 Using a Microscope . . . 72
7.6 Following the Trail . . . 73
7.7 Extreme Programming . . . 75

8 Formula Translating 77

8.1 A Body with Vectors . . . 77
8.2 Shrinking Code . . . 78
8.3 FORTRAN . . . 80

8.4 An Old Friend 81
8.5 Magic 82
8.6 A Matter of Taste 84
8.7 *Voila* 85

9 Energy Conservation 87
9.1 Estimating Numerical Errors 87
9.2 A Driver 89
9.3 The Body Class 90

10 Diagnostics 93
10.1 Evolve 93
10.2 Brevity and Modularity 94
10.3 Keeping Time 95
10.4 More Brevity 97
10.5 Ruby Smiling 98
10.6 A Here Document 100

11 Testing Forward Euler 103
11.1 Starting Again 103
11.2 Linear Scaling 104
11.3 A Full Orbit 105
11.4 Zooming In 107
11.5 Pericenter Passage 108
11.6 Apocenter Passage 109

12 Analytical Checks 113
12.1 Celestial Mechanics 113
12.2 The Role of the Masses 115
12.3 Reduced Mass 116
12.4 Wrapping It Up 118

13 Literature References 121

Preface

0.1 N-Body Simulations

The current manuscript is Volume 1 in the *Development Series* of the *Maya Open Lab* project, for constructing a laboratory for simulations of dense stellar systems in astrophysics. The Maya project in turn is the first concrete example of projects developed in an initiative titled *The Art of Computational Science (ACS)*. The aim of the larger ACS series, of which the Maya Series will be a part, is to provide real-world guides to designing, developing, using, and extending computational laboratories. We refer to the preface of Volume 0 for more background.

This volume presents the first steps on a road toward constructing computer codes for N-body simulations. We start in a very modest way, by considering only the 2-body problem. In addition, the only integration scheme we use here is the forward Euler scheme. While of little practical use, its extreme simplicity makes it a useful toy algorithm to illustrate some of the general issues involved in building N-body codes. Various additional algorithms, such as the leapfrog, and a few Runge-Kutta methods, will be discussed in Volume 2, also within the context of the 2-body problem. The general N-body problem will be addressed in Volume 4.

0.2 Ruby

We have decided to present our initial codes in the Ruby language, rather than the more traditional choices of Fortran, C, or C++. Ruby is a fully object-oriented, flexible and extremely powerful scripting language. Just like assembly language is far more expressive than machine language, and Fortran or C or C++ are in turn far more expressive than assembly languages, Ruby is an example of a new generation of languages that occupies yet one level higher in the hierarchy of shifting the burden of programming from the programmer to the compiler – or in Ruby's case, to the interpreter.

Within a few weeks of working with Ruby, we already became convinced of its superiority for our purpose of developing a state-of-the-art computer code for dense stellar system, with sufficient flexibility to allow us to set up a framework for connecting codes old and new, from ancient legacy codes to new products written in a variety of languages. Among the many attractive features of Ruby, it was especially its practical ease of allowing rapid prototyping that appealed to us most.

0.3 Speed

An essential requirement for any computer code to be used in scientific simulations is its ability to deliver sufficient raw speed. Without that, no matter how much a language excels in elegance and clarity, it would not be practically useful. Therefore, one of our first checks, after turning to Ruby, was to investigate its speed. We knew that we could not expect any significant speed from Ruby out-of-the-box. Unlike C or Fortran, Ruby was designed expressly for flexibility and clarity, rather than for speed. At the same time, Ruby was designed in such as way as to make it easy and natural to replace some compute-intensive core modules with equivalent modules written in more traditional, lower-level languages.

Indeed, after only a few days of experimentation, we convinced ourselves that we could find several ways in which to regain the speed of a C or Fortran code within Ruby, by replacing only a very small fraction of the code by functions written in C, while leaving the vast majority of the code complexity within Ruby's domain. We will describe this process of speeding up Ruby in great detail only in Volume 5, after we have presented Ruby in sufficient detail in practical applications, to set the stage for an informed discussion about what does and does not need such hand-coded speed-up.

0.4 Programming Issues

The current volume contains a rather detailed introduction, from scratch, to how Ruby can be used for numerical applications. We expect this volume to be of use to a broad public, interested in trying out Ruby for applications in computational science, starting with the N-body problem as a comparatively simple example. Even those who already have gained considerable expertise in Ruby may want to consult this volume for a real-life example of how to set up compute-intensive applications, something that is generally left out of Ruby literature, both on introductory and on advanced levels.

In addition, we present reports of bug-tracing sessions, with a degree of realistic detail seldom encountered in the literature. It is our belief that the essential art of debugging code can only be learned through practice. And to the extent that books can help one to get exposed to this type of practice, examples have to come directly from applications, or else they will be, at best, only of theoretical value. Chapter 7, *Debugging*, is devoted in its entirety to this topic.

0.5 Acknowledgments

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

Finally, it is our pleasure to thank Douglas Heggie, Stephan Kolassa, Ernest Mamikonyan, Bill Paxton, Michele Trenti, John Tromp and Jason Underdown for their comments on the manuscript.

Piet Hut and Jun Makino

Kyoto, July 2004

Chapter 1

Setting the Stage

1.1 Ruby

Alice: So here we are, ready to begin writing a toy model to simulate a dense stellar system. We have decided to begin with a very simple integrator, since we will use the material in a course for students with little or no prior background in differential equations and numerical methods.

Bob: That pretty much defines the first integrator to discuss: one based on the forward Euler integration scheme. To take one step, you add to each variable its derivative multiplied by the value of the time step. In other words, you just step forward by incrementing each variable by its derivative, as specified by the differential equation.

$$\frac{dx}{dt} = f(x)$$

$$x_{\rm new} = x_{\rm old} + f(x_{\rm old})\Delta t$$

Alice: The only way to make that sentence clear to someone with little experience in this area is to give an example. So let's code it up.

Bob: We decided to do this in Ruby, but neither of us have any background in the language. To get started, I had a look at an introductory book, yesterday evening. It is called *Programming Ruby* by Dave Thomas and Andy Hunt, a.k.a. the Pragmatic Programmers. I found it quite useful, and it is also well written.

Alice: I'll have a look at it then as well. Did you find enough to get started?

Bob: I guess so. Let's try and see. Before getting to gravity and equations of motion, how about building a simple skeleton first, something that just reads in the data and prints them out, without doing any integrating?

Alice: But before doing that, we need to decide upon a data format. Let's not get too fancy, for now. How about just listing for each particle the seven numbers on one line: mass, position, and velocity, with the latter two each being a three-dimensional vector, and therefore with three components each?

Bob: Well, in that case, let's start with an even simpler problem, with reading and writing the data for a single particle. For an object oriented language like Ruby, that suggests that we create a class `Body` for a particle in an N-body system.

Alice: Can you remind me what a class is?

Bob: I thought you were going to tell me, while pointing out how important they are for your obsession with modular programming!

Alice: Well, yes, I certainly know the general idea, but I must admit, I haven't really worked with object oriented languages very much. At first I myself was stuck with some existing big codes that were written in rather arcane styles. Later, when I got to supervise my own students and postdocs, I would have liked to let them get a better start. However, I realized that they had to work within rather strict time limits, within which to learn everything: the background science, the idea of doing independent research, learning from your mistakes, and so on.

The main problem for my students has been that there is hardly any literature that is both interesting for astrophysicists and inspiring in terms of a really modern programming attitude. When students are pressed for time and eager to learn their own field, they are not likely to spend a long time delving into books on computer science, which will strike them as equally arcane, for different reasons, as the astrophysical legacy codes.

Given that there was no middle ground, I did not want them to focus too much on computational techniques, because that would have just taken too much time. My hope is, frankly, that our toy model approach will bridge this huge gap, between books that are clear but of very limited use, and codes that are useful but of very limited clarity.

Bob: You seem to find a way to introduce a world wide vision for every small task that you encounter. As for me, I'm happy to just build a toy model, and if students will find it helpful, I'd be happy too. But just to answer your question, defining a class is just a way to bundle a number of variables and functions together. Just like a number of scalar values can be grouped together in one array, which can stand for a physical vector for example, you can group a more heterogeneous bunch of variables together. You do this mainly for bookkeeping reasons, and to keep your program simpler and more robust.

In practice, a good guide to choosing the appropriate class structure for a given problem is to start with the physical structures that occur naturally, but that may not always be the best option, and certainly not the only one.

1.2 A Body Class

Bob: For example, a single particle has as a minimum a mass, a position and velocity. Whenever you deal with a particle, you would like to have all three variables at hand. You can't put them in a single array, because mass is a scalar, and the other two variables are vectors, so you have to come up with a more general form of bundling.

The basic idea of this kind of programming is called object-oriented programming. In many older computer languages, you can pass variables around, where each variable can contain a number or an array of numbers; or you may pass pointers to such variables or arrays. In an object-oriented programming language, you pass bundles of information around: for example all the information pertaining to a single particle, or even to a whole N-body system. This provides convenient handles on the information.

If you look in a computer science book, you will read that the glorious reason for object-oriented programming is the ability to make your life arbitrarily difficult by hiding any and all information within those objects, but I don't particularly care for that aspect.

Alice: I do think encapsulation has its good sides, but we can come back to that later. What exactly is it that you can put inside an object? I guess an object can contain internal variables. Can it contain functions as well?

Bob: To take the specific case of Ruby, a typical class contains both. For a class to be useful, at least you have to be able to create an instance of a class, so you need something like what is called a constructor in C++. In the case of Ruby, like in the case of C++, you have the freedom to define an initializer, through which you can create an instance of a class with your desired values for the internal variable. Or you can choose not to define an initializer, that is fine too.

Note as a matter of terminology that what I have called a function, or what would be called a subroutine in Fortran, is called a method in Ruby. There are other objects in Ruby, besides classes. Sometimes you have a group of functions that are either similar or just work together well, and you may want to pass them around as a bundle. In Ruby, such bundles of functions are called modules. But to get started, it is easier to stick to classes for now.

Alice: Ah, that is nice! Does this mean that we can define an integration algorithm as a module, independent of the particular variables in the classes that define a body or an N-body system? I mean, can you write a leapfrog module that can propagate particles, independently of their type? You could have point particles in either a two-dimensional of a three-dimensional world. Or you could have particles with a finite radius, that stick together when they collide; as long as they are not too close, they could be propagated by the same leapfrog module.

Bob: You really have an interesting way of approaching a problem. We haven't even defined a single particle, and you are already thinking about general modules that are particle independent. I suggest we first implement a particle. I think this is how we can introduce a minimal class for a single particle:

```
class Body

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

end
```

In a few minutes, we can go through the precise meaning of these constructs, but here is the general idea: you can give three arguments in a call to `initialize`, to specify the mass, position, and velocity. If you don't specify some of these arguments, they will acquire the default values that are given here by all the zeroes in the first line of the definition of `initialize`. The next line assigns these values to internal variables that have names starting with a `@` sign.

Alice: That is remarkably short and simple! In fact, it seems too simple. I am surprised that we do not have to declare the internal variables. In other languages that I am familiar with, it is essential that you tell the computer which memory places to set aside before naming them.

Bob: Ruby is dynamically typed. This means that the type of a variable is determined at run time. In other words the type of a variable is simply the type of the value that is assigned to a variable.

Alice: And you can change the type of that value, whenever you want? Can we try that? I'd like to see the syntax of how you do that.

Bob: Before we do that, just one thing: staring at this amazingly simple class definition makes me realize both how similar it is to what you would write in `C++`, and how different.

Alice: Indeed, the logical structure of `C++` class definitions is very similar.

Bob: But the big difference is that a `C++` class definition is quite a bit longer.

Alice: I'm curious how much longer. Do you remember how to write a similar particle class in `C++`?

Bob: That shouldn't be too hard. Always easiest to look at an existing code. Ah, here I have another C++ code that I wrote a while ago. Okay, now I remember. Of course. Here is how you do it in `C++`:

```
##include  <iostream>
using namespace std;
```

```
class body
{
private:
    double pos[3];
    double vel[3];
    double mass;
public:
    body(double inmass, double inpos[3], double invel[3]){
mass = inmass;
for(int i=0;i<3;i++){
    pos[i] = inpos[i];
    vel[i] = invel[i];
}

    }
    void print(){
cout << mass <<endl;
cout << pos[0] <<" "<< pos[1] <<" "<< pos[2] <<endl;
cout << vel[0] <<" "<< vel[1] <<" "<< vel[2] <<endl;
    }

};
main()
{
    double zero3[3]={0.0,0.0,0.0};
    body x =  body(0.0,zero3,zero3);
    x.print();
}
```

1.3 The irb Interpreter

Alice: That is quite an impressive difference, between Ruby and C++! But aren't you cheating a bit? That last part, with the `main` function down below, does not occur in your short and sweet Ruby class definition.

Bob: It doesn't occur there, because you don't need it. All that `main` does for you is create an object and then printing its internal values. You can let Ruby do that for you without even asking for it.

Here is what you do. The easiest way to work with Ruby is to use the command `irb`. The acronym stands for *interactive Ruby*. You invoke it by simply typing `irb` on the command line. Now as soon as you create an instance of a class in Ruby, the interpreter echoes the content to you, for free!

Alice: I'll try it out. But rather than wrestling with a whole particle, I prefer to start with a single variable, to see how Ruby functions at its most basic level. Also, remember that I asked you whether you can change the type of the value of a variable, whenever you want? I want to see that for myself. One thing at a time!

Let me introduce an identifier `id`. I will first give it a numerical value, and then I will assign to it a string of characters, to give it a name. Since Ruby is friendly enough not to insist on declaring my variables beforehand, I presume I can just go ahead and use `id` right away.

```
|gravity> irb
irb(main):001:0> id = 12
=> 12
```

And as you predicted, a value gets produced magically. But where does that come from?

Bob: Just like in C, variables are not the only things that have values. In fact, every expression has a value. And `irb` makes life more clear by echoing the value of each line as soon as you enter it.

Alice: I like that. It will make debugging a lot easier. Okay, let me try to change the type of `id`.

```
irb(main):002:0> id = cat
NameError: undefined local variable or method ‘cat’ for main
  :Object
from (irb):2
```

Bob: Ah, Ruby treats your `cat` in an equally friendly way as your `id`, assuming it is itself a name of a variable (or a method), rather than a content that can be assigned to a variable.

Alice: But that line works fine when I write shell scripts. Since Ruby is called a scripting language, I thought it might work here too.

Bob: Each scripting language has different conventions. In your shell case, I bet you have to invoke the value of a variable `cat` by typing `$cat`, each time you use it. Ruby has another solution: typing `cat` echoes the value of `cat`. When you want to introduce a string consisting of the three letters `c`, `a`, and `t`, you type `"cat"`.

Alice: Here goes:

```
irb(main):003:0> id = "cat"
=> "cat"
```

It worked! And presumably `id` has now forgotten that it ever was a numerical variable.

Bob: Indeed. And at any time you can ask Ruby what the type of your dynamically typed variable currently is. Any variable is an instance of some class. And the class it belongs to in turn has a method built in, not surprisingly called `class`, which tells you the type of that class. In Ruby, you invoke a method associated with a variable by writing that variable followed by a period and the method name.

Alice: I find it surprising, if you ask me; I would have expected something like `type`. But I'll take your word for it.

```
irb(main):004:0> id.class
=> String
irb(main):005:0> id = 12
=> 12
irb(main):006:0> id.class
=> Fixnum
```

Hey, that is nice! You can immediately check what is going on. Let's see what happens when I type in the text of the `Body` class declaration above.

```
irb(main):007:0> class Body
irb(main):008:1>   def initialize(mass = 0, pos = [0,0,0], vel = [0,0,0])
irb(main):009:2>     @mass, @pos, @vel = mass, pos, vel
irb(main):010:2>   end
irb(main):011:1> end
=> nil
```

I see another nice feature. I had been wondering about the meaning of the `:0` after each line number. That must have been the level of nesting of each expression. It goes up by one, each time you enter a block of text that ends with `end`.

Bob: And since you only give the definition of `Body` without yet creating any of its instances, there is no value associated with it. Here `nil` means effectively 'undefined'.

Alice: But we have just defined the `Body` class; why does Ruby claim it is undefined?

Bob: In Ruby, the definition of a class does not return a value, since there is no reasonable answer that can be given to a non-existent question. But since the interpreter wants to echo something, it just returns, 'nil' as meaning an undefined value, or literally nothing. And this has nothing to do with the definition of a class. This may sound more complicated than it is, but it is quite logical.

Alice: I see, yes, that makes sense. So the class `Body` has only one function, starting with `def` and ending with the inner `end`, correct?

Bob: Indeed. And the last `end` is the end of the class definition, that starts with `class Body`. Note the grammatical rule that the name of a class such as `Body` always starts with a capital letter. The names of normal variables, in contrast, start with a lower case letter: we have three such variables, `mass`, `pos`, and `vel`. All three are given here as possible parameters to the initialization function `initialize`.

Chapter 2

Introducing the Players

2.1 Creating a Body

Alice: So how do you create a particle?

Bob: According to the book I read, you can simply type `b = Body.new`, to get a new particle with the default values, all zero in this case.

Alice: Wait a minute. According to the rules you just told me, it should be `b = Body.initialize`, since `initialize` is the name of the one method that we have defined in our `Body` class.

Bob: All I can tell you is what I read in the manual. Let's try it both ways!

```
irb(main):012:0> b = Body.new
=> #<Body:0x4008a1b0 @mass=0, @vel=[0, 0, 0], @pos=[0, 0, 0]>
irb(main):013:0> c = Body.initialize
NoMethodError: private method ‘initialize’ called for Body:Class
        from (irb):13
```

Alice: I guess the writer of Ruby decided that typing `new` is both shorter and more natural than typing `initialize`.

Bob: No, it is more subtle than that. I now see what is going on. There is a fundamental distinction between *class variables* and *instance variables* and similarly between *class methods* and *instance methods*. Here is the idea.

The method `new` is a *class method*. There is only one way to create a new particle. But as soon as you have done that, and identified that new particle with the variable `b` as a handle, then you can use the *instance method* `initialize` to give initial values to that new particle. Let us say that we create two particles. Another way to say this is that we create two different objects of the same class `Body`. Yet another way to describe this: we create two *instances* of the one *class* `Body`. The process of creating the two instances is the same. But once an instance has come into being as a separate particle, then it can be given individual values for its mass, position and velocity. This is why the *class method* `new` calls the *instance method* `initialize` to give each particle its proper internal values.

Note also the hierarchy. When you issue the command `new`, then in turn `new` invokes the *instance* method `initialize`, in a way that is hidden from the user. Note that the error message mentioned a `private method 'initialize'` of the class `Body`. This means that the *class* method `initialize` is there alright, it just is not publicly available – which means in turn that you cannot use the function from outside the scope of the class definition. The method `Body.new`, however, does have access; it is defined to be part of the class `Body`, so it does have access to all the methods of `Body`, even those that are private.

Finally, at the danger of confusing you, but to make the explanation complete, note that methods are by default always public. This makes sense, since in most cases you define methods as ways to interact with particles. It is relatively rare to define private methods, and you do that only if you have a good reason: generally because you have only one specific use for it, and a use that is strictly internal to the module. Clearly, the details of *how* to initialize a module is something that is nobody else's business; the only thing the user needs to do is to specify *what* the values will be with which the particle will be initialized. Now the only place where it is reasonable to specify initial values is . . . initially! And the only method that you invoke to make a new `Body` is when you give the command `Body.new`. This makes it crystal clear that `new`, and no other method but `new`, should be allowed to pass initial values to `initialize`. So `new` really does three things: it (1) creates a standard empty version of a `Body`; (2) gives it a unique `id`, thereby turning it effectively into an instance of the class `Body`; (3) hands over the initial values to `initialize` which in turn assigns those values to the *instance variables* `@mass`, `@pos`, and `@vel`.

Alice: I am glad you spoke slowly, since that was quite a stack of ideas to keep track of, but it makes sense. And I had to smile seeing you praising the wisdom of encapsulation. But what you explained is a clear and consistent approach, once you see what is going on.

2.2 Initializing a Body

Bob: Phew, yes, that was a long explanation, but I'm very glad you asked, and that we actually typed the wrong thing, since now I finally understand completely what I read yesterday in the introduction to Ruby. And I also understand now why there are two types of internal variables within a class. Just as there are *class methods* and *instance methods*, there are similarly *class variables* and *instance variables*. Instance variables always start with a single `@` sign, while class variables always start with a `@@` sign. In our `Body` definition we have introduced only instance variables. That makes sense, since different particles may have different masses, likely have different velocities, and certainly will have different positions.

Alice: What would be an example of a class variable?

Bob: Any variable that has a value that is shared by all instances of that class. In the case of a `Body`, we could introduce a class variable `@@time` that gives us the current time for all bodies in the system. If we have an integration scheme with a shared time step, it might be a good idea to encapsulate the time within the `Body` class. However, if we then switch to an individual time step scheme, we have to give each particle its own time of last update. In that case we should specify it as `@time`, to make it an instance variable.

Alice: I see. Yes, that seems like a good example. So now I feel completely comfortable in using `Body.new`. It's time to inspect the value that was echoed upon creation of a new body. It looked quite complex.

```
irb(main):012:0> b = Body.new
=> #<Body:0x4008a1b0 @mass=0, @vel=[0, 0, 0], @pos=[0, 0, 0]>
```

I recognize the values of the mass, and the components of position and velocity, which are all zero by default. But what is that hexadecimal number doing there on the left?

Bob: After giving the class name of the object, it prints the value of the pointer to the object, a unique integer that is presumably somehow associated with the location in memory of the object. The object is what we called an instance of the class that it belongs to. The pointer may be useful for debugging, perhaps, but for now we can ignore that number.

Alice: A while ago you said that `mass`, `pos`, and `vel` are possible parameters. What do you mean with 'possible'?

Bob: I meant that you don't have to use them. When you leave all of them out, you get the default values. When you specify one or more of them, you have to specify them from left to right. For example, when you type `b = Body.new(1)` you give the particle mass an initial value 1, rather than the default value 0, while keeping the other values 0. At least that's what I read.

Alice: easy enough to try:

```
irb(main):014:0> c = Body.new(1)
=> #<Body:0x401018b8 @mass=1, @vel=[0, 0, 0], @pos=[0, 0, 0]>
```

Good! The mass is indeed 1, and this particle `c` is distinct from our previous particle `b`, since it has a different `id`. But now how do you give a value to a vector?

Bob: That I'm not sure yet. We should experiment and try it out.

Alice: that's what I like about software. You can break things without hurting and destroying something. Okay, here are some non-zero values for some of the position and velocity components:

```
irb(main):015:0> d = Body.new(1, 0.5, 0, 0, 0, 0.7, 0)
ArgumentError: wrong # of arguments(7 for 3)</tt>
from (irb):15:in 'initialize'
from (irb):15:in 'new'
from (irb):15
```

Bob: How nice to get such clear instructions! Quite a bit more helpful than *segmentation fault* or something cryptic like that. The debugger works its way out from the innermost nesting, back to its caller function `initialize` that is in turn called by `new` – just as I told you!

Alice: Indeed. And yes, I can guess now what went wrong. I bet I should have presented the positions and velocities as arrays. That would indeed make 3 arguments in total, instead of 7.

```
irb(main):016:0> d = Body.new(1, [0.5, 0, 0], [0, 0.7, 0])
=> #<Body:0x400df1f0 @mass=1, @vel=[0, 0.7, 0], @pos=[0.5, 0, 0]>
```

So there. It worked!

2.3 Assigning New Values

Bob: Now that you have successfully created and initialized a particle, I bet you would like to change some of its internal state.

Alice: If we are going to integrate the orbit of a particle, we'll certainly have to update its position and velocity. But let me try first with the simplest case, the scalar value of the mass. How about a bit of mass loss?

```
irb(main):017:0> Body.@mass = 0.9
SyntaxError: compile error
(irb):17: syntax error
Body.@mass = 0.9
          ^
        from (irb):17
        from ^C:0
```

Bob: That's what you get for not reading the manual! In Ruby, the internal variables of a class are all private. There is no way that you can access them from outside, either for reading or for writing. If I would have designed a language, I wouldn't have been so strict, but that's the way it is.

Alice: I actually think that is a very *good* feature of Ruby. If you change some piece of code you or someone else wrote a long time ago, it is good to have a clear protocol about how to access internal data. A house has walls, an animal has a skin, a cell has a cell wall, and there are good reasons for that!

Bob: Rather than getting into a modularity argument again, the good news is that Ruby makes both of us happy: you have your encapsulation, and I can almost pretend that it wasn't there, since it is very easy to set up a mechanism to get around this cellular approach, with a very natural syntax. Let me show you how to do this in the most straightforward way once, but then I'll move on to a much better shortcut.

In order to change the mass we have to add the following line to the class definition.

```
irb(main):018:0> class Body
irb(main):019:1>   def mass
irb(main):020:2>     @mass
irb(main):021:2>   end
irb(main):022:1>   def mass=(m)
irb(main):023:2>     @mass = m
irb(main):024:2>   end
irb(main):025:1> end
=> nil
```

Alice: Wait a minute! You are now giving a new definition of the class `Body`. Doesn't that override the older definition that we have given before?

Bob: Ah, but that's the beauty of Ruby, one of its many beauties, that you can always add new features to a class, whenever you want! Each time you define features of a class or a module that already exists, Ruby adds those features to whatever is already there.

Alice: It does look as if Ruby will make both of us happy! You don't feel encapsulated, because you can add anything any time, and I still feel modular, since I know it all winds up internally inside one class definition. As for the two methods you introduced, they are effectively 'get' and 'set' functions, I presume?

Bob: Yes. The first method echoes the value of the internal variable `@mass`. While the variable `@mass` itself is private, the method `mass` is public by default, so this gives you the simple way I promised, to access data that would be hidden otherwise.

The second method allows you to change the internal state of an object. Note that the equal sign is part of the name. The method is called `mass=`, with one parameter `m`. And the effect of calling this method is to assign the value of its parameter to the internal variable `@mass`.

Alice: The syntax `mass=(m)` looks rather odd, if you ask me, this combination of an equal sign and parentheses. Anyway, let's see whether I understand it correctly. I will try to read the old value of the mass and then write a new value into the same variable

```
irb(main):026:0> d.mass
=> 1
irb(main):027:0> d.mass=(2)
=> 2
irb(main):028:0> d.mass
=> 2
```

Great! It works as advertised. Of course there was no need to type the third line, but I just wanted to be sure that everything was consistent. However, I still don't like the syntax of `=()`.

Bob: Ah, but here is where Ruby's freedom of expression helps out: those parentheses are optional. You needed them in the method definition, to tell the Ruby interpreter that you were dealing with a parameter that was an argument of a method. But once defined, you can leave them out when you give a command. And you can even introduce a space between `mass` and `=` if you like. In general, of course you cannot introduce spaces in the middle of a name, but in the case of a method name ending on `=`, this is allowed, but only just before the equal sign. This is one of the many places where Ruby caters to the pleasure of the user, and not to the pleasure of someone with a rigid logical bend. Here are a couple examples.

```
irb(main):029:0> d.mass = 3
=> 3
irb(main):030:0> d.mass=4
=> 4
```

Alice: Much better! I like the pragmatic compromise between clarity and ease of use. Let me do it once more to show the symmetry between reading and writing:

```
irb(main):031:0> new_mass = 8
=> 8
irb(main):032:0> old_mass = d.mass
=> 4
irb(main):033:0> d.mass = new_mass
=> 8
irb(main):034:0> d
=> #<Body:0x400df1f0 @mass=8, @vel=[0, 0.7, 0], @pos=[0.5, 0, 0]>
```

As it should be. Now what about the position and velocity?

2.4 Syntactic Sugar

Bob: You may guess how we can provide 'get' and 'set' functions for the other internal variables. Here is how we deal with the position:

```
irb(main):035:0> class Body
irb(main):036:1>   def pos
irb(main):037:2>     @pos
irb(main):038:2>   end
irb(main):039:1>   def pos=(p)
irb(main):040:2>     @pos = p
irb(main):041:2>   end
irb(main):042:1> end
=> nil
irb(main):043:0> d.pos
=> [0.5, 0, 0]
irb(main):044:0> d.pos = [0.5, 0, 0.1]
=> [0.5, 0, 0.1]
```

Alice: The way you use the 'set' function is different, in the sense that for the position you now provide a vector, rather than a scalar. But everything else is the same, in particular the definition. It is nice that Ruby is so homogeneous in its notation: it doesn't care at the level of the definition whether a variable describes a scalar or a vector.

Bob: That is a nice aspect of dynamic typing. And it also invites a more compact syntax. Rather than writing everything all over again for `vel` instead of `pos` above, you can use a convenient shorthand notation, a piece of syntactic sugar as these are sometimes called. The six lines above that define the two methods to read and write `pos` values can be replaced by the following single line:

```
attr_accessor :pos
```

You can even add more than one variable on one line. This line:

```
attr_accessor :mass, :pos, :vel
```

replaces no less than eighteen lines of code written out in full.

Alice: Why the cryptic name?

Bob: Because there are two more elementary commands:

```
attr_reader :pos
```

replaces the first method definition above, and

```
attr_writer :pos
```

replaces the second method definition. The word `accessor` is meant to suggest that you can both read and write, *i.e.* you have two-way access to the variables, from outside.

Alice: I certainly prefer this compact notation. But if we now add that to our class definition, I may get confused, with bits and pieces of the class definitions spread here and there throughout our `irb` session. Is it possible to put everything in a file, and somehow let `irb` have access to the definitions in that file?

Bob: Yes, that can be done. First we put all the definitions in the file `body2.rb`[1], where `.rb` is the standard ending for a file name that contains Ruby code. I added the `2` here because this is our second attempt to define a `Body` class, and I'm sure there will be more. I'll type it straight into the file, since it's so short. Here it is, `body2.rb`:

```
class Body

  attr_accessor :mass, :pos, :vel

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

end
```

Now we can start a new `irb` session by giving the name of a file that will be loaded when `irb` starts up, as follows:

```
|gravity> irb -r body2.rb
irb(main):001:0> b = Body.new
=> #<Body:0x400d4930 @pos=[0, 0, 0], @mass=0, @vel=[0, 0, 0]>
irb(main):002:0> b.mass
=> 0
```

Alice: That is much more convenient, to start a session with the previous knowledge already in place. Let me try something new

```
irb(main):003:0> b.pos[1]
=> 0
```

[1] `body2.rb`

Ah, that works. So you can select a component of a vector and use that directly in a reader function, and presumably also in a writer:

```
irb(main):004:0> b.pos[1] = 0.5
=> 0.5
irb(main):005:0> b
=> #<Body:0x400d4930 @pos=[0, 0.5, 0], @mass=0, @vel=[0, 0, 0]>
```

As expected. And an array index in Ruby obviously starts with a 0, as in C and C++, rather than with a 1, as in Fortran. `b.pos[1]` is the second element of the array, while `b.pos[0]` is the first element.

Bob: I saw you hesitating when you typed line four. I would have thought you would type something like:

```
irb(main):006:0> b.vel = [0.1, 0, 0]
=> [0.1, 0, 0]
irb(main):007:0> b
=> #<Body:0x400d4930 @pos=[0, 0.5, 0], @mass=0, @vel=[0.1, 0, 0]>
irb(main):008:0>
```

which is an alternative but more clumsy way to change the element in an array. When you want to change more than one value, it is of course easier to use array notation:

```
irb(main):008:0> b.vel = [1, 2, 3]
=> [1, 2, 3]
irb(main):009:0> b
=> #<Body:0x400d4930 @pos=[0, 0.5, 0], @mass=0, @vel=[1, 2, 3]>
```

Alice: Yes, you read my mind. I had understood that "`b.pos =`" is parsed by Ruby as an assignment operator "`pos=`" associated with `b` and frankly I did not expect that I could throw in the component selector "`[1]`" without complaints from the interpreter.

Bob: But it did the right thing! This must be what they mean when they say that Ruby is based on the principle of least surprise.

Chapter 3

Pretty Printing

3.1 Tinkering with the Output

Bob: So far we've gotten output as a side effect, with the interpreter echoing the value of everything it evaluates. Let's see whether we can get specific output. Ruby seems to have a general command `p` for printing the contents of any object. Let's see what that gives us:

```
|gravity> irb -r body2.rb
irb(main):001:0>
```

Alice: Before you continue, is there a way to make the `irb` prompt shorter? All this "`irb(main):`" part at the beginning of each line distracts me from the real information. With Ruby being so flexible, I bet it would be easy to customize, and to leave that out.

Bob: As you may expect under a unix system, there is an `irb` configuration file called `.irbrc` and I'm sure we can give you a nicer prompt. What does the manual say? Aha, here is the default definition of the prompt. Let me copy that as a comment in the configuration file, for comparison, followed by the new and shorter version; a `#` sign in Ruby indicates the start of a comment, like `//` in C++. And let me make the shorter version optional, since you may well want to go back to the longer standard version when you get into more complicated situations. So let me create an extra option for invoking `irb`, as `irb --prompt short_prompt`, which calls the new variation of the prompt. Give me a few minutes . . .

. . . Done! Here is our first customized `.irbrc` file:

```
#  .irbrc
```

```
# This is the definition of the default prompt:
#
#    IRB.conf[:PROMPT_MODE][:DEFAULT] = {
#          :PROMPT_I => "%N(%m):%03n:%i> ",
#          :PROMPT_S => "%N(%m):%03n:%i%l ",
#          :PROMPT_C => "%N(%m):%03n:%i* ",
#          :RETURN => "%s\n"
#    }
#
# This short version leaves out the first two parts.
#
# usage: irb --prompt short_prompt

IRB.conf[:PROMPT][:SHORT_PROMPT] = {
      :PROMPT_I => "%03n:%i> ",
      :PROMPT_S => "%03n:%i%l ",
      :PROMPT_C => "%03n:%i* ",
      :RETURN => "%s\n"
}
```

Alice: You're amazing. That would have taken me a few hours to figure out. It looks impressive. But does it work?

Bob: I hope so. Let's start again:

```
|gravity> irb --prompt short_prompt -r body2.rb
001:0>
```

Alice: Now I'm really impressed!

Bob: Your wish is my command.

Alice: Oh yeah? Then let's make *everything* modular.

Bob: Not that wish; don't push it! Where were we? Oh yes, Ruby seems to have a general command `p` for printing the contents of any object. Let's see what that gives us:

```
001:0> b1 = Body.new(0.1, [1.3, 0, 0], [0, 0.5, 0])
#<Body:0x400d11a4 @mass=0.1, @vel=[0, 0.5, 0], @pos=[1.3,0, 0]>
002:0> p b1
#<Body:0x400d11a4 @mass=0.1, @vel=[0, 0.5, 0], @pos=[1.3, 0, 0]>
nil
```

Alice: I guess your `p` command is effectively a dump command that simply dumps the contents of an object. That doesn't get us much further, since `irb` already does that when printing the value of an object. The only difference is that the `p` command itself does not have a value, it returns `nil` after doing its dumping.

Bob: Right you are. Notice by the way that you got even more shortening than you asked for: remember that `irb` used to give us this `=>` at the beginning of each response, presumably to draw our attention with a little arrow? That's gone now too, though I'm not sure why.

Alice: I'm much obliged, but let's not get side tracked. There must be better print commands than the `p` dump method.

Bob: There seems to be a generic `print` command:

```
003:0> print b1
#<Body:0x400d11a4>nil
```

Alice: Are you teasing me by making everything shorter: the prompt, the response, and now even the dump?

Bob: No, I'm only puzzled. What is that little `nil` doing there at the end? Ah, I see. This `print` does not give a new line, unlike `p`. Both methods return `nil`, but with `print` it shows up on the same line.

But my Ruby book told me that `print` is the standard way to get the contents from an object printed out. Let's check the book again.

3.2 A Body-to-String Converter

Bob: Ah, here is a hint. The manual page for `print` states: "Objects that aren't strings will be converted by calling their `to_s` method."

Alice: But our `Body` class does not have a `to_s` method. Does that mean that we have to write one?

Bob: I bet we do. And this book gives us some examples, so it will be easy to adapt that to our purpose. This one here. Aha. Okay, I see how it works. Let me write it directly into our `Body` class definition, in file `body3.rb`:

```
class Body

  attr_accessor :mass, :pos, :vel

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

  def to_s
    "  mass = " + @mass.to_s + "\n" +
```

```
    "   pos = " + @pos.to_s + "\n" +
    "   vel = " + @vel.to_s + "\n"
  end

end
```

Alice: Why the + signs?

Bob: Ruby has overloaded operators. When the plus sign is applied to numbers, it causes an addition, as you would expect. But when a plus sign occurs between two strings, the strings are concatenated, which is the most plausible thing to do when you want to add strings. Principle of least surprise!

Alice: I like that. And the `"\n"` is a new line, just like in the C language?

Bob: You guessed it. Some things never change, it seems. Note that we don't have to use the `mass` method to access the internal variable `@mass`: since our `to_s` method lives inside the `Body` class definition, it automatically has access to its internal variables.

Alice: But what are those three `to_s`'s doing inside the definition of our own `to_s`? Is there something recursive going on?

Bob: In a way, at least in name, though the actual methods are quite different. You see, to convert the value of whole object to a string, we have to convert the values of its internal variables to strings as well. Those three internal variables will quickly become floating point variables, as soon as we initialize them properly, since they represent physical quantities. Fortunately, Ruby knows how to convert floating point variables, or integers for that matter, to strings. The method that does this is called `to_s`, consistent in name with what we are trying to write now.

Alice: This `to_s` seems to be the opposite of `atof` in C; what does `a` that stand for, perhaps ASCII to floating point?

Bob: Yes, but you don't have the reverse, an `ftoa`, in C, let alone a generic version `toa` that can convert any object to ASCII so that it can be printed out. In Ruby you do have such a generic tool, and instead of `a` for ASCII, Ruby uses the more appropriate `s` for string, in naming it `to_s`.

Alice: By generic, you really mean that `to_s` can convert *any* object to a string?

Bob: Yes, but only if you provide the appropriate `to_s` method to the class that defines that object, since Ruby has no way of guessing beforehand how a new object should be printed out. That is our responsibility, and I have just done that. Ruby's task is to sew it altogether seamlessly.

Alice: Which it can do well, because it is designed in such a modular way: there is a nice hierarchy of separate `to_s`'s that are calling each other down the chain of command.

3.3 Taming the Arrays

Bob: the M word again! At some point you have to define what *modular* exactly means. If you call every good idea that works 'modular', you have a tautology. Anyway, I'm curious whether it will come out better now, when we type `print`.

```
|gravity> irb --prompt short_prompt -r body3.rb
001:0> b1 = Body.new(0.1, [1.3, 0, 0], [0, 0.5, 0])
#<Body:0x400d0cf4 @mass=0.1, @vel=[0, 0.5, 0], @pos=[1.3, 0, 0]>
002:0> print b1
  mass = 0.1
   pos = 1.300
   vel = 00.50
nil
```

Alice: It looks nicer all right, but what happened to poor `pos` and `vel`? The mass came out fine though.

Bob: Ah, of course, look, the elements of each array are all concatenated: `[1.3, 0, 0]` has become `1.300`. Remember what I said about strings? In that case addition means concatenation. So all the characters of the array elements are strung together.

Alice: That must be why they call it a string! But how do we separate the fields?

Bob: Guess what, we use a field separator. The book tells me that we can define our own version. Here it is: the method `join` converts an array to a string, and you can give a separator as an argument. Let's try. Here is `body4.rb`:

```
class Body

  attr_accessor :mass, :pos, :vel

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

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

end
```

I'll do the same exercise:

```
|gravity> irb --prompt short_prompt -r body4.rb
001:0> b1 = Body.new(0.1, [1.3, 0, 0], [0, 0.5, 0])
#<Body:0x400d0c7c @mass=0.1, @vel=[0, 0.5, 0], @pos=[1.3, 0, 0]>
002:0> print b1
  mass = 0.1
   pos = 1.3, 0, 0
   vel = 0, 0.5, 0
nil
```

Alice: Well done! But I don't understand why `pos` and `vel` suddenly have acquired this nifty method `join`, which in C++ would be called a member function. Who gave that to them?

Bob: The manual says that `join` is an instance method of the class `Array`. And since `pos` and `vel` are each arrays, they both have access to the `join` method. It was given to them the moment they were created as arrays. So whenever you create an array, you immediately have a whole arsenal of useful methods right at your finger tips. Look, isn't this an impressive list of methods?

Alice: I guess so, but let's see how useful they will turn out to be. You mentioned that `join` is an *instance* method. What would be an example of a *class* method?

Bob: Here they are listed, in the Ruby documentation pages, under the heading `class Array`. There are only two: `[]` and `new`.

Alice: We have seen `new`; you told me that that is a generic method for any class. But what is `[]`?

Bob: Typing `Array.new` gives you an empty array, to which you can add elements afterwards. Typing `Array[](1,a,3)` gives you an array that is already initialized with some elements, as specified in the arguments of this method; in this case it will return the array `[1, a, 3]`.

Alice: I see. If you look at it that way, it makes sense. The only thing different is that there is no dot between `Array` and `[]`.

Bob: However, it would still be a bit cumbersome to initialize an array by having to type the expression `Array[]` each time. There are two better alternatives. You can simply type `Array[1,a,3]`, which gives you the exact same thing. Or you can even type `[1,a,3]`, leaving out the word `Array` altogether, and you still get `[1, a, 3]`.

So the class method `[]` is a versatile and somewhat slippery thing. Yet it really is a class method, and the various ways of invoking it are again forms of syntactic sugar, making your life a lot easier.

Alice: Ruby must be a really sweet language! I guess I'd better study the manual. There must be a lot of tools as well as notations, ready to be used.

3.4 Executing Ruby Files

Bob: So now our Body class has grown up: it can be printed out correctly with print.

Alice: I wonder though, can't we give our Body its own pretty printing method, pp say? In such a way that we can type b1.pp, and it will give us this nicely formatted output?

Bob: Easy. Here is body5.rb. Note that I put a comment in, since otherwise I would never remember what pp meant.

```
class Body

  attr_accessor :mass, :pos, :vel

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

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

  def pp                # pretty print
    print to_s
  end

end
```

Alice: Ah, that's because you are not a Lisp programmer. But you are right. That was very easy. Let's check.

```
|gravity> irb --prompt short_prompt -r body5.rb
001:0> b1 = Body.new(0.1, [1.3, 0, 0], [0, 0.5, 0])
#<Body:0x400d0ad8 @mass=0.1, @vel=[0, 0.5, 0], @pos=[1.3, 0, 0]>
002:0> b1.pp
  mass = 0.1
   pos = 1.3, 0, 0
   vel = 0, 0.5, 0
nil
```

And it works! But I'm getting pretty tired from retyping that whole initialization line for our single body. An interactive interpreter is fine for small jobs, but perhaps it makes more sense to start putting both the class definition and the commands all in one file. We can then keep that file in an editor, and each time we modify either the class or the command, we can just run the file.

Bob: That's an excellent idea. In fact, I was getting tired of creating that whole series of files from `body1.rb` up to `body5.rb`. Let us just take one file and call it simply `test.rb`. Here is the first version, and from now on we can add and change whatever we want. You see, at the end I have included our two commands, the first to create and initialize a particular particle, and the second to print it out.

```
class Body

  attr_accessor :mass, :pos, :vel

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

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

  def pp               # pretty print
    print to_s
  end

end

b1 = Body.new(0.1, [1.3, 0, 0], [0, 0.5, 0])
b1.pp
```

Alice: But how do we run this file, in order to execute those two commands that you added at the end?

Bob: The simplest way is to type `ruby` followed by the file name:

```
|gravity> ruby test.rb
  mass = 0.1
   pos = 1.3, 0, 0
   vel = 0, 0.5, 0
```

Alice: Much easier indeed! No more retyping. And no more `nil` at the end either. Ah, of course. The interactive interpreter echoes the value of each line that you type. But when you execute a file, you only get the results of each command that you give.

Bob: Yes, that's the difference, and that's why this looks cleaner. Of course, for debugging purposes you may want to have an echo of all that passes in front of you, but I'm happy to work with a file, for the time being.

Chapter 4

Input and Output

4.1 Double Precision Input/Output

Alice: So where are we? We have successfully defined a class `Body`, and have learned to give it initial values and to print those out. Shall we code up an integrator?

Bob: Before we do that, I would prefer to get the right I/O tools in place first. On the level of input, we can type in the values by hand when we create a particle, but we have not yet learned how to read particle data from a file. And on the level of output, we really should figure out how to print particle data in 64-bit floating point accuracy, often called "double precision" for historical reasons.

Alice: Why do you want to be so accurate?

Bob: There will be many cases where we integrate the orbit for an N-body system for a while, save the data, and then later continue the integration. If we don't save the data to the same level of accuracy as what is used in the internal calculations, we will lose precision.

Alice: Good point. Let us get our basic tools ready then. Isn't it interesting that learning the I/O always seems to be one of the trickiest parts of learning a new language?

Bob: That must be because all the internal computations take place in a separate universe, defined fully by the language specification. It is only at I/O times that a language is forced to interface with the world. And this necessarily brings in much more baggage.

Most languages specify a simple but rather rigid way to do I/O, as a default. But sooner or later you need more control, for particular purposes, and that requires more complicated I/O routines, with a more complicated syntax.

In other words, doing your own thing is a lot easier than interfacing with the world.

Alice: I'm glad to see you stressing interface issues! Everything that you said about I/O applies to the way I think the various modules in an N-body code should communicate. For each module, there is a well controlled internal environment, that has to interface with a wild world out there, over which the module has no control, and from which the module has to protect itself, so that it does not get tripped by confusing signals

Bob: You always have an excuse to talk about principles, just when we're about to do some real work. Let's get our double precision first. And because we are writing a toy model, I suggest we do our I/O in ASCII, not in binary form.

Alice: Absolutely. I am always happier when I can look into a file, and see the data there directly, without everything being encoded in some weird way. Such encryptions are often machine dependent anyway, and ASCII characters are platform independent.

But in that case, we'd better print everything out with enough digits. Let's look at the web. The IEEE 64-bit floating point number definition specifies 1 bit for the sign, 11 bits for the exponent, and the remaining 52 bits for the mantissa. 2^{52} is less than 5.10^{15}, so 16 digits should be enough.

Bob: No, there are effectively 53 digits. By definition of floating point notation, the most significant digit of the mantissa is always 1, and cleverly the IEEE people have defined their standard so that that 1 is being left out. Effectively we are using 65-bit precision in double precision calculations on our computers.

Alice: I had no idea! But 2^{53} is about 9.10^{15}, so 16 digits is still enough. I was lucky there.

4.2 A Simple Version

Bob: I should be able to figure out how to do this. I like this kind of tinkering.

Alice: I'm glad you do! I'll get a cup of tea in the meantime.

. . .

Here's a cup for you too. Ah, that looks impressively short. Is that really enough to do the job?

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

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

Bob: Yes, it works! But let me first show you what it all means. The `printf` statement in the second line comes straight from C. It specifies that the mass will be printed in a field that is 24 characters wide, with 16 decimal places of precision.

The third line shows a nice Ruby feature: each array knows how long it is, so instead of stepping through the array with a do loop or for loop, you can simply ask an array to do something for each of its elements. `pos.each` invokes the method `each` that is built in already within the class `Array`. It is called an iterator.

Alice: I presume it corresponds to the use of iterators in the C++ standard template library. But it looks quite a bit simpler. Instead of first asking the array for its begin and end, and then looping over the elements in between, here the one word `each` does the whole job.

Bob: Welcome to Ruby! What the iterator does to each element is specified in the region between the parentheses, immediately following `each`. First comes a variable name `x` between bars, that stands for the name of each element of the array, while the array is traversed. In this case, each element is printed in the same format as we printed the mass in the previous line.

Alice: The `simple_read` method puzzles me. What is this `gets` in the first line?

Bob: It is short for 'get string': it reads in one line of input, and returns as its value the string of characters that it has read in.

Alice: Ah, and `to_f` converts the string, as a collection of characters, to a floating point value, just as `atof` would do in C.

Bob: Exactly. And the important thing to notice here is that `gets` returns a string, which in Ruby is an instance of the class `String`. And this `to_f` is a method associated with that class – it plays the role that a member function plays in C++, as we saw before. You can ask an object to invoke one of its methods by putting a dot in between the object and a method. And by implication this is what happens to the object returned by `gets`.

Alice: The principle of least surprise all right.

Bob: As you can see in `simple_print`, I have chosen a data format in which a single particle prints its mass, position, and velocity on three consecutive lines.

To be compatible, `simple_read` should perform a separate `gets` for `@mass`, for `@mpos`, and for `@vel`.

The trickiest thing to get right was to read the three components from a vector. Here I have used `split` which is a string method that splits the string into smaller chunks, and then returns an array which contains each chunk as an element. If you want, you can specify the way a string is being split into such subarrays, by giving a parameter to `split`, but the default delimiter is white space.

Now `map` is an array method that takes each element of the array in turn, and executes a block of code for that element. The syntax is the same as what we saw for `each` above. The block is delimited by curly braces, and the free variable that loops over the array elements is specified by the vertical bars. The rest of the block contains the commands that are executed for each element. In our case each component of the position is converted into a floating point number, and the same happens for the velocity on the next line.

By the way, `map` is actually an alias for `collect`, and you can use both words interchangeably for the same method. Because this method effectively 'maps' an action onto each element and then 'collects' the results into a new array, each word describes part of the process. I prefer `map` both because it is shorter and because it describes the step where the actual work is done. In Ruby, there are many examples of such aliases. To find the length of an array `a`, for example, you can equally well use `a.length` as `a.size`. Sometimes you even have freedom in spelling: `a.indexes` and `a.indices` do the exact same thing. I consider all this freedom another friendly aspect of Ruby.

Alice: The only drawback is that when you look at someone else's program, you might be surprised to see someone using unfamiliar aliases. However, I presume that you get used to that pretty quickly.

What I am curious about is that you haven't specified anywhere that we live in three dimensions. I find that pretty remarkable: in Fortran or C or C++, you could not get away with that. This must mean that the code will work equally well for a two-dimensional simulation, where a body has position and velocity vectors with only two components, as for a three-dimensional simulation.

Bob: Right you are, and that indeed gives you a wonderful flexibility. In C you would first define `#define NDIM 3`, and then specify something like `for (k = 0, k < NDIM, k++)` followed by the block of code you would loop over, containing expressions like `pos[k]` for the kth element of the position vector. What a breeze this is, by comparison!

4.3 Trying It Out

Alice: Remarkable. Once you gain familiarity with those notions such as `gets` and `split` and `map`, it must become second nature to string a few together. The result is a notation that is compact, yet still informative. Can you show me that all this really works?

Bob: Here is what I wrote in our test file `test.rb`, immediately following the `Body` class definition:

```
b = Body.new()

b.simple_read

b.simple_print
```

As you can see, I create a new blank Body, and I immediately perform an input. This means that we have to put in the values by hand, on three consecutive lines, one for each internal variable. The script then performs an output, printing out the particle state. Here is an example.

```
|gravity> ruby test.rb
3
0.1 0.2 0.3
4 5 6
  3.0000000000000000e+00
  1.0000000000000001e-01  2.0000000000000001e-01  2.9999999999999999e-01
  4.0000000000000000e+00  5.0000000000000000e+00  6.0000000000000000e+00
```

Alice: Very good. And since we can now read and write, how about testing your script by reading what you just wrote out? We can pipe the output into another invocation of `test.rb`.

Bob: My pleasure:

```
|gravity> ruby test.rb | ruby test.rb
3
0.1 0.2 0.3
4 5 6
  3.0000000000000000e+00
  1.0000000000000001e-01  2.0000000000000001e-01  2.9999999999999999e-01
  4.0000000000000000e+00  5.0000000000000000e+00  6.0000000000000000e+00
```

Alice: Congratulations! This is what a mathematician would call a fixed point, if we would view the operation `ruby test.rb` as a mapping.

Actually, this would be an appropriate way to view this script: when we finish our integrator, it will transform initial conditions to final conditions after a certain time t. In other words, the integrator will act as a propagator, mapping initial conditions onto final conditions.

Bob: I'm glad you haven't forgotten the goal of these exercises, to move particles around. The term 'propagator' comes from elementary particle physics, I presume? Well, I guess that our point particles are about as elementary as they come, so it is not altogether inappropriate.

Alice: Before we get going with our particle pushing, I can't help wondering whether you can't further simplify your reading and writing routines. I bet these were not what you first wrote down; you must have compactified things already.

Bob: You're right. My first attempt at the read method was a lot longer. Here, I still have it:

```
def read
  s = gets
  @mass = s.to_f
  s = gets
  a = s.split
  a.map! { |x| x.to_f }
  @pos = a
  s = gets
  a = s.split
  a.map! { |x| x.to_f }
  @vel = a
end
```

Alice: Quite a reduction. But why the exclamation marks after `map` ?

Bob: In Ruby there is a general convention that a command followed by an exclamation mark in its name does the same thing as the command without that exclamation mark, but it does it to the object it is associated with. Here `map!`operates on the array `a` that is calling the method. Previously, I used `map` which returns a new array that contains the values resulting from the operations. Be careful here: the bang sign "!" is *not* an operator in itself, it is only an allowed character for the last part of the name of a command. It is up to the code writer to choose sensible names such that `do_something` and `do_something!` do the same thing, the first one producing a new array, and the second one changing the elements of the array on which it was called.

4.4 A Surprise

Alice: So you managed to reduce the number of lines of your method by a factor of three, presumably also in your write method. Currently you have three lines left in each method. Can I challenge you to reduce it even further?

Bob: By another factor of three? But that would only leave one line for each method. Are you serious?

Alice: Well, not really. But I'm still thirsty. Let me get us some more tea.

. . .

Here's another cup. WHAT? Are you serious? Does that work???

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

```
  def simple_read
    (@mass,),@pos,@vel = (1..3).map{|i| i = gets.split.map{|x| x.to_f}}
  end
```

Bob: Here is the keyboard. Try it.

```
|gravity> ruby test.rb
8
0.2 2 20
10 1 0.5
  8.0000000000000000e+00
  2.0000000000000001e-01  2.0000000000000000e+00  2.0000000000000000e+01
  1.0000000000000000e+01  1.0000000000000000e+00  5.0000000000000000e-01
```

Alice: I'm shocked. How can you do input in one line and output in one line for a whole particle, with a heterogeneous data set? Even writing this question down would take more than one line of text!

Bob: I must admit, I surprised myself. But then again, you asked, and I like to take up a challenge. Would you like to know how/why it works?

Alice: Hmmm. Perhaps not in detail yet. Since I was going to look at the manual tonight, this will give me my own challenge: to figure this one out.

Chapter 5

The Two-Body Problem

5.1 Equations of Motion

Alice: That was great, to be able to do such rapid prototyping in a language we hardly knew. I can see the advantages of an interpreted, dynamically typed computer language. If we had tried to do this in C++, it would have taken quite a bit more time, and we would have had to write many more lines of code.

Bob: Yes. Defining a class, getting it to behave, providing I/O, and chaining it together by piping data from one program to another, all that is a nontrivial beginning. This is encouraging! Let's move on, to see how much we have to add before we can let the integrator integrate.

Alice: Not much, I think, at least if we start with the simplest integrator possible, forward Euler, for the simplest N-body system, namely to solve the 2-body problem.

Bob: The other day I was asked by a friend outside astronomy why we call it the N-body problem. What was the problem with those bodies, she asked. I had to think for a moment, because we are just in the habit to talk about it that way.

Alice: We talk about solving the equations of motion; normally when you solve something, you solve a problem. I guess the terminology comes from seeing a differential equation as posing a problem, that you then solve, either analytically or numerically. Newton's equations of motion for N bodies form a system of N differential equations, and the challenge to obtain solutions is called the N-body problem.

I must admit, I don't like the term "N-body" at all. Why N, and not some other symbol? I would much rather use the term "many body", as they do in solid state physics, say. Don't you agree that the "gravitational many-body problem" sounds much more elegant than the "gravitational N-body problem"?

Bob: I really don't care, as long as I can solve it. Okay, let's start with the 2-body problem, which is really a 1-body problem of course, since you only have to solve the relative motion between the two particles.

Alice: You say "of course", but if this is going to be a presentation for students, we'd better be more explicit. We should start with Newton's equations of motion for a gravitational many-body system:

$$\frac{d^2}{dt^2}\mathbf{r}_i = G\sum_{\substack{j=1\\j\neq i}}^{N} M_j \frac{\mathbf{r}_j - \mathbf{r}_i}{|\mathbf{r}_j - \mathbf{r}_i|^3} \tag{5.1}$$

Here M_j and $\mathbf{r}_j$ are the mass and position vector of particle j, and G is the gravitational constant. To bring out the inverse square nature of gravity, we can define $\mathbf{r}_{ji} = \mathbf{r}_j - \mathbf{r}_i$, with $r_{ji} = |\mathbf{r}_{ji}|$, and unit vector $\hat{\mathbf{r}}_{ji} = \mathbf{r}_{ji}/r_{ji}$. The gravitational acceleration on particle i then becomes:

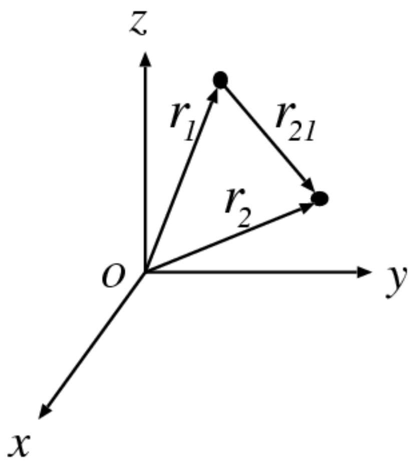
$$\mathbf{a}_i = G\sum_{\substack{j=1\\j\neq i}}^{N} \frac{M_j}{r_{ji}^2}\,\hat{\mathbf{r}}_{ji} \tag{5.2}$$

5.2 Relative Motion

For a 2-body system, everything simplifies a lot. Instead of dealing with position $\mathbf{r}_1$ for the first particle and $\mathbf{r}_2$ for the second particle, we can write the above system of equations of motion as a single equation instead, in terms of the relative position, defined as:

$$\mathbf{r} = \mathbf{r}_2 - \mathbf{r}_1 \tag{5.3}$$

This can be visualized as a vector pointing from particle 1 to particle 2, in other words with its head at the position of particle 2, and its tail at the position of particle 1.



Introducing $r = |\mathbf{r}_2 - \mathbf{r}_1|$, we then get:

$$\begin{aligned}\frac{d^2}{dt^2}\mathbf{r} &= \frac{d^2}{dt^2}\mathbf{r}_2 - \frac{d^2}{dt^2}\mathbf{r}_1 \\ &= GM_1\frac{\mathbf{r}_1 - \mathbf{r}_2}{|\mathbf{r}_1 - \mathbf{r}_2|^3} - GM_2\frac{\mathbf{r}_2 - \mathbf{r}_1}{|\mathbf{r}_2 - \mathbf{r}_1|^3} \\ &= -G\frac{M_1 + M_2}{r^3}\mathbf{r}\end{aligned}$$

We can choose physical units for mass, length, and time in such a way that $G = 1$. With the shorter notation $M = M_1 + M_2$, we then get

$$\frac{d^2}{dt^2}\mathbf{r} = -\frac{M}{r^3}\mathbf{r} \tag{5.4}$$

You often see an even more abbreviated 'dot' notion for time derivatives. If we place a dot on top of a variable for each time derivative we take, we wind up with

$$\ddot{\mathbf{r}} = -\frac{M}{r^3}\mathbf{r} \tag{5.5}$$

Bob: End of lecture. Yes, let's start solving that last equation. Here is how the forward Euler integration scheme works:

$$\begin{aligned}\mathbf{r}_{i+1} &= \mathbf{r}_i + \mathbf{v}_i dt \\ \mathbf{v}_{i+1} &= \mathbf{v}_i + \mathbf{a}_i dt\end{aligned}$$

for the position **r**, the velocity **v**, and the acceleration **a** of our single particle, that describe aspects of the relative motion between two particles. The index i indicates the values for time t_i and $i+1$ for the time t_{i+1} after one more time step has been taken: $dt = t_{i+1} - t_i$.

Of course, we have to provide intial conditions for the position and velocity vectors, before we can solve the equations, and we will do that as soon as we have a working integrator. For now, all we have to do is to code up these equations.

Alice: That was an even shorter lecture. Yes, let's do it.

5.3 Modularity

Bob: Shall we type the code in the same file `test.rb` where we put the `Body` class? We can still use the `Body` format for our 'relative' particle, as long as we remember that the mass of that particle corresponds to the sum of the masses of the original particle.

Alice: We can certainly use the `Body` class, but I suggest that we put the definition of the `Body` class in a file `body.rb`, and only the actual integrator in our file `test.rb`.

Bob: A modular approach, I take it?

Alice: Sure, whenever I can get away with it!

Bob: It may not be a bad idea, in this case. And to show you how modular I can be, I will even call my first attempt `body0.rb`. When we create other versions, we can give them higher numbers, and when we are really happy, we can call the final version `body.rb`.

Alice: That sort-of goes in a modular direction. It all depends on what you do with that stack of versions. But I appreciate your attempt!

Bob: Coming back to your suggestion, yes, we can put the `Body` class definition in a separate file. This is similar to what you do in C and C++ with an include file – only here it is easier to do so.

Alice I remember well those constructs in C where you had to write things like `#ifndef` *this* and `#ifndef` *that* before you could be sure that it was safe to include a file without including it more than once? You're saying that Ruby makes life easier?

Bob: Yes, in Ruby you can use a construct called `require` *"filename"*: it only includes the file if it hasn't been included yet, directly or indirectly.

Here is the file `body0.rb`:

```
class Body
```

```
  attr_accessor :mass, :pos, :vel

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

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

  def pp               # pretty print
    print to_s
  end

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

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

end
```

It is just as we left it, but without my one-liner I/O hacks. Now give me some time to figure out how to implement the forward Euler idea . . . .

5.4 The First Integrator

. . . Here it is, the new version of `test.rb`. As you can see, it starts with requiring that `body0.rb` gets included at the top.

```
require "body0.rb"

include Math
```

```
dt = 0.01            # time step size
ns = 100             # number of time steps

b = Body.new
b.simple_read

ns.times do
  r2 = 0
  b.pos.each {|p| r2 += p*p}
  r3 = r2 * sqrt(r2)
  acc = b.pos.map { |x| -b.mass * x/r3 }
  b.pos.each_index { |k| b.pos[k] += b.vel[k] * dt }
  b.vel.each_index { |k| b.vel[k] += acc[k] * dt }
end

b.pp
```

Alice: Short and simple, but it looks simple only because we are used to coding up integrators. For the students, it would be good to spell out, just one time, what the equations above look like, in the simplest case where we are working with vectors in two dimensions. To be specific in our notation, we can use subscript x for the first component of each vector and subscript y for the second component, as follows:

$$\begin{aligned}
\mathbf{r} &= \{r_x, r_y\} \\
\mathbf{v} = \dot{\mathbf{r}} &= \{v_x, v_y\} \\
\mathbf{a} = \ddot{\mathbf{r}} &= \{a_x, a_y\}
\end{aligned}$$

The last three equations in the earlier sections then become six equations, one for each vector component.

There are two equations for the calculation of the relative acceleration components a_x and a_y, each of which can be calculated in terms of the relative position components r_x and r_y, since $r = \sqrt{r_x^2 + r_y^2}$:

$$\begin{aligned}
a_x &= -\frac{M}{(r_x^2 + r_y^2)^{3/2}} r_x \\
a_y &= -\frac{M}{(r_x^2 + r_y^2)^{3/2}} r_y
\end{aligned}$$

Two equations tell us how to find the position vector components at the new time step: we take the components at the previous time, and we add the increment in position provided by the product of the corresponding velocity vector component and the time step size:

$$
\begin{aligned}
(r_x)_{i+1} &= (r_x)_i + (v_x)_i dt \\
(r_y)_{i+1} &= (r_y)_i + (v_y)_i dt
\end{aligned}
$$

Similarly, we update the velocity vector components, by incrementing those using the acceleration, which is the time derivative of the velocity:

$$
\begin{aligned}
(v_x)_{i+1} &= (v_x)_i + (a_x)_i dt \\
(v_y)_{i+1} &= (v_y)_i + (a_y)_i dt
\end{aligned}
$$

If we would do this calculation in three-dimensional space, all vectors would acquire a third component: $\mathbf{r} = \{r_x, r_y, r_z\}$, and so on. Each pair of equations above would then be replaced by a triple of equations, because an extra equation would appear for the corresponding z component.

To conclude the story, the forward Euler algorithm for the two-body problem can be summarized as follows. Given the relative position and the relative velocity between the two particles at a given time t_i, first calculate the relative acceleration in terms of the relative position at t_i. Then compute the new values for the relative positions at t_{i+1}, in terms of the relative positions and velocities at t_i. Similarly, compute the new values for the relative velocities at t_{i+1}, in terms of the relative velocities and accelerations at t_i.

Bob: And after you have once said that in words, it becomes obvious how much more compact and efficient equations are in component notation. And then it is clear that vector notation is even more efficient than component notation.

5.5 Writing Clean Code

Alice: But now I want to understand how exactly you coded these six equations in the file `test.rb` above.

Bob: The line `include Math` tells Ruby to make the `Math` module visible to the program. This is an example of what is called a mixin.

However, this is merely a convenience. We could have left that out, but then the square root `sqrt` would not have been recognized; we would have had to write `Math.sqrt` instead. It is just more convenient to mixin `Math` right at the beginning.

Alice: Then you define the size of the time step and the number of steps you want to take. It is a good idea, to introduce them as variables at the top of your program, instead of hard coding them below inside the loops. It will be much easier to change the values only once at the top, rather than having to inspect the whole program to see where the values occur.

Bob: You would be surprised how many scientists of the hard-coding type you can still find!

Alice: That is because most scientists have never been taught how to include levels of data abstraction in their program, right from the beginning in the conceptual view of the architecture of a program. One of the main principles of data abstraction is: no constants except 0 and 1 are allowed in the body of the code. Everything else should be given a symbolic name.

Bob: I'm in principle against principles, but I like the content of what you just said. As for those many scientists, at least they all use subroutines.

Alice: Not all! There are still legacy codes being used widely that jump around through many pages of programs using do loops between blocks of code that are effectively a poor man's subroutine. One problem is that scientists view subroutines merely as a convenient way to save time: if you use the same piece of code in three places in your program, you save time by writing it only once as a subroutine, and calling it from three places. But that is the *least* important reason to use functions or subroutines in a program.

A much more important reason is the issue of code maintenance. If the same piece of code occurs in more than one place, it becomes essentially impossible to maintain the code in a consistent way. Change something in one place in someone's legacy code, and most likely you don't even know that it would have to be changed in the other place to. Bugs appear, for no apparent reason, while you are *sure* that you did the right thing; you just didn't know about the other place which has now become incompatible. I bet that literally tens of person-millennia have gone down the drain this way, during the last fifty years, chasing those bugs.

Bob: You mean more than 10,000 person-years?

Alice: I wouldn't be surprised. I would estimate there to be at least a hundred thousand programmers in the world. Most of them have struggled for a total of a month in their life, at the very least, chasing bugs that originated in codes which they tried to update, only to find out that there were unexpected and typically undocumented side effects. That already makes a person-millennium worth of effort. And I think that is a vast underestimate.

Bob: Impressive. I never thought about it quite that way.

Alice: So the name of the game is modularity, and I'm glad to see you giving the students the right example, in the third and fourth line of your integrator.

Bob: This type of what you call modularity at least I agree with. It would have been easy to write `100.times` in the seventh line instead of `ns.times`, and change that number by hand later. I thought about doing that. But when I saw that `dt` occurs not once but twice in the body of the `do` loop, I realized that it would be all to easy to change the numerical value of one of them, and not the other.

Alice: With the result of having the position and velocity stepping forward in time at different speeds – quite a nightmare, when you want to debug it. Most likely you will start your debugging on the assumption that you made a mistake in the physics of gravitational interaction, or in the mathematical equations, or in the way you solve them numerically, or just a typo in the way you coded it all. The last thing you suspect would be that you would update the position with time step 0.01 and the velocity with time step 0.001, say.

Bob: I know it all too well, from past experience. That's why I've wizened up.

5.6 Where the Work is Done

Alice: Talking about the `do` loop, I presume it is being traversed `ns` times, almost as if you read Ruby like English. How can that possibly work?

Bob: The period means that `times` is a method associated with the class of which the variable `ns` points to an instance. Since `ns` has been initialized with an integer, it now has become an instance of class Integer. And conveniently, this class has a method `times` built in that takes the value of the instance of the class, here the value of `ns`, and iterates the following block of code `ns` times.

As often is the case in Ruby, the explanation of a construction sounds far more complicated than the way you use it. As you already remarked, `ns.times` reads like English. Another example of the principle of least surprise.

Alice: Within the `do` loop, you first compute $|\mathbf{r}|^2$ by introducing a variable `r2`, initialize it to zero, and then accumulating the result of squaring the value of each component of the vector **r**. What you need for the acceleration is the $3/2$ power, so you compute that in the next line.

Finally, you solve the three pairs of equations I wrote above. The array method `each_index` presumably does what it says, it executes the next block of code once for each possible value of the index of the array?

Bob: Yes. In the case of a two-dimensional array `a`, the two components are `a[0]` and `a[1]`; remember that Ruby arrays start at zero by default. For such an array, the block of code following `a.each_index` will be executed once for `k=0`, and once for `k=1`.

Alice: And if we would have used three-dimensional arrays for positions and velocities and accelerations, the block would be executed also for `k=2`. How neat to see an integrator without any need to remind the computer *ad nauseam* to go through an inner loop for `k` is 1 to 3, or something like that – just as we saw earlier when we wrote the I/O routines.

Bob: But we can do even better. Granted, we have avoided the use of an explicit variable `NDIM` for the number of dimensions, but there is still the lingering smell of components hanging in the air, in terms of the use of `k` in the last two blocks of code. I wanted to get the code working quickly, so I didn't think about it to much, but I have an idea of how to get rid of even `k`.

Alice: That would be even better. But shall we first check whether everything works as advertised?

Chapter 6

Orbit Integration

6.1 Two Dimensions

Bob: Time to test our Forward Euler code!

Alice: I see there is a call to `simple_read`, so I presume we have to provide an input file, that will contain the initial conditions, from which we can then start to integrate the equations of motion.

Bob: Let's call it `euler.in`; how about this one?

```
1
1 0
0 0.5
```

Alice: A nice example of our dimensional freedom: a two-body problem is intrinsically two-dimensional. It was not just laziness that I showed the equations in component form only for two dimensions. Even in three dimensions, a 2-body problem can always be reduced to a two-dimensional problem. The point is that you can always find a plane in which the relative motion takes place.

Bob: That may not be immediately obvious for a student. When I heard this for the first time, I thought about two particles passing each other at right angles at a distance, like the two arms of a cross but then offset in the third dimension.

Alice: You are right. The best way to convince a student is probably to let her or him do the exercise of translating the motion to the center of mass system of the two particles. In that system, the motion of the one particle is the same but opposite as the motion of the other particle, apart from a scaling factor involving the ratios of the two masses. The position vector of one particle as defined from the center of mass, together with the velocity vector of that particle, spans a unique plane. The fact that the other particle moves in the (scaled) opposite way then implies that the other particle moves in the same plane as well.

Bob: Yes, even if you know the answer, it always requires some thinking to reconstruct the reason for the answer. So because a two-body problem is inherently two-dimensional, we might as well start with specifying only two components for the position and velocity of the relative motion of the two particles, which made me choose the above numbers. Total mass of unity, initial position on the x-axis, also unity, and initial velocity perpendicular to that, and one half, for a change.

6.2 Forward

Bob: Time to test our Forward Euler code:

```
|gravity> ruby test.rb < euler.in
  mass = 1.0
   pos = 0.443229564341409, 0.384215558686636
   vel = -1.29436531289392, 0.0258120006669036
```

Alice: That looks reasonable. We started with a relative velocity of 0.5, so in one time unit of integration, you would expect the position vector to have changed by something like half a length unit. We started with position {1.00, 0.00} and we now have {0.44, 0.38}. So we have crossed a distance vector of {-0.56, 0.38}, which has a length of $\sqrt{0.56^2 + 0.38^2} = 0.68$, not far from one half.

Bob: And look, the velocity has increased. It started off with vector length unity, and now it is about 1.3 length. Since we started with a velocity direction perpendicular to the line connecting both particles, we were either at pericenter or apocenter.

Alice: You'll have to explain that to the students.

Bob: Pericenter is the point in a Kepler orbit where the distance between two particles is smaller than anywhere else in that orbit. The apocenter is reached when the two particles are furthest away from each other. From the Greek *peri* or near, and *apo* or away. Now if you are closest, you move fastest, since the gravitational force is larger at smaller distances. Similarly, when you are at apocenter, you move slowest. Moving away from pericenter, you slow down, while moving away from apocenter you speed up. Ergo: we obviously started the particles at the maximum distance, given their energy and angular momentum, specified at the start.

But all this will makes much more sense to the students when we show them the orbit with some real graphics. We'll come to that soon. For now, let's make sure that the integrator does what it is supposed to do. Let me make the step size ten times smaller, to see whether we get roughly the same answer. I will make the number of time steps ten times larger, to make sure the integration time stays the same:

```
dt = 0.001            # time step size
ns = 1000             # number of time steps
```

And then we run it:

```
|gravity> ruby test.rb < euler.in
  mass = 1.0
   pos = 0.433021780531577, 0.37860453335417
   vel = -1.31479264498764, 0.00718433856141339
```

Alice: Not bad, for such a simple integrator. The differences with the previous run are typically only one unit in the second decimal place, in each vector component.

Bob: That's about all that I would have expected from such a simple integrator.

6.3 Error Scaling

Alice: How do we expect things to scale when we will make the time step even smaller? We are dealing with a first order integrator. How did that go?

Bob: A first order integrator is first-order accurate. which means that the errors per time step are quadratic in the size of the time step. Making the time step ten times as small will give you errors that are each one hundred times as small. However, you have ten times more steps, so if the errors are systematic, as they often are, the total error is ten times larger than a single error per time step. In other words, making the time step ten times smaller will make the total error will go down also by a factor of ten.

Or so I think. Let me put in the values

```
dt = 0.0001            # time step size
ns = 10000             # number of time steps
```

and run it:

```
|gravity> ruby test.rb < euler.in
  mass = 1.0
   pos = 0.431974652285879, 0.378023068595097
   vel = -1.31693301875844, 0.00522927166213894
```

Alice: How nice. The differences between the last two runs are now mostly one unit in the third decimal place. Just as you predicted!

Bob: Let me push my luck with another factor of ten

```
dt = 0.00001           # time step size
ns = 100000            # number of time steps
```

and see whether the trend continues:

```
|gravity> ruby test.rb < euler.in
  mass = 1.0
   pos = 0.431869664380511, 0.377964708368075
   vel = -1.31714808792365, 0.00503278405037421
```

It does. Differences of one unit in the fourth decimal place. And most of the differences in comparing the last two runs must come from the much less accurate run before the last one. So the last one can be expected to be accurate to about one unit in the fifth decimal place. So it would seem safe to say that the first component of the relative position will be 0.43187 ± 0.00001. I think we have shown that we can solve the 2-body problem with Ruby!

6.4 A Full Orbit

Alice: Before we declare victory, I'd like to see us integrating at least a full orbit. How about increasing the time from 1 to 10 time units? That should be more than enough.

Bob: I would think so. With a total mass of one and an initial distance of one, in a system of units where the gravitational constant $G = 1$ as well, the orbital period should be of order unity. But there must be a factor π somewhere in there as well. My guess would be that the period would be at least five, in our units, given that the relative motion started on the right hand side of the x axis moving upward, and after one time unit we are still in the first quadrant, with positive y and x values.

Alice: Something like that, but it could be a bit smaller. We started at apocenter, which means that the relative motion is speeding up. In fact, we saw before that the velocity had increased. Anyway, pretty soon we'll have to install some form of graphics, since I'd sure like to see the orbit, rather than staring at numbers. But one thing at a time; we'll get to that soon.

Bob: Here is an integration for ten time units, starting with our original time step of 0.01:

```
dt = 0.01            # time step size
ns = 1000             # number of time steps
```

For one time unit we got a position error of order one percent, for a time step of 0.01. For ten time units we may expect ten or twenty percent, I guess. But in order to get an estimate of our errors we have to do at least two runs, to compare. So we'll do this run first, and then we'll do a run with much smaller time steps. Here is the first run.

```
|gravity> ruby test.rb < euler.in
  mass = 1.0
   pos = 7.6937453936572, -6.27772005661599
   vel = 0.812206830641815, -0.574200201239989
```

Alice: Hmmm. Whatever the second run will be, the error must be quite a bit more than twenty percent, I'd say. After you explained that the initial position was at apocenter, we should never encounter a situation where the particles are at a relative distance of more than unity. And here we have a distance of more than seven, in fact almost a distance of ten units!

Bob: With expected errors of a few tens of percent it is easy to get into a nonlinear regime. And come to think of it, our relative particle motion was headed for pericenter, where the distance between the particles gets smaller, the inverse square law force increases even faster, and the curvature of the orbit becomes larger as well. And since we have constant time steps, the errors per time step can be expected to be quite a bit larger.

Alice: How much larger?

Bob: I'm sure we can derive that easily, but let's be lazy and ask the computer. Later we can be more systematic about all this, when we start implementing higher order integrators. So let's go down with powers of ten again in time steps:

```
dt = 0.001            # time step size
ns = 10000             # number of time steps
```

I hope the relative motion behaves better now.

```
|gravity> ruby test.rb < euler.in
  mass = 1.0
   pos = 2.01435512882368, 0.162565336385657
   vel = -0.152875528688122, 0.258696442895482
```

6.5 Taking Time

Alice: Better already. Still too far, at a relative distance of more than two, but not as outrageous as before. Another step of ten down?

Bob: My pleasure:

```
dt = 0.0001            # time step size
ns = 100000             # number of time steps
```

```
|gravity> ruby test.rb < euler.in
  mass = 1.0
   pos = 0.292716737826681, 0.38290774857968
   vel = -1.56551896976999, -0.313957063867402
```

Alice: Ah. Much better. This begins to be believable.

Bob: But unless we really converge, I for one won't believe it quite yet.

```
dt = 0.00001            # time step size
ns = 1000000             # number of time steps
```

```
|gravity> ruby test.rb < euler.in
  mass = 1.0
   pos = 0.519970642634004, -0.376817992041834
   vel = 1.17126787143698, 0.114700879739653
```

Ah, perhaps we are getting there. Now we have a difference between runs that is closer to what I had hoped for, a few tenths of percent.

Alice: If we are really converging, with a first-order integrator, we should see the errors shrinking by a factor of ten.

Bob: Let's check. But the computer sure took its time to give us that last result. Something tells me that Ruby is not very fast at numerical calculations

```
dt = 0.000001            # time step size
ns = 10000000             # number of time steps
```

This may take a while, ten times longer than the previous run!

Alice: You're right. With a C or Fortran code, doing a million time steps should take much less than a second. There must be a huge efficiency factor involved.

Bob: Perhaps not too surprising, given that Ruby is an interpreted language. And besides, the fact that it allows such short and powerful expression must mean that a lot is going on behind the scenes. One of the books I looked at mentioned that almost everything in Ruby is done not just with one level of redirection, as you might expect, but with two levels of redirection!

Alice: That makes you wonder about the wisdom of choosing Ruby for a computationally intensive project.

Bob: That's what I thought all along, but . . .

Alice: . . . you mean, it's no problem for a student project, as a toy model?

Bob: No, it *is* a problem, even for a student project. Which student would want to have to drink a cup of coffee before he or she can integrate a few orbits in a Kepler problem? Ruby must be at least two orders of magnitude slower than C or Fortran.

6.6 Convergence

Alice: Does this mean that we have to abandon our pretty new language?

Bob: Not at all! What I was about to say is: I was worried about speed, at first, but then I read on the net that it is quite easy to speed Ruby programs up, simply by replacing the most compute-intensive part of the code by a small C program, which then gets called from Ruby. The way it was described was rather simple, and I expect that we can regain most of the speed we now have lost.

Alice: I sure hope so. I'm quickly growing fond of Ruby. As soon as we get a chance, let's test it for ourselves.

Bob: Will do! Hey, we finally got an answer:

```
|gravity> time ruby test.rb < euler.in
  mass = 1.0
  pos = 0.592168165567474, -0.362592197667279
  vel = 1.04418312945112, 0.205156629088709
```

Alice: And you were right: it is converging.

Bob: But I was wrong in thinking that the error would be only a few percent. It is still about ten percent in the x position, and even more in the y component of the velocity. I want to get to the bottom of this. Let us send Ruby out for a long errand, with one hundred million time steps:

```
dt = 0.0000001            # time step size
ns = 100000000             # number of time steps
```

We may as well get a bit to eat, since we can't wait for this to finish.

Alice: Good idea.

```
. . . .

|gravity> time ruby test.rb < euler.in
  mass = 1.0
  pos = 0.598877592948656, -0.360833442654858
  vel = 1.03213853108295, 0.213031665991379
```

Bob: That was not bad, to stretch our legs and get some good food. A nice side effect of a slow integration.

Alice: And look, Ruby has done it's job. Now the error is down to about one percent in any of the position or velocity coordinates.

Bob: Good! Now I believe that we have done things correctly. But boy, that took a long time to converge. Two morals of the story: to do anything at all in orbit calculations, you must use a higher-order integrator, and you'd better speed up its inner loop if you are using an interpreted language.

Chapter 7

Debugging

7.1 A Vector Class

Bob: Now that we can integrate, let me come back to these lines in the code that you were not so happy with, containing a `k` for the components, remember?

Alice: Yes, do you have a way of avoiding any component notation, even on the code level?

Bob: I think I do. In a way, it is pure syntactic sugar, but then again, you may say that a compiler or interpreter is one giant heap of syntactic sugar on top of the machine language. If it makes the code much easier to read, I'm happy with it.

The trick must be to make sure that you can add vectors the way you expect to add them in physics.

Alice: But according to the principle of least surprise, I would have guessed that Ruby would be able to add two vectors, in array notation, at least if they contain numerical entrees, no?

Bob: One person's expectation is another person's surprise! Not everyone approaches arrays with your physicist's view of the world. Here, this is a good question for which to go back to `irb`.

```
|gravity> irb --prompt short_prompt
001:0> a = [1, 2]
[1, 2]
002:0> b = [3, 3]
[3, 3]
003:0> a + b
[1, 2, 3, 3]
```

Alice: Hey, that's not fair! Making a four-dimensional vector out of two two-dimensional ones, how did that happen??

Bob: Ah, but see, if you were a computer scientist, or anyone not used to working frequently with vectors, what could it possibly mean to add two arrays? You have an ordered collections of components, and another such collection. The components can be apples and oranges.

Alice: Or cats.

Bob: Yes. Or numbers. And the most sensible way to add those collections is to make one large ordered collection, in the order in which they were added. Note that there is something interesting here for a mathematician: addition suddenly has become noncommutative.

Alice: I see what you mean. Least surprise for most people. How was that again, about striving for the largest happiness for the largest amount of people? But still, I sure would like to have my vector addition.

Bob: Now the good news, in Ruby, is: you can, and rather easily! All you need to do is to introduce a new class, let us call it `Vector`, and implement it as an array for which the addition operator `+` does what you want it to do. Let's try it right away. I will put the definition of the `Vector` class in the file `vector1.rb`:

```
  class Vector < Array
   def +(a)
     sum = []
     self.each_index{|k| sum[k] = self[k]+a[k]}
     sum
   end
   def *(a)
     if a.class == Vector              # inner product
       product = 0
       self.each_index{|k| product += self[k]*a[k]}
     else
       product = []                    # scalar product
       self.each_index{|k| product[k] = self[k]*a}
     end
     product
   end

 end
```

Alice: Ah, I see that Ruby uses the same `==` construct as C does, to test for equality. Many students are going to trip on that, when they write `if a = b`, not realizing that they are now assigning the value of the variable `b` to the variable `a`.

Bob: Yes, in Ruby, like in C, this can easily give rise to confusion. Here is the correct usage:

```
|gravity> irb --prompt short_prompt
001:0> a = 3
3
002:0> if a == 4
003:1>   p "amazing!"
004:1> else
005:1*   p "of course not"
006:1> end
"of course not"
nil
```

And here is what happens often during the first few weeks that you are programming in C, C++ or Ruby:

```
007:0> if a = 4
008:1>   p "amazing!"
009:1> else
010:1*   p "of course not"
011:1> end
(irb):7: warning: found = in conditional, should be ==
"amazing!"
nil
```

Alice: Hey, that is nice! `irb` tells us that we are probably making a mistake. Very clever!

Bob: And very friendly!

7.2 Debugging

Alice: Before further analyzing what it all means, what you have put in the `Vector` class definition in `vector1.rb`, let us first see whether it works.

Bob: Here we are:

```
|gravity> irb --prompt short_prompt -r vector1.rb
001:0> a = Vector[1, 2, 3]
[1, 2, 3]
002:0> b = Vector[0.1, 0.01, 0.001]
[0.1, 0.01, 0.001]
003:0> a += b
[1.1, 2.01, 3.001]
```

```
004:0> p a
[1.1, 2.01, 3.001]
nil
005:0> a = b*2
[0.2, 0.02, 0.002]
006:0> a*b
TypeError: cannot convert Vector into Integer
from (irb):6:in ‘*’
from (irb):6
```

Alice: Ah, what a pity. We had vector addition, and also scalar multiplication of a vector, but the inner product of two vectors didn't work.

Bob: That is puzzling. Hmmm. Cannot convert Vector into Integer. And all that somewhere in the definition of `*`. There are only three vectors involved, within the definition of `*(a)`. There is the instance of the vector class itself, called `self`, there is the other vector `a` with which it can be multiplied in a inner product, and there is the temporary vector `product[]`. Neither of the first two are threatened anywhere with conversion into an integer. Nor is the third one. Hmmm.

But wait, I have tried to use the same name `product[]` for a vector, in the `else` clause, and for a scalar, in the `if` clause. I thought that Ruby was so clever that you could do that. Because a type is determined only at run time, I thought that you could wait to see which branch of the `if` statement would be actually traversed, to see which type Ruby is giving to `product` array or scalar value. In either case, that is what should be returned.

I guess I was wrong. After all, Ruby is doing a lot of clever thinking behind the scenes, in order to give us this nice behavior of minimum surprise. So I guess that Ruby noticed that product is being defined as an array in the `else` clause, while the value 0 is assigned to it in the `if` clause just above. Could that be the reason that `irb` is complaining that a `Vector` cannot be converted to an `Integer`?

If my theory is right, I should be able to resolve it by giving two different names to the product, `iproduct` for inner product, and `sproduct` for scalar product. Let me try that, and name the file `vector2.rb`:

```
class Vector < Array
  def +(a)
    sum = []
    self.each_index{|k| sum[k] = self[k]+a[k]}
    sum
  end
  def *(a)
    if a.class == Vector              # inner product
      iproduct = 0
```

```
      self.each_index{|k| iproduct += self[k]*a[k]}
      iproduct
    else
      sproduct = []                   # scalar product
      self.each_index{|k| sproduct[k] = self[k]*a}
      sproduct
    end
  end
end
```

We have seen that addition and scalar multiplication all went fine. Let me see whether we can get the inner product to work now.

```
|gravity> irb --prompt short_prompt -r vector2.rb
001:0> a = Vector[1, 2, 3]
[1, 2, 3]
002:0> b = Vector[0.1, 0.01, 0.001]
[0.1, 0.01, 0.001]
003:0> a*b
0.123
```

Ha! That was it. Interesting. But at least it now works.

7.3 An Extra Safety Check

Alice: I heard you talking to yourself, but I must admit, I couldn't follow all that. We'll have to step through the class definition a lot slower, so that I can catch up. But before doing that, are you *sure* that it now works, and more importantly, that your understanding why it went wrong is correct, so that you can avoid that error, whatever it was, in the future.

Bob: Well, I must be right! I reasoned my way through, starting from the error message, and *voila*, the correct result appeared.

Alice: Well, you *may* be right, but just to make sure, why don't you repeat the exact same sequence of commands that you give before, all six of them.

Bob: What good would that do? They worked, and I didn't change anything about the way they worked!

Alice: Just to make me feel better.

Bob: Oh, well, it's only six lines typing. Here you are:

```
|gravity> irb --prompt short_prompt -r vector2.rb
001:0> a = Vector[1, 2, 3]
```

```
[1, 2, 3]
002:0> b = Vector[0.1, 0.01, 0.001]
[0.1, 0.01, 0.001]
003:0> a += b
[1.1, 2.01, 3.001]
004:0> p a
[1.1, 2.01, 3.001]
nil
005:0> a = b*2
[0.2, 0.02, 0.002]
006:0> a*b
TypeError: cannot convert Vector into Integer
from (irb):6:in `*'
from (irb):6
```

I'll be darned!!

Alice: No comment.

Bob: How can that possibly be??? The same error message as before, with the same operation. And all the previous five tests were still okay. And yet, I just showed you that `a*b` worked, by itself, with the same vectors no less! This is spooky.

Alice: The same vectors? But didn't you add `b` to `a` in line 3? And wait, you changed `a` again in line 5.

Bob: Oh yes, true, but what difference can that make? Hmm. Forget I just said that. All bets are off now, for me. I will multiply those last two vector values. I won't go home before I've traced this bug, somehow. This is perplexing. Okay:

```
|gravity> irb --prompt short_prompt -r vector2.rb
001:0> a = Vector[0.2, 0.02, 0.002]
[0.2, 0.02, 0.002]
002:0> b = Vector[0.1, 0.01, 0.001]
[0.1, 0.01, 0.001]
003:0> a*b
0.020202
```

WHAT?!? I now typed in the exact values, and it still works. But the original sequence of six commands, above, didn't work.

7.4 And Yet It Is

Alice: At least now we know what works and what doesn't. And all we have to do is to find out why. Just to put it back to back, let me repeat all six commands again, and print out the values of a and b before taking their inner product:

```
|gravity> irb --prompt short_prompt -r vector2.rb
001:0> a = Vector[1, 2, 3]
[1, 2, 3]
002:0> b = Vector[0.1, 0.01, 0.001]
[0.1, 0.01, 0.001]
003:0> a += b
[1.1, 2.01, 3.001]
004:0> p a
[1.1, 2.01, 3.001]
nil
005:0> a = b*2
[0.2, 0.02, 0.002]
006:0> p a
[0.2, 0.02, 0.002]
nil
007:0> p b
[0.1, 0.01, 0.001]
nil
008:0> a*b
TypeError: cannot convert Vector into Integer
from (irb):8:in ‘*’
from (irb):8
```

Bob: This can't be.

Alice: And yet it is. Something must be different. Something must have happened. I wonder, do we know what a and b *really* are? They are the only variables in town, in this small session. And the session is so short, we should be able to look at them under a microscope at every step. And let us try to make our steps as small as we can, doing only one thing at a time.

Bob: Okay, divide and conquer. That's generally a good method. Let me start a new irb session. Above, we started by creating a vector with three components, to which we assigned values. One way to split this command into smaller steps is to separate the creating and assigning parts. First, I'll create a vector and then I'll assign its values. I'll do the same for the second vector, replacing the second command above by a creation and assignment statement.

Alice: And then I suggest you replace the third command by a simple addition, rather than the += combination.

Bob: Good, more divide and conquer strategy! Let's see what sort of surprise we'll run into next.

```
|gravity> irb --prompt short_prompt -r vector2.rb
```

```
001:0> a = Vector.new
[]
002:0> a = [1, 2, 3]
[1, 2, 3]
003:0> b = Vector.new
[]
004:0> b = [0.1, 0.01, 0.001]
[0.1, 0.01, 0.001]
005:0> a + b
[1, 2, 3, 0.1, 0.01, 0.001]
```

AAHHHAAAA!! There we have a clear smoke signal. A new trail to follow! But the trail seems to hang in mid air . . .

7.5 Using a Microscope

Alice: Somehow Ruby has fallen back onto its old habit of adding by concatenation. But I thought you had taught Ruby to not do that any longer?

Bob: I taught Ruby not to concatenate when adding vectors. Normal arrays will still be concatenated by addition. But `a` and `b` are *not* normal arrays. You are my witness that we have created both `a` and `b` as vectors, brand new vectors, right out of the oven! And yet they don't behave like vectors. I'm beginning to despair. There seems to be no room left here for introducing a bug.

Hmmmm. Maybe we should go over the vector class definition in more detail. There just has to be something wrong there.

Alice: Before doing that, let us enlarge the magnification of our microscope just a bit more. We have taken smaller steps now, but we haven't yet looked into the results of each step further. In other words, we have divided, but not yet conquered.

How about asking `a` and `b` themselves what they think is going on? For starters, we can ask them their type, what class they think they are, with the `class` method we have seen before.

Bob: That's not a bad idea. Actually a great idea. Let's try it.

```
|gravity> irb --prompt short_prompt -r vector2.rb
001:0> a = Vector.new
[]
002:0> a.class
Vector
003:0> a = [1, 2, 3]
[1, 2, 3]
004:0> a.class
```

```
Array
```

What the heck . . . !! How can that possibly be? We created `a` as a vector, it told us that it was a vector, we gave it some values, and now it thinks it is an ordinary array.

Alice: It seems to have forgotten its vectorness. Growing up in a bad environment, it seems.

Bob: But how bad can this environment be? Let me think, and stare at the third line for a while. Hmmppfff . . . On the left hand side we have a variable called `a` that has just told us that it considers itself to be a decent citizen of class `Vector`. Good. On the right hand side we have an array with three components. Good. We give those values to the vector and . . . hey, that must be it!

Alice: what must be what?

Bob: That's the answer!

Alice: You sure look happier now, but can you enlighten me?

Bob: Dynamic typing!!

Alice: Yes, Ruby has dynamic types, how does that solve our problem?

7.6 Following the Trail

Bob: Elementary, my dear Watson. Because Ruby has dynamic typing, upon assignment of an array to a vector, the vector changes into an array. Instantly. Like a chameleon, changing colors to adapt to its environment!

Alice: I told you `a` was growing up in a bad environment! But yes, now I see what you mean. How very tricky.

Bob: And this explains all the error messages we have seen so far. As long as we did not feed any of our vectors, and only printed the results of operations on them, they remained pristine little vectors. But as soon as we fed them a value, they flipped into the type of that value, confusing us to no end. It all makes sense now . . . .

Alice: I have heard that before.

Bob: I know, I know. But no, I can *prove* to you that we have traced the bug. We can easily repair our `Vector` class. So let's go back into the file `vector2.rb`. Ahh, no, it is even easier!

Alice: What is?

Bob: See, this business with introducing `iproduct` and `sproduct`, you know what? I bet that was not necessary. I got sidetracked, in the wrong direction.

Alice: Sidetracking usually gets you in the wrong direction all right, but what was it that was wrong?

Bob: Ah, remember? You said something about hearing me talking to myself, and not being able to follow all that. That was when I had convinced myself that the type problem lie in the use of different types for the variable `product` in the `Vector` class definition. But my first attempt *was* right, after all. You *can* perfectly well use the same variable to stand for different types in different branches of an `if` statement. It is *exactly* because Ruby is a dynamically typed language that you can do that!

So if I would have *really* thought through what I was trying at the very start, while I was writing the `Vector` class, I would have realized the chameleon behavior of Ruby variables. The problem was *not* with switching color somewhere in the process, but the problem was to make sure that the right color was being returned at the end! Object oriented programming! Epiphany!

Alice: Modularity!

Bob: Modularity if you like, right now I'm ready to go along with anything. This is wonderful, that a language can be so malleable! Wow. I can just begin to see how powerful Ruby is, when you can turn any type into anything else, whenever and wherever you need it, without having to declare and define various things at multiple places and then recompile and link again and so on. What a delight!

Alice: I'm glad that you're glad, and I see that you have undone all your edits in `vector.rb`; how are you going to return the right color?

Bob: Here, at these two places, where I previously wrote `sum = []` and `product = []`. By doing so, I dynamically typed both `product` and `sum` to become an array. At the end of the `+` method, `sum` appears, so the value of `sum` is the value that is returned by `+`, and it is an array, not a vector! That is why `a+b` returned the concatenation of two arrays. Similarly, the `*` method returns the value of `product`, also an array.

So the remedy is really simple. We just have to return vectors instead of arrays. And we can do that by declaring both `sum` and `product` to be vectors from the start. That is all! Here it is, in file `vector3.rb`:

```
class Vector < Array
  def +(a)
    sum = Vector.new
    self.each_index{|k| sum[k] = self[k]+a[k]}
    sum
  end
  def *(a)
    if a.class == Vector              # inner product
      product = 0
      self.each_index{|k| product += self[k]*a[k]}
    else
```

```
      product = Vector.new          # scalar product
      self.each_index{|k| product[k] = self[k]*a}
    end
    product
  end
end
```

7.7 Extreme Programming

Alice: Mind if we test it, by going through the *exact* same moves as before?

Bob: My pleasure! And a pleasure it will be. Here you go:

```
|gravity> irb --prompt short_prompt -r vector3.rb
001:0> a = Vector[1, 2, 3]
[1, 2, 3]
002:0> b = Vector[0.1, 0.01, 0.001]
[0.1, 0.01, 0.001]
003:0> a += b
[1.1, 2.01, 3.001]
004:0> p a
[1.1, 2.01, 3.001]
nil
005:0> a = b*2
[0.2, 0.02, 0.002]
006:0> p a
[0.2, 0.02, 0.002]
nil
007:0> p b
[0.1, 0.01, 0.001]
nil
008:0> a*b
0.020202
```

Alice: As it should be. Congratulations!

Bob: Well, I couldn't have done it without you. Once I got my teeth firmly into the problem, I must admit I felt quite stuck.

Alice: I couldn't even have started if you hadn't showed me how to get going with Ruby. I guess this idea of pair programming is much more efficient than people think. It seems that you can get more lines of code written when you're alone behind a keyboard, and that may or may not be true, but at least when chasing bugs, more eyes make all the difference.

Bob: Yeah, I agree. I've done almost all my coding in the privacy of my own office, or home, depending; but when I worked with someone else on a debugging session, it often helped to look at the same problem from two different directions.

Alice: I find it funny to read so much about it these days in computer productivity literature. They make it sound as if this whole notion of pair programming is this wonderful new invention. When I was at MIT as a graduate student, I dropped in at the AI department quite frequently. And everywhere I saw people coding together. When I did a little project there, I also teamed up with one of my fellow students. That was an eye opener for me, and ever since I have preferred to program together with someone else. Besides, it is more fun too!

Bob: You're right. Apart from my total annoyance at first with that recalcitrant bug, it has been fun to explore together. And I must admit, even that bug gave me a lot of excitement. Like watching a horror movie and then finding that you're part of the movie itself. But still, it sounds a bit extreme to do all your programming with somebody sitting next to you.

Alice: That must be why they call it *extreme programming* these days, again as if they have just invented something new.

Bob: You're kidding.

Alice: No, I'm not. They really call it that! But enough management talk. We have certified our vector class, let's now put it to work for our integrator.

Bob: Certified, he? You must be born to become a manager.

Alice: Go integrate.

Chapter 8

Formula Translating

8.1 A Body with Vectors

Bob: Okay, here is the next step. We first have to tell our `Body` class that the positions and velocities are no longer arrays, but vectors. Let me create a new file `vbody1.rb` for this modified class. I will call the modified class `Body` as well. If we are happy with it, we can discard the previous `Body` class definition.

```
require "vector3.rb"

class Body

  attr_accessor :mass, :pos, :vel

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

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

  def pp               # pretty print
    print to_s
  end

  def simple_print
    printf("%24.16e\n", @mass)
```

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

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

end
```

Here it is. I have added one line and modified one line, that's all! The line on top `require "vector3.rb"` I have added so as to make Ruby aware of our new class `Vector`; as we have seen, the Ruby interpreter effectively replaces this line by the file `vector3.rb`. Now the only thing I changed is the initialization line for `pos` and `vel`. They are now *bona fide* vectors, right when they see the light of day.

The next step is to adapt our integrator, so that it can handle particles with vectors for their positions and velocities. Remember, that was the whole reason for the exercise: to simplify the notation in our forward Euler integrator. You were unhappy with the `|k|` notation, and that started us off on this long trek.

Alice: But we learned a lot on our journey. Yes, I remember now. I was complaining about component notation, and then you decided you can give me real vector notation, without components. Seems like a while ago!

8.2 Shrinking Code

Bob: But now we're getting close. Let me rename the old file `euler1.rb`, and then add vectors to `euler2.rb`. No, let me be more careful. Let me call it `euler2a.rb`, for alpha version of `euler2.rb`. If and when it works, we can rename it `euler2.rb`.

Alice: You're getting cautious! Can you remind me what `euler1.rb` looked like?

Bob: Here it is, or was; or will be – but not for much longer:

```
require "body0.rb"

include Math

dt = 0.01             # time step size
ns = 100              # number of time steps
```

```
b = Body.new
b.simple_read

ns.times do
  r2 = 0
  b.pos.each {|p| r2 += p*p}
  r3 = r2 * sqrt(r2)
  acc = b.pos.map { |x| -b.mass * x/r3 }
  b.pos.each_index { |k| b.pos[k] += b.vel[k] * dt }
  b.vel.each_index { |k| b.vel[k] += acc[k] * dt }
end

b.pp
```

All we have to do is to change the first `require` line to require `vbody1.rb` to be loaded instead of `body.rb`, and then we are free to purge the `do` loop from the obnoxious `k` component notation. Are you ready for this?

Alice: Sure thing!

Bob: Here it is. One line shorter in total, with most of the lines shorter in length as well. I've put it into `euler2a.rb` for now:

```
require "vbody1.rb"                                                      #1

include Math

dt = 0.01           # time step size
ns = 100            # number of time steps

b = Body.new
b.simple_read

ns.times do
  r2 = b.pos * b.pos
  r3 = r2 * sqrt(r2)
  acc = b.pos * (-b.mass / r3)
  b.pos += b.vel * dt
  b.vel += acc * dt
end

b.pp
```

8.3 FORTRAN

Alice: One of these days you'll shorten it so much that you can hand me a one-line integrator, just like you produced one-line read and write functions, earlier!

Bob: Don't count on it; I think a five-line integrator is good enough already. What a difference with `euler1.rb`! Instead of asking `b.pos` to add the square of each of its components to `r2`, after setting `r2` to zero, we simply order `r2` to be the inner product of `b.pos` with itself, in the first line of the `do` loop.

The next line is unchanged. But then the fun really starts: the acceleration vector is now directly given as the product of the velocity vector with the terms mass over distance cubed.

Alice: Just as we saw it in formula form earlier:

$$\mathbf{a} = -\frac{M}{r^3}\mathbf{r} \tag{8.1}$$

Bob: So you see, now we can translate our formulas directly into computer code!

Alice: Perhaps Ruby should be called FORmula TRANslator.

Bob: I'm afraid that name has already been taken, quite a few decades ago . . .

Alice: And the next two lines clearly mean:

$$\begin{aligned}\mathbf{r}_{i+1} &= \mathbf{r}_i + \mathbf{v}_i dt \\ \mathbf{v}_{i+1} &= \mathbf{v}_i + \mathbf{a}_i dt\end{aligned}$$

also just like we wrote it before. What a relief, to write a computer code as if you write equations directly down into the software!

Bob: Indeed. I've seen C++ doing that, what they call overloading operators, and you can get a similar notation, but with far more work.

Alice: And with a complex class definition and declaration structure that is very hard to play with and to change at will. In contrast, here with Ruby we can really do what they always advertise as rapid prototyping.

Bob: But before congratulating ourselves too much, let us first see whether this now runs. I've grown a bit more cautious after those surprises we got earlier. Ruby is certainly powerful, but as we have seen, we'd better know what we are doing, in order to use all that power wisely.

Alice: Well said, Bob! Let's run the new version of the integrator with the same values we had before, which I see you have still there in the file `euler2a.rb`: a hundred time steps of 0.01 time units each.

Bob: Here goes:

```
|gravity> ruby euler2a.rb < euler.in
euler2a.rb:12:in ‘*’: can’t convert Array into Integer (TypeError)
from euler2a.rb:12
from euler2a.rb:11:in ‘times’
from euler2a.rb:11
```

8.4 An Old Friend

Alice: Where did we see that error message before?

Bob: I know, it is an old friend, isn't it? I had hoped we had banned it forever. But at least we have an idea now where it could come from. I bet that there is still a vector somewhere that somehow thinks it is an array . . .

Alice: Which line is line 12 in `euler2a.rb`

Bob: it's the very first line in our `do` loop. Just a few lines above, `b` is freshly minted, so perhaps `b.simple_read` is the bad influence that turns some vector into an array? Could be. But before jumping to conclusions, let me sprinkle in a few `class` statements.

Alice: You *are* getting cautious!

Bob: Yeah, but I don't want to waste more time, this time around. Let me do it in `irb` directly, so that we can get everything echoed.

```
|gravity> irb --prompt short_prompt -r vbody1.rb
001:0> include Math
Object
002:0> dt = 0.01
0.01
003:0> ns = 100
100
004:0> b = Body.new
#<Body:0x400cda40 @vel=[0, 0, 0], @pos=[0, 0, 0], @mass=0>
005:0> b.class
Body
006:0> b.pos.class
Vector
007:0> b.simple_read
```

And now it hangs. Hello, computer!? What are you waiting for?

Alice: You haven't given it the `euler.in` input file.

Bob: Ah, of course. I can see that there are drawbacks of using an interactive interpreter if you're testing something that has dependencies with other things. Well, the file is short enough to type in by hand. Here goes:

```
007:0> b.simple_read
1
1 0
0 0.5
[0.0, 0.5]
008:0> b.pos.class
Array
```

Alice: Our old friend, the vector-turned-array bug! So you were right, after all, suspecting that `simple_read` was the culprit.

Bob: What do you mean 'after all'? I was almost certain, but now I know for sure.

8.5 Magic

Alice: Right you are! Now what is wrong with `simple_read`?

Bob: Let's inspect. Here is the definition:

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

And, yes, it is obvious! `@pos` receives its value from a `gets.split.map` operation, in other words first `gets` is invoked to read a line, then the string method `split` splits the string that `gets` just returned, in the form of an array, and finally `map` does something on all the elements of that array, returning a new array with the results. And that last part is where the trouble arises: `map` returns an array that is assigned to `@pos`.

Alice: And poor `@pos`, Ruby chameleon that it is, instantly degrades into an array, forgetting its whole rich vectorness. Of course. Well, when you get some more experience, debugging becomes a breeze, doesn't it?

Bob: Ah, but then you start writing more complicated programs and then you run into more subtle bugs. It's like an arms race.

Alice: We'll see. For now I'm just glad that we tracked this one down quickly. So what do we do about it?

Bob: Simple. We just turn the right-hand side of the offending statement into a vector, rather than an array.

Alice: But how do you do that?

Bob: Ah, when I browsed in the manual, I saw some neat trick. Just a moment. Here it is. Are you ready for this? This is good job for `irb`. Neat and simple.

```
|gravity> irb --prompt short_prompt -r vector3.rb
001:0> a = [1, 2, 3]
[1, 2, 3]
002:0> a.class
Array
003:0> b = Vector[*a]
[1, 2, 3]
004:0> b.class
Vector
```

Alice: Huh? How did that work?? What type of magic did you just invoke, putting a star in front of an array, then wrapping it in square brackets, and pulling a `Vector` spell on it?

Bob: Another convenient Ruby trick. It takes a while to learn all these tricks, but they sure are useful. Here is what happened. The `*` notation in front of an array effectively dissolves the outer `[]` brackets, turning an array into a comma separated list of its elements.

Alice: Ah, and then you feed that list into the initializer of a new `Vector` object. Very clever, if not devious!

Bob: So I'll just clothe the right hand side of the `@pos` and `@vel` assignments in `Vector[* ]` garb. Let me call this file `euler2b.rb` with `b` for beta version. The only difference is that in `euler2a.rb` I started at the top with

```
require "vbody1.rb"
```

while in `euler2b.rb` I started at the top with

```
require "vbody2.rb"
```

instead. I think we're getting closer now. Here it is, in `vbody2.rb`:

```
  def simple_read
    @mass = gets.to_f
    @pos = Vector[*gets.split.map{|x| x.to_f}]
    @vel = Vector[*gets.split.map{|x| x.to_f}]
  end
```

And let me try it:

```
|gravity> ruby euler2b.rb < euler.in
  mass = 1.0
   pos = 0.443229564341409, 0.384215558686636
   vel = -1.29436531289392, 0.0258120006669036
```

Alice: Congratulations!

8.6 A Matter of Taste

Bob: Thanks. So now we have a FORmula TRANslation device that actually works!

Alice: But it is ugly.

Bob: I beg your pardon?

Alice: It is ugly.

Bob: How so?

Alice: I don't like that `[*...]` notation. Looks like a hack.

Bob: I like hacks.

Alice: I know you do. But can't we come up with something cleaner?

Bob: Like what?

Alice: Well, we have seen that `to_s` will happily turn something into a string; we even gave our `Body` class its own `to_s` method, so that it could show its face in public. I bet there is a method called `to_a` that turns objects into arrays.

Bob: There is. And it does just that.

Alice: Can't we then write our own `to_v` method which turns something into a vector?

Bob: Now that's a thought. Sure. That would be fun! And I just know how to do that! I only have to add a few lines to the `vector3.rb` file. I think this is really what we want. To celebrate, let me call the file `vector.rb` now.

Here, it is short, let me show the whole file:

```
class Vector < Array
  def +(a)
    sum = Vector.new
```

```
    self.each_index{|k| sum[k] = self[k]+a[k]}
    sum
  end
  def *(a)
    if a.class == Vector              # inner product
      product = 0
      self.each_index{|k| product += self[k]*a[k]}
    else
      product = Vector.new           # scalar product
      self.each_index{|k| product[k] = self[k]*a}
    end
    product
  end
end

class Array
  def to_v
    Vector[*self]
  end
end
```

8.7 *Voila*

Alice: That's much better. I don't mind the `[*...]` *per se*, it is only that the line with `gets.split.map` followed by a block of code just got too crowded for comfort. But your definition of `to_v` is clean and simple. So let me check: by writing `class Array` and so on, you are just extending the Ruby-defined standard array definition, right?

Bob: Right. Not only can you extend your own classes by giving new bits and pieces like this, as we have seen before, you can do the same thing to the predefined classes. In general, almost nothing in Ruby is off limits; you can typically treat the built-in stuff as if you had just written it all yourself.

Alice: That's really neat. So now can we get rid of the clutter at the bottom of the `vbody2.rb` file?

Bob: Sure thing. Here is our `simple_read`, no clutter, and with the new `to_v` at the end. And let me call this one `vbody.rb`, to celebrate that now we know what we are doing:

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

```
end
```

Alice: Very nice. And presumably it runs?

Bob: I bet you it will! And to show my confidence, I will call the file that includes `vbody.rb` now `euler2.rb`, since I think we will be beyond alpha and beta testing once we see this.

Alice: You do sound confident. Let's see.

```
|gravity> ruby euler2.rb < euler.in
  mass = 1.0
   pos = 0.443229564341409, 0.384215558686636
   vel = -1.29436531289392, 0.0258120006669036
```

Bob: *Voila!*

Alice: *Very* good.

Chapter 9

Energy Conservation

9.1 Estimating Numerical Errors

Bob: That was quite a long session yesterday, when we were chasing those bugs while we were trying to implement a better `Body` class!

Alice: Yes, but we learned a lot about Ruby.

Bob: And even a thing or two about debugging.

Alice: Perhaps this is a good time to return to our real project: to build progressively better integrators. But before we move on to second-order integrators, as well as introducing graphics and other complications, I would like to do one more test. Even though we now know that our forward Euler scheme converges, we don't know yet how well it conserves energy. Shouldn't we test that explicitly?

Bob: Great minds think alike. I had come to the same conclusion. For the two-body problem, of course you can check all kind of things. You can measure how accurately a bound orbit remains the same ellipse, as it should be, rather than slowly drifting and changing shape, and whether the motion traverses the ellipse at the right speed, and so on. The reason is that we have an analytic solution for the two-body problem.

For the many-body problem in general, however, we are a lot less lucky. All we know is that the total energy and total angular momentum of the system are conserved. That gives us one scalar and one vector, or in three dimensions it gives us four conserved quantities. In addition, there are the position and velocity of the center-of-mass motion, which give us another six conserved quantities, ten in total. So we cannot compare all the details of the motion of the N particles with theory, not by a long shot. An N-body system in three dimensions has $6N$ degrees of freedom, far more than the 10 handles we have in our 2-body case.

However, things are not as bad as they may seem. If you make errors in your integration, and you always make errors at some level, those errors will let the simulated orbit drift away from the true orbit in a random fashion. It would require a very clever conspiracy for those errors to work together in such a way as to keep the energy conserved to a high degree of accuracy, while still introducing much larger errors elsewhere. In practice, therefore, checking energy conservation in an N-body system has become the standard test to see how accurate the integration of an N-body system is.

Alice: Another way of saying this is to picture the phase space of the whole system.

Bob: You mean the six-dimensional space in which you plot for each particle its position and velocity?

Alice: I was thinking about the alternative way of picturing the whole system as one point in a $6N$ -dimensional system. The evolution of an N-body system can then be viewed as the complex motion of this one point through this huge space. Now in this space you can define layers of subspaces, on each of which the total energy of the system is constant. Once the particle starts on such a hypersurface, it should stay there, because energy is conserved. But if we make arbitrary errors in computing its motion, our simulated particle will show some numerical drift in all dimensions available, including the direction perpendicular to the energy hypersurface.

Bob: Which reaches the same conclusion as what I just summarized.

Alice: Yes. Since the dynamics of this one point in $6N$ dimensions is incredibly complicated, the distance away from the energy hypersurface will typically be of the same order of magnitude as the total length of the drift of that master particle that symbolizes our whole N-body system.

Bob: More mathematically minded students might prefer your explanation; I think I'll just stick to my more lowbrow explanation. But however we tell the story, we should of course warn them that this argument fails when we are dealing with an integrator that has a built-in type of energy conservation.

Alice: Yes, indeed, that is very important. It would be easy to construct an integrator that projects the orbit of the master particle in $6N$ dimensions back onto the original constant-energy hypersurface. But that would be cheating. It would get rid of only one degree of freedom of the total error, and leave the other $6N-1$ degrees of freedom uncorrected. It would look great when you test for energy conservation, but at the cost of having lost your handle on checking what is going on.

Bob: Or the way I would put it, you could just wiggle one particle far out, perhaps a particle that has already escaped from the system. By changing the velocity of that particle a little, you will change its kinetic energy, and so you can accurately balance the numerical errors that occur in the rest of the system. But of course that does not make the rest of the system behave in a more accurate way.

Alice: Of course, the two examples that you and I just cooked up are extreme. It is not at all a bad idea to conserve energy, if your approach is part of a systematic way to make the whole integration more accurate. Symplectic integrators are an example.

Bob: I have heard of the name, but I must admit I don't know what they are.

Alice: They are a lot of fun, from a mathematical point of view. How useful they may be for real N-body simulations, that is not too clear. They are more accurate, for sure, but they are hard to generalize to individual time steps. And for our game, we have no other choice but to go to individual time steps. Perhaps we can come back to this topic, some day, but not any time soon. But just to give you one example of the simplest symplectic integrator: the good old leapfrog scheme, also known in some areas of physics as the Verlet algorithm.

Bob: Which is probably the next algorithm we want to implement.

Alice: Yes. So shall we first test energy conservation of our forward Euler code?

9.2 A Driver

Bob: Well, I have a surprise for you. Last night I continued to tinker with Ruby, and I had so much fun doing so that I stayed at it long enough to wind up with a version of our code that produces energy conservation diagnostics.

Alice: How nice! Can I see it?

Bob: First of all, here is the driver, in file `euler.rb`, the only part of the program that we will change, while exploring the effect of different time step sizes and integration times:

```
require "body.rb"

include Math

dt = 0.01          # time step
dt_dia = 1         # diagnostics printing interval
dt_out = 1         # output interval
dt_end = 1         # duration of the integration

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

b = Body.new
```

```
b.simple_read
b.evolve(dt, dt_dia, dt_out, dt_end)
```

Alice: So at the bottom you create a new particle, which will represent the relative motion between the two bodies in our system, with the call to `Body.new`, then you read in the initial conditions, as we did before, and then you tell the particle to evolve itself.

Bob: Yes, with `evolve` I mean that we invoke here a method that computes the evolution of the system in time. I have made this a method in the `Body` class, so we can invoke it with `b.evolve`. The only extra thing needed is to tell the particle how long to integrate, with what time step, and how frequently to report the degree of energy conservation and the actual position and velocity of the relative motion.

Alice: So the actual program is a mere three lines, at the bottom, with the four lines of assignments at the top just the way to specify each of those four parameters you just mentioned.

Bob: Exactly. Ruby is *very* economical! And just as a courtesy, I decided to let the program print out the values of the four parameters. We can then change those parameters in the current file to our hearts' content, and we can always check from earlier outputs what parameters were used for a given integration.

Alice: Good idea. Clean and careful. Of course, some day soon we will want to give those values as command line options, rather than always editing the file.

Bob: I suppose so, although we'll have to think about that. It is of course possible to give Unix-style command line options, but there are several other ways as well; in fact Ruby has a nice built-in way to do that.

Alice: Later. Can I have a look at what the `Body` class now looks like? It must have grown quite a bit, since it now contains not only the forward Euler integrator, but also your whole `evolve` method, as well as whatever you wrote to get all the energy diagnostics done.

Bob: It's not that bad, actually. Ruby really is a compact language. Here, let me show you the whole file `body.rb` first, and then we can go through each of the methods used.

9.3 The Body Class

```
require "vector.rb"

class Body

  attr_accessor :mass, :pos, :vel
```

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

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

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

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

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

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

def epot                      # potential energy
  -1/sqrt(@pos*@pos)
end
```

```
  def e_init                   # initial total energy
    @e0 = ekin + epot
  end

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

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

  def pp               # pretty print
    print to_s
  end

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

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

end
```

Alice: Indeed, not as long as I would have guessed. Can you give me a guided tour?

Bob: My pleasure!

Chapter 10

Diagnostics

10.1 Evolve

Bob: You are familiar already with the accessor shorthand that gives read and write handles to the internal variables for the mass, position, and velocity, as well as with the initialization method, that uses our nifty vector class.

The first new function is `evolve`, the one function that the driver program invokes, and that guides all the work. Here it is, by itself:

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

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

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

```
        t_out += dt_out
      end
    end
  end
```

Bob: At the top, it sets two counters to zero. One is time, which measures how the simulated time during which the two-body motion is integrated, so that the calculation can be halted properly after an amount of time equal to dt_end has passed. The other is nsteps, the number of time steps that the system has taken. I keep track of that, since I like to know how much work the code has done, at any given point.

Alice: With a constant time step, you could also divide the total integration time by the step size.

Bob: Of course, but in future codes we'll soon switch to variable time steps, in which case you have no independent way of knowing how many steps the code took. So we may as well get into the habit of reporting that number.

Alice: In the third line you introduce a variable. What is it doing there, all by itself? It looks a bit lonely and lost.

10.2 Brevity and Modularity

Bob: Ah, the brevity of Ruby! e_init is not a variable, it is a method, in other words a function or subroutine, depending on your dialect.

Alice: Without parentheses?

Bob: Without parentheses! This is a method without arguments. So why put parentheses around nothing?

Alice: But in most languages I have worked with, if a function do_something has no arguments, you still have to invoke it with do_something(), or call do_something. Otherwise how will the compiler know that it is dealing with a function, sorry, method, rather than a variable?

Bob: The compilers of the languages you are familiar with cannot. But here, as is also the case in Perl, we are dealing with a really *smart* language.

Alice: You mean the Ruby compiler can figure out . . .

Bob: . . . the Ruby interpreter, you mean . . .

Alice: . . . ah yes, the Ruby interpreter, how can it figure out when a name stands for a variable and when it stands for a method?

Bob: Ruby will look around within the proper context, in this case within the body of the class definition we are in . . .

Alice: . . . the body of Body you mean . . .

Bob: . . . yes, names can get confusing, even for human interpreters. And the Ruby interpreter will then see a statement `def e_init`, and will realize that the name `e_init` belongs to a method.

Alice: How elegant.

Bob: Indeed. Of course, you are perfectly free to write `e_init()` instead, if you like, but I prefer brevity. So this third line of `evolve` invokes the method `e_init` below. We'll look at that in a moment, but all it does is compute the total energy of the system at the beginning of a run, and store the value in an internal variable, so that we can compare the energy at a later time with the initial value, to see how much the energy has drifted.

The next line invokes the method `write_diagnostics`, which does exactly what its name says. It has two arguments, `nsteps` and `time`, since those two counters have been defined locally within the body of the `evolve` method, and therefore they are not visible outside that method. I want `write_diagnostics` to write those numbers as well, so I have to pass them explicitly.

Alice: You could of course have defined them as internal variables for the `Body` class, what did you call them again? Ah, instance variables.

Bob: Yes, but that would be confusing and potentially dangerous. The `Body` class is growing bigger now, so I don't like to have an unnecessary proliferation of internal variables that float around inside, visible for every function. Debugging then becomes much harder. There is a reason that the human body has cells and that cells have cell walls and that within a cell there is the membrane around the nucleus as well as a variety of other compartmentalized structures.

Alice: See, that is what I mean when I insist on modular programming.

Bob: If so, I agree with you on this level. The problem I had with your earlier description was that you seemed to want to compartmentalize on higher levels as well. I am glad that each cell of my body does not share its contents with every other cell. But I would be unhappy if my left hand would not know what my right hand would be doing. And even though I don't like to put my foot in my mouth, I'm glad I have the freedom to do so!

Alice: We'll come back to that. For now, I'm glad we're in agreement about modularity on a local level. While creating larger programs, we will move up to more global issues, and there we can fight things out with concrete examples. Meanwhile, I'll try not to tease you any longer about modularity. This code already looks as modular as you can ask for.

10.3 Keeping Time

Bob: Moving right along, we see three lines of assignment. Here I set up a book keeping process that will tell us when we need to take some action other than integrating. The variable `t_dia` stores the next time at which we want to report energy diagnostics. Initially we set it equal to the time interval between diagnostics output, which is `dt_dia`. Later on during the integration, there may be many times at which you want to see the energy being reported. Each time that is done, the variable `t_dia` is updated to the subsequent diagnostics output time, as we will see in a moment. So first we have a trivial looking `t_dia = dt_dia`, but after the first output `t_dia` will be changed to `2dt_dia`, then to `3dt_dia`, and so on, until the integration finishes.

Alice: Except that there is this extra term `- 0.5*dt`.

Bob: Ah, I have added that because the time variables are all floating point numbers. So when we make a comparison between the actual time `time` and the time `t_dia` at which you want to have an output done, it is possible that the actual time is still ever so slightly smaller than the time for the next scheduled output. In that case, the integrator will happily take another step. But when you then change the time step, the second run will stop at a different place. This will make it difficult to compare the results between two different runs: are the final particle positions different because the integrations have different accuracy or because one of the two calculations happened to overshoot? Subtracting half a time step guarantees that the system is ready to trigger an output call, already half a time step before the predicted time. Whether the system lands exactly on the predicted time or slightly before or after, in all cases it will halt at the right time. The reason is that floating point round off occurs typically in the fifteenth decimal or so, so the error is far smaller than any conceivable time step.

Alice: I'm glad you thought about that.

Bob: In a similar way, the variable `t_out` stores the time at which the next particle output will be done, *i.e.* the time at which we print the mass, position, and velocity of the relative motion. The mass will not change, of course, although we could introduce mass losing stars or even exploding stars at some point later on. But the position and velocity will change, as we have seen. Sometimes we may be interested in getting frequent output of these numbers, for example when we will make a movie of a particle motion, as we will do soon. At other occasions we may be more interested in studying the energy drift with a fine comb. So by having `dt_out` and `dt_dia` as free dials to determine the output frequency of positions and velocities on the one hand, and the energy on the other hand, gives us a lot of flexibility.

The variable `t_end` is similar to the other two I just discussed, except that it will not be updated during the integration: when `t_end` is first reached, that will be the end . . .

Alice: . . . of a single call to `evolve`. I presume that you can give repeated calls to the `evolve` method, from your driver, if you chose to do so.

Bob: Indeed. But each time you invoke `evolve`, everything starts from scratch again within the body of `evolve`; the `evolve` method itself has no idea whether it has been called before or not. It just does its thing, starting with what it thinks is time zero.

Alice: So if you call `evolve` twice, the time is set back to zero in between.

Bob: For now, yes. That was just the simplest way to implement it. Time is used here only internally, for bookkeeping during a single invocation of `evolve`. Of course, I could have introduced a time variable on the level of the driver. But then I would have to pass that also to `evolve` and I thought that using already four arguments was enough for a first attempt. Anything can be made arbitrarily complex quickly, and I like to keep things simple.

Alice: That makes a lot of sense. So all we have left to look at within `evolve` is the `while` loop, which keeps track of time. When the time `time` exceeds the value of `t_end`, the method stops and control is handed back to the driver, which called `evolve`.

Bob: Indeed. And the first thing that happens is a call to `evolve_step`. Like `evolve`, it integrates the orbit of our particle, but only for one step.

Alice: Divide and conquer.

Bob: Yes, the level of `evolve` is pure management; all the footwork is down in `evolve_step`, which gets invoked anew for each step.

10.4 More Brevity

Alice: In the next line, the time gets updated, by the amount of time that was spent while stepping our particle forward by one step, and similarly the `nsteps` counter is increased by one. I see what happens next. Unless there is an immediate need to output energy, or particle positions and velocities, we just move on. And in those cases where you do an output, you immediately schedule the next output, by increasing the future output time by the corresponding output interval.

Bob: Indeed. And that's it! The next method we have seen before:

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

Newton in action: the acceleration is calculated and both position and velocity are updated.

Alice: Using this coordinate-free notation. I love it. It is so compact! Like a minicar that you love to drive in a crowded city.

Bob: Wait till you get to the diagnostics method, you'll love that even more. But first we have three short methods. The first two calculate the kinetic and potential energy for the relative motion:

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

```
def epot                          # potential energy
  -1/sqrt(@pos*@pos)
end
```

Alice: You can just read them off, kinetic energy quadratic in velocity, potential energy proportional to inverse distance. Wonderful. One-liners are great.

Bob: But before you sometimes called them ugly!

Alice: Ah, *clear* one-liners are great, I meant to say. And if a one-liner can be as clear as multi-liners, then I prefer a one-liner. What I like about Ruby is that I feel that I'm talking to a colleague about science, rather than that I'm talking to a compiler, about types and pointers and various declarations. Without all that clutter, brevity = clarity. But never forced brevity, for its own sake.

Bob: You see, that's why I don't like principles: as soon as you confront people with them, they come up with exceptions. Anyway, note that both `ekin` and `epot` are methods, not variables, just like the method we already saw earlier, `e_init`:

```
def e_init                        # initial total energy
  @e0 = ekin + epot
end
```

As long as this function is called responsibly, *i.e.* only at the beginning of the integration, the internal variable `@e0` will be set to the initial sum of the kinetic and potential energy, in other words to the total initial energy.

Alice: That is funny, that you can just add `ekin` and `epot`, as if they were variables, even though they are really functions.

10.5 Ruby Smiling

Bob: Yes, but as I promised you, it is getting even better. Look at the first line in the method that does the output for our energy diagnostics:

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

Alice: It reads `etot = ekin + epot`. Yes, of course, the total energy is the sum of the kinetic and potential energy. But wait, the first variable `etot` must be a local variable, local to the body of the method `write_diagnostics`, while `ekin` and `epot` are methods. That is remarkable! It seems as if we are freely mixing nouns and verbs, and yet everything looks so natural, and I presume it does the right thing.

Bob: It does, yes, I did test it.

Alice: How *very* elegant!

Bob: I knew you would like this line in particular. And I was wondering what grandiose philosophical explanation you would connect with it.

Alice: Easy! It shows that Ruby is a natural kind of language, almost a natural language. In English, you mix nouns and verbs in one sentence without having to think at the start of each sentence how to connect them. You just talk, because you have become familiar with them. In contrast, in a language like C++ you can 'talk' in it for years, and still have to look up the rules of the game, each time you do something new. I've never felt a natural familiarity with C++, no matter how much I use it. And here, how can you not love Ruby, with `etot = ekin + epot` smiling at you!

Bob: You have a point there, Ruby does feel comfortable, like a natural language, or an old shoe.

Alice: You're mixing metaphors, but I know what you mean. Well, let's see what you did with the total energy that you so magically created. You are sending something to the standard error stream, yes?

Bob: That's what you would call it in C++, indeed. We have seen that `print` directs output to the standard output stream, by default. But by calling `STDERR` 's `print` method, we are now sending the string that follows it to the standard error stream.

Alice: Why do you send positions and velocities to the standard output channel, and the energy diagnostics to the error channel? Because you expect errors in your energy?

Bob: Very funny. No, because I may want to separate the two streams, later on. In order to make a graphics movie, for example, we could output the positions every time step, or every ten time steps or so. If we then would ask for energy diagnostics only occasionally, the diagnostics information would get completely lost in the middle of all that other ouput. Besides, the graphics reader could get confused, if it expected only particle data.

Of course you can work around all that with a little awking and grepping, but it is cleaner to separate things right at the start. In that way, you can redirect the standard output to a file, while letting the diagnostics appear on your screen. Or you can put both types of output into separate files.

Alice: I agree completely. You are locally modular! I won't tease you anymore.

Bob: Only praise me? That's fine, that I don't mind at all.

10.6 A Here Document

Alice: While I get the idea of what you want to print out, in the statement starting with `STDERR.print`, I'm completely confused by what follows on the next few lines. Here, let me have the keyboard, and let me at least properly indent it, it hurts my eyes.

Bob: No, stop, don't do that! It has a meaning.

Alice: Meaning?

Bob: I'm glad emacs has an undo. Yes, meaning. The funny symbols `<<END` indicate the beginning of what is called a "here document".

Alice: As opposed to a "there document"?

Bob: It is really called that, and for a simple reason: everything that is here, in between `<<END` and `END` is interpreted as one long string, and that string is the taken as the argument for the `STDERR.print` command.

Alice: What are the backslash symbols `"\"` doing there at the end of the second and third line?

Bob: They indicate that the next line should continue after the previous line. Without them, each of the three lines starting with `E_kin`, `E_pot`, and `E_tot`, would appear on separate lines, and I prefer to have them appear on one line of the output. So I have to escape the first two of the three new-line characters at the end of those three lines.

Alice: I think I get the idea. Basically, the print command gobbles up everything it sees until it comes across the magic word `END` and then it stops.

Bob: Effectively, yes, but more precisely speaking the print command itself doesn't know about "here documents". The symbol `<<END` is what causes the rest up to `END` to be formed into a string. Then that string, once formed, is

what the print command sees. Here, let me show you. Ruby invites you not to take anything on faith, since it is so easy to test things for yourself. When in doubt, test it out!

```
|gravity> irb --prompt short_prompt -r vbody1.rb
001:0> <<END
002:0" this is special string,
003:0" for Alice, from Bob.
004:0" END
"this is special string,\nfor Alice, from Bob.\n"
005:0> s = <<END
006:0" here is another string,
007:0" with another new-line in the middle
008:0" END
"here is another string,\nwith another new-line in the middle\n"
009:0> print s
here is another string,
with another new-line in the middle
nil
010:0> print <<END
011:0" as you can see, and END in the middle does not really end
012:0" the here document.
013:0" END
as you can see, and END in the middle does not really end
the here document.
nil
014:0> print <<END
015:0" Also, no space is allowed between << and END,
016:0" as I will show in a moment
017:0" END
Also, no space is allowed between << and END,
as I will show in a moment
nil
018:0> << END
SyntaxError: compile error
(irb):18: syntax error
<< END
  ^
from (irb):18
```

Alice: It should be called a mountain document, with all that echoing; or better even a valley document, in between those two mountainous ENDs.

Bob: Well, the remaining four methods are the ones we've seen before, so this is the end of the guided tour. Pretty pretty, hey?

Alice: Yes, very, apart from the here document. I still think that looks ugly, but I see that you have no choice.

Bob: The basic question is: do you want pretty code or pretty output. In this case you can't have both.

Alice: I guess in that case I prefer pretty output. But speaking of output, I haven't seen anything yet. You said you've tested the code, and it really works?

Bob: I'll show you.

Chapter 11

Testing Forward Euler

11.1 Starting Again

Bob: Time to run the driver for our new integrator, which will give us the energy conservation diagnostics we have been talking about for so long now! Let us start with the same variables we used for our very first forward Euler run:

```
dt = 0.01           # time step
dt_dia = 1          # diagnostics printing interval
dt_out = 1          # output interval
dt_end = 1          # duration of the integration
```

We concluded that the position was probably right to within a percent or so, from comparing it with a more accurate integration. I'm curious whether the energy check gives us a similar answer.

```
|gravity> ruby test.rb < euler.in
dt = 0.01
dt_dia = 1
dt_out = 1
dt_end = 1
at time t = 0, after 0 steps :
  E_kin = 0.125 , E_pot =  -1, E_tot = -0.875
             E_tot - E_init = 0
  (E_tot - E_init) / E_init =-0
at time t = 1, after 100 steps :
  E_kin = 0.838 , E_pot =  -1.7, E_tot = -0.867
```

```
            E_tot - E_init = 0.00822
  (E_tot - E_init) / E_init =-0.0094
  1.0000000000000000e+00
  4.4322956434140903e-01  3.8421555868663582e-01
 -1.2943653128939152e+00  2.5812000666903590e-02
```

Alice: Did you set this up? A relative energy accuracy of 0.94%, very close!

Bob: That was just pure luck, my argument was only an order of magnitude estimate.

11.2 Linear Scaling

Alice: As we argued, a first-order method like the forward Euler should converge linearly: making the step size ten times smaller should decrease the error size also by roughly a factor of ten. Shall we try?

```
dt = 0.001          # time step
dt_dia = 1          # diagnostics printing interval
dt_out = 1          # output interval
dt_end = 1          # duration of the integration
```

Bob: So we expect an error of 0.1% now. Let's see.

```
|gravity> ruby test.rb < euler.in
dt = 0.001
dt_dia = 1
dt_out = 1
dt_end = 1
at time t = 0, after 0 steps :
  E_kin = 0.125 , E_pot =  -1, E_tot = -0.875
            E_tot - E_init = 0
  (E_tot - E_init) / E_init =-0
at time t = 1, after 1000 steps :
  E_kin = 0.864 , E_pot =  -1.74, E_tot = -0.874
            E_tot - E_init = 0.000825
  (E_tot - E_init) / E_init =-0.000943
  1.0000000000000000e+00
  4.3302178053157708e-01  3.7860453335416983e-01
 -1.3147926449876397e+00  7.1843385614138713e-03
```

Alice: Almost too good: the relative error is now 0.094%. It has decreased by the predicted factor ten. Let's do this:

```
dt = 0.00001        # time step
dt_dia = 1          # diagnostics printing interval
dt_out = 1          # output interval
dt_end = 1          # duration of the integration
```

making the time step smaller by a jump of a factor of a hundred.

```
|gravity> ruby test.rb < euler.in
dt = 1.0e-05
dt_dia = 1
dt_out = 1
dt_end = 1
at time t = 0, after 0 steps :
  E_kin = 0.125 , E_pot =  -1, E_tot = -0.875
             E_tot - E_init = 0
  (E_tot - E_init) / E_init =-0
at time t = 1, after 100000 steps :
  E_kin = 0.867 , E_pot =  -1.74, E_tot = -0.875
             E_tot - E_init = 8.25e-06
  (E_tot - E_init) / E_init =-9.43e-06
  1.0000000000000000e+00
  4.3186966438051116e-01  3.7796470836807455e-01
 -1.3171480879236508e+00  5.0327840503739613e-03
```

Bob: And indeed the error comes down by a factor of a hundred. Note that the integrator halts after exactly 100,000 steps, not one more and not one less. Yesterday evening, when I tried it out, it kept overshooting to 100,001 steps, so that was why I added that extra half step in the halting criteria.

11.3 A Full Orbit

Alice: Let's follow your earlier suggestion to do a longer integration, for ten time units. And let's use your flexible output options. I would like to see four energy diagnostics, but at most one final position and velocity output, so that we don't get too distracted.

Bob: That means:

```
dt = 0.01           # time step
dt_dia = 2.5        # diagnostics printing interval
dt_out = 10         # output interval
dt_end = 10         # duration of the integration
```

I have given dt_out a value larger than the total run time dt_end, so we won't get the regular particle output, only the diagnostics.

```
|gravity> ruby test.rb < euler.in
dt = 0.01
dt_dia = 2.5
dt_out = 10
dt_end = 10
at time t = 0, after 0 steps :
  E_kin = 0.125 , E_pot =  -1, E_tot = -0.875
             E_tot - E_init = 0
  (E_tot - E_init) / E_init =-0
at time t = 2.5, after 250 steps :
  E_kin = 0.936 , E_pot =  -0.54, E_tot = 0.395
             E_tot - E_init = 1.27
  (E_tot - E_init) / E_init =-1.45
at time t = 5, after 500 steps :
  E_kin = 0.603 , E_pot =  -0.209, E_tot = 0.394
             E_tot - E_init = 1.27
  (E_tot - E_init) / E_init =-1.45
at time t = 7.5, after 750 steps :
  E_kin = 0.529 , E_pot =  -0.135, E_tot = 0.394
             E_tot - E_init = 1.27
  (E_tot - E_init) / E_init =-1.45
at time t = 10, after 1000 steps :
  E_kin = 0.495 , E_pot =  -0.101, E_tot = 0.394
             E_tot - E_init = 1.27
  (E_tot - E_init) / E_init =-1.45
  1.0000000000000000e+00
  7.6937453936571965e+00 -6.2777200566159870e+00
  8.1220683064181454e-01 -5.7420020123998905e-01
```

Alice: That's funny, a large energy error right away and then the energy seems to be frozen in.

Bob: Ah yes, this was the run where the system exploded, with the relative distance increasing to a value of about ten.

11.4 Zooming In

Alice: And look, the velocity vector is almost parallel to the position vector, and pointing in the same direction. Clearly, the two bodies are escaping from each other.

Bob: And that explains why the energy error is frozen: there is no longer much interaction going on between the particles. The numerical explosion must have happened at pericenter, as we deduced earlier. Let's have a closer look at what happens in the first part:

```
dt = 0.01           # time step
dt_dia = 0.5        # diagnostics printing interval
dt_out = 10         # output interval
dt_end = 2.5        # duration of the integration
```

A four times shorter run, with five times more frequent diagnostics.

```
|gravity> ruby test.rb < euler.in
dt = 0.01
dt_dia = 0.5
dt_out = 10
dt_end = 2.5
at time t = 0, after 0 steps :
  E_kin = 0.125 , E_pot =  -1, E_tot = -0.875
             E_tot - E_init = 0
  (E_tot - E_init) / E_init =-0
at time t = 0.5, after 50 steps :
  E_kin = 0.231 , E_pot =  -1.1, E_tot = -0.872
             E_tot - E_init = 0.0033
  (E_tot - E_init) / E_init =-0.00377
at time t = 1, after 100 steps :
  E_kin = 0.838 , E_pot =  -1.7, E_tot = -0.867
             E_tot - E_init = 0.00822
  (E_tot - E_init) / E_init =-0.0094
at time t = 1.5, after 150 steps :
  E_kin = 3.37 , E_pot =  -2.95, E_tot = 0.417
             E_tot - E_init = 1.29
  (E_tot - E_init) / E_init =-1.48
at time t = 2, after 200 steps :
  E_kin = 1.26 , E_pot =  -0.865, E_tot = 0.398
             E_tot - E_init = 1.27
  (E_tot - E_init) / E_init =-1.45
at time t = 2.5, after 250 steps :
```

```
  E_kin = 0.936 , E_pot =  -0.54, E_tot = 0.395
             E_tot - E_init = 1.27
  (E_tot - E_init) / E_init =-1.45
```

Alice: So the disaster happened between a time of 1 and 1.5. That must have been when the first pericenter passage occurred, the place where the particles made their closest approach during one orbit.

11.5 Pericenter Passage

Bob: Let's check:

```
dt = 0.01           # time step
dt_dia = 10         # diagnostics printing interval
dt_out = 0.25       # output interval
dt_end = 1.5        # duration of the integration
```

I'll ask only for particle output, once every quarter time unit:

```
|gravity> ruby test.rb < euler.in
dt = 0.01
dt_dia = 10
dt_out = 0.25
dt_end = 1.5
at time t = 0, after 0 steps :
  E_kin = 0.125 , E_pot =  -1, E_tot = -0.875
             E_tot - E_init = 0
  (E_tot - E_init) / E_init =-0
  1.0000000000000000e+00
  9.6984896296142042e-01  1.2383206879933301e-01
 -2.5280864394541708e-01  4.8458243922783906e-01
  1.0000000000000000e+00
  8.7455226567482147e-01  2.3948976867782423e-01
 -5.2597914456085060e-01  4.3083603675399151e-01
  1.0000000000000000e+00
  7.0559656638283363e-01  3.3480149737403997e-01
 -8.5114538471899592e-01  3.1158537532636715e-01
  1.0000000000000000e+00
  4.4322956434140903e-01  3.8421555868663582e-01
 -1.2943653128939152e+00  2.5812000666903590e-02
  1.0000000000000000e+00
  4.3573377318392045e-02  2.9481575851334291e-01
```

```
 -1.9466715027999648e+00 -1.1087280827004695e+00
  1.0000000000000000e+00
 -1.1742854070954888e-01 -3.1759379188140036e-01
  1.2444634633070424e+00 -2.2783744800811112e+00
```

Alice: This is sure hard to read. It is definitely time to get some form of graphics going. This must be how people analyzed the first N-body calculations, in the sixties!

Bob: Yes, we'll get to graphics soon. But let's do some real lab bench work here. Ah! You see, at time 1.25 the separation of the particles is clearly smaller than at other times. That must have been the periastron passage, roughly. So the period of the orbit must be 2.5, give or take.

11.6 Apocenter Passage

Alice: Let's check!

Bob: We can only check when we use smaller step sizes, since this one here is numerically exploding. Let's try this:

```
dt = 0.001          # time step
dt_dia = 2.5        # diagnostics printing interval
dt_out = 2.5        # output interval
dt_end = 2.5        # duration of the integration
```

A ten times smaller step size, and both types of output at time 2.5:

```
|gravity> ruby test.rb < euler.in
dt = 0.001
dt_dia = 2.5
dt_out = 2.5
dt_end = 2.5
at time t = 0, after 0 steps :
  E_kin = 0.125 , E_pot =  -1, E_tot = -0.875
             E_tot - E_init = 0
  (E_tot - E_init) / E_init =-0
at time t = 2.5, after 2500 steps :
  E_kin = 0.202 , E_pot =  -0.844, E_tot = -0.643
             E_tot - E_init = 0.232
  (E_tot - E_init) / E_init =-0.266
  1.0000000000000000e+00
  1.1300052574190800e+00 -3.5392829077199284e-01
  5.6745700562449897e-01  2.8574832366093905e-01
```

Alice: You were right, close to apocenter, the place we started from. Let's print out the initial conditions for comparison:

```
1
1 0
0 0.5
```

Bob: Close indeed. But still a larger energy error. Let's try a ten times smaller time step.

```
dt = 0.0001         # time step
dt_dia = 2.5        # diagnostics printing interval
dt_out = 2.5        # output interval
dt_end = 2.5        # duration of the integration
```

```
|gravity> ruby test.rb < euler.in
dt = 0.0001
dt_dia = 2.5
dt_out = 2.5
dt_end = 2.5
at time t = 0, after 0 steps :
  E_kin = 0.125 , E_pot =  -1, E_tot = -0.875
             E_tot - E_init = 0
  (E_tot - E_init) / E_init =-0
at time t = 2.5, after 25000 steps :
  E_kin = 0.144 , E_pot =  -0.994, E_tot = -0.85
             E_tot - E_init = 0.0255
  (E_tot - E_init) / E_init =-0.0291
  1.0000000000000000e+00
  9.9746583635490227e-01 -1.3247708073972395e-01
  2.6197303614764500e-01  4.6897087837512991e-01
```

Alice: Better! And a linear behavior, with the energy errors getting ten times smaller. Let's shrink the time step by one more factor of ten:

```
dt = 0.00001        # time step
dt_dia = 2.5        # diagnostics printing interval
dt_out = 2.5        # output interval
dt_end = 2.5        # duration of the integration
```

```
|gravity> ruby test.rb < euler.in
dt = 1.0e-05
dt_dia = 2.5
dt_out = 2.5
dt_end = 2.5
at time t = 0, after 0 steps :
  E_kin = 0.125 , E_pot =  -1, E_tot = -0.875
             E_tot - E_init = 0
  (E_tot - E_init) / E_init =-0
at time t = 2.5, after 250000 steps :
  E_kin = 0.143 , E_pot =  -1.02, E_tot = -0.872
             E_tot - E_init = 0.00257
  (E_tot - E_init) / E_init =-0.00294
  1.0000000000000000e+00
  9.7908881919480084e-01 -1.0885124045716744e-01
  2.2087963584449741e-01  4.8637780261985308e-01
```

Bob: Good. Another factor ten more accurate. It is converging slowly but surely. And we remain close to apocenter. I think we have the situation under control.

Chapter 12

Analytical Checks

12.1 Celestial Mechanics

Alice: You know, we really should be able to derive the orbital period analytically. Let me try to remember my celestial mechanics. I remember that there was one equation that had no factors of π or whatever in it. Ah yes,

$$G(M_1 + M_2) = a^2\omega^3 \tag{12.1}$$

Here a is the semi-major axis, ω is the angular frequency of the motion, in other words the period of the orbit is given in terms of ω as $T = 2\pi/\omega$.

Bob: That's a handy formula to remember. What does that give in our case? We started with $G = 1$ and took $M_1 + M_2 = 1$ but what was our initial value for a ?

Alice: We'll have to reconstruct that. It must be larger than 0.5, but not much larger. At the time of pericenter the particles were much closer than at apocenter, which means that the eccentricity was fairly large, and the apocenter distance not much smaller than $2a$.

I remember another handy formula: the total energy in a two-body system is equal to

$$E = -G\frac{M_1 M_2}{2a}. \tag{12.2}$$

That is easy to remember, since the virial theorem tells you that the potential energy is twice as large as the kinetic energy, on average, and therefore the total energy is half the potential energy, also on average. Now in a Kepler orbit, it turns out that the average of $1/r$ happens to be $1/a$.

Bob: I can see that it is useful to remember those qualitative facts. That is easier than trying to remember factors of 3 or 4 in formulas that you learn by heart and than later half forget.

Alice: Yes, the only numbers I like are 1 and 0 and infinity. So let us determine the total energy, and then we now our semi-major axis a.

Bob: But we have that already: according to the output of my program it is `E_tot = -0.875`, initially. In other words, $E = 7/8$.

Alice: Inverting my previous equation, we get

$$a = -G\frac{M_1 M_2}{2E}. \tag{12.3}$$

With $G = M_1 + M_2 = 1$, we get

$$a = -\frac{1}{8E}. \tag{12.4}$$

So with your value $E = 7/8$, that gives us $a = 1/7$. But hey, that can't be right. It should be larger than 0.5, since the maximum distance between two particles in *any* Kepler orbit is $2a$, and we started at a distance of unity!

Bob: That is puzzling. But you just stepped through my code. You were so happy with the clarity of the statements like `etot = ekin + epot`.

Alice: Let's do an independent check. This is like debugging, but now on the level of the physics, rather than the numerics. Let me just use pen and paper to determine the initial total energy. Here are the equations:

$$E = E_{kin} + E_{pot} = \tfrac{1}{2}M_1 v_1^2 + \tfrac{1}{2}M_1 v_1^2 - G\frac{M_1 M_2}{r} \tag{12.5}$$

with $G = 1$ and $M_1 = M_2 = 0.5$, right?

Bob: Right. And these are velocities in the center of mass frame of the two particles. These are equal in magnitude but opposite in direction, so each one is one half of the relative velocity. The original relative velocity was $1/2$, so each of the two is $1/4$, and $v_1^2 = v_1^2 = 1/16$. Let's do it very carefully, to make sure we don't drop some factor somewhere. We have now for the initial total energy:

$$E = \tfrac{1}{2}\tfrac{1}{2}\tfrac{1}{16} + \tfrac{1}{2}\tfrac{1}{2}\tfrac{1}{16} - \frac{\tfrac{1}{2}\tfrac{1}{2}}{1} = \tfrac{7}{32} \tag{12.6}$$

Alice: That is exactly four times smaller than the value that your program gave us. And it would imply a semi-major axis of

$$a = -\frac{1}{8E} = \frac{4}{7} \tag{12.7}$$

Now that is a much more reasonable number! Just as I had predicted: larger than 0.5, but not much larger. Look:

```
|gravity> bc -lq
4/7
.57142857142857142857
```

This has to be correct. We did it from first principles, little step for little step, and the result is just what was expected.

12.2 The Role of the Masses

Bob: If this is right, then the question is what went wrong with my program? I agree that a value of $a = 1/7$ is unphysical. But like I said, you just checked with me every statement in the code!

Alice: Well, your calculation can't be all wrong. You had the factor 7 in the denominator, that's unlikely to come out correctly by chance. You were off by a factor 4. I think there must be something wrong with your units.

Bob: My units??? You saw as well as I did that I used $G = 1$, and there were no other scaling units involved. We gave each particle a mass of 0.5, with a total mass of 1.0, which went into the `Body` description for the relative motion of the two particles.

Hey, wait a minute. We use a mass of 1 for our *relative* particle and a mass of 0.5 for each *individual* particle. There is a factor two between them, and two times two makes four.

Alice: Indeed, the factor four that your program was off with. And I think you found the solution, or at least the direction of the solution. Look, in your code you use the correct scaling for kinetic and potential energy, but you don't have any mass factors in there.

Bob: Perhaps I was thinking about the fact that we started with a total mass of one, or perhaps I just forgot. Can we correct that? The potential energy is

$$E_{pot} = -G\frac{M_1 M_2}{r} \tag{12.8}$$

For our case, $M_1 = M_2 = 1/2$, so neglecting the mass factors, I have overestimated the potential energy by a factor of four. The kinetic energy is

$$E_{kin} = \tfrac{1}{2}M_1 v_1^2 + \tfrac{1}{2}M_1 v_1^2 \tag{12.9}$$

In our equal mass case, the two velocities in the center of mass are each exactly half of the relative velocity, so their squares are four times smaller. The masses sum up to unity, so yes, I have overestimated the kinetic energy by the exact same value of four.

Alice: Problem solved.

Bob: Still, I wonder, I thought I had done something similar in another code, quite a while ago, and I think I did give that one considerable thought. The question is, should I do my energy diagnostics in the center of mass frame, or is there a way to save my current code?

What I mean is that the whole two-body problem is specified in terms of relative positions and relative velocities and the sum of the masses. In the equations of motions, nowhere do the individual velocities in the center of mass frame come in, nor do the individual masses appear. That makes me think that my mistake might not have been that bad after all. Could it be that I am *always* off by a constant factor, or at least by the same factor in potential and kinetic energy, so that it still makes sense to add the two and thus check for energy conservation?

Alice: What we have to do is to check how the reduced mass comes in.

Bob: Ah yes, that rings a bell, from my celestial mechanics class. The relative motion of two bodies under the influence of gravity, or of electrostatic forces for that matter, can be described by the equivalent motion of a pseudo-particle with a different mass, the reduced mass. But how did that go? We can look in any old celestial mechanics book, but it would be more fun to try to reconstruct it ourselves.

12.3 Reduced Mass

Alice: It can't be that hard. The potential energy is already given in terms of relative coordinates, also in the center-of-mass frame. It is

$$E_{pot} = -G\frac{M_1 M_2}{r} \tag{12.10}$$

What we have to do now is to rewrite the kinetic energy in relative coordinates. We know that in the center-of-mass frame the following lever-like relations hold:

$$v_1 = \frac{M_2}{M_1+M_2}v \qquad \text{and} \qquad v_2 = \frac{M_1}{M_1+M_2}v \tag{12.11}$$

where v is still the relative velocity between the two particles.

This gives us for the total kinetic energy, in the center of mass frame:

$$E_{kin} \quad = \quad \tfrac{1}{2}M_1 v_1^2 + \tfrac{1}{2}M_2 v_2^2$$

$$
\begin{aligned}
&= \tfrac{1}{2}\left[\frac{M_1M_2^2}{(M_1+M_2)^2}+\frac{M_2M_1^2}{(M_1+M_2)^2}\right]v^2 \\
&= \tfrac{1}{2}\frac{M_1M_2}{M_1+M_2}v^2
\end{aligned}
$$

The total energy can thus be written as

$$E = E_{kin} + E_{pot} = \frac{M_1M_2}{M_1+M_2}\left[\tfrac{1}{2}v^2 - \frac{M_1+M_2}{r}\right] \tag{12.12}$$

You were right about the reduced mass: this is defined for two particles as:

$$\mu = \frac{M_1M_2}{M_1+M_2} \tag{12.13}$$

It has the physical dimensions of mass, and for a light particle in orbit around a heavy particle, it reduces to the mass of the light particle. If $M_2 = 0.1M_1$, for example, $\mu = 0.09$, to within about ten percent the same as the mass of the light particle. Most of the relative motion also occurs in the motion of the light particle, so in the limit that the mass ratio grows even much larger, the relative motion becomes effectively that of the light particle around a fixed center of attraction.

Bob: And I can see now what I have done in my code: I have given the kinetic and potential energy per unit reduced mass. And I remember now why I have done that before, several years ago: there I had chosen units in which the total mass of the two-body system was unity. In that case what I did would have been correct. But now I should go back and include the $M_1 + M_2$ factor to my potential energy. If in the future we or someone else will use our code for a case where the sum of the masses is not unity, the code as it is will give a wrong answer.

I will rewrite the relevant code right away as follows:

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

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

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

Alice: I'm glad we checked not only that the code ran correctly, but that we had the right conversions between physical formula and expressions in the code.

Bob: Yes, this is something students always get entangled with when they start coding something themselves, and I can't really blame them, since I'm still making similar mistakes myself.

Alice: The nice thing about getting more experience is not so much that you stop making errors, it is more that you get better at spotting them, and then figuring out where they come from, what the wrong assumptions were that led you to the error in the first place. Once you are that far, correcting the error is generally quite simple.

12.4 Wrapping It Up

Bob: I think we have now collected enough material to get my students going for quite a while.

Alice: Shall we wrap it up, and write up what we have learned?

Bob: Yes, we can do that, but I have one worry. We are both quite happy with what we have learned now about Ruby, but there remains the fact that we cannot yet complete a simple Kepler orbit at reasonable precision without sitting here in front of our terminals, twiddling our thumbs waiting for the computer to return an answer. This will give people a bad impression about Ruby as being too slow for numerical applications.

Alice: You said that you can probably speed up the Ruby programs by two orders of magnitude.

Bob: I will try to do that soon, but I think I have a better idea. Our last program has become so structured now, that I think it will be easy to generalize it to higher order integrators, such as the second-order leapfrog, or even fourth order integrators. That should speed up our Kepler orbits quite a bit, and also help us to get beyond the single-precision level we are currently stuck with.

Alice: If you think you can do that with only minor modifications, that would be great, but I don't think you should start a whole new project. We should round off our current part, and present that to the students. We can learn from their reactions what is and is not clear, and then we can see better what to do next.

Bob: Yeah, I know, I have a tendency to just keep going on and on when I have fun tinkering with things. Okay, I promise you: I'll try to see whether I can splice a higher order integrator into our current code without changing more than a few dozen lines.

Alice: A few dozen? That still sounds like a lot, depending how large "a few" is in your dictionary. You're so good at compressing things into a few lines. What if I challenge you: can you introduce a second-order integrator, while adding or changing no more than only a dozen lines?

Bob: You are either joking or you have an unreasonably high opinion of my programming skills. I don't want to promise anything, since I think one dozen lines is plainly unrealistic.

Alice: I *was* joking. Clarity over brevity, definitely. But seeing how you were glowing over brevity, I just couldn't help myself.

Bob: But perhaps I can stay under two dozen.

Alice: I shouldn't have said anything. Anyway, we'll see tomorrow!

Chapter 13

Literature References

Programming Ruby – The Pragmatic Programmer's Guide, by Dave Thomas, with Chad Fowler and Andy Hunt, 2004 [Pragmatic Bookshelf].

This is a wonderful introduction to Ruby. There is a complete *online version*[1] available for the first edition of this book, but it is far better to read the second edition, which is available in both paper form and pdf form, from the *pragmatic programmer's*[2] web site.

[1] `http://www.rubycentral.com/book/index.html`
[2] `http://www.pragmaticprogrammer.com/titles/ruby/index.html`