*The Art of Computational Science*

*The Kali Code*

*vol. 14*

# A Matter of Speed:

# C Modules for Ruby

**Piet Hut and Jun Makino**

September 14, 2007

# Contents

# Preface

This volume will show several ways for speeding up Ruby code, by replacing a small amount of time-critical Ruby lines by short C functions. We have started writing some text here, but the bulk of the text will appear later, by late 2004 or early 2005.

## 0.1 Acknowledgments

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

Piet Hut and Jun Makino

Kyoto, July 2004

# Chapter 1

# Performance

## 1.1 A Matter of Speed

**Alice**: Now that we can build a real star cluster model, with our Plummer's model generator `mkplummer.rb`, we're getting closer do some real physics. I would love to use our N-body code to demonstrate gravitational thermodynamics effects, such as negative heat capacity.

**Bob**: Before we can do that, we'll have to speed up our Ruby code significantly! Right now, we don't stand a chance. We can play with a few particles, but there is no way we can handle even a few hundred particles, at this moment.

**Alice**: Is it that bad?

**Bob**: It is that bad, yes. Let's do a test, to see how long a single time step takes, say for 256 particles. I like to run tests with particle numbers that are powers of 2, to make it easier to compare timings between different runs.

**Alice**: Why don't we start more modestly, just to see whether the combination of our Plummer generator and our N-body code does what it is supposed to do, say for 8 particles.

**Bob**: Okay, better safe than sorry. Here we go, for one time unit. And I'll use a hefty dose of softening, just in case two of the particles happen to be born very close to each other. Before too long, we should introduce variable time steps, to handle close encounters. But until we can teach particles to shrink their time step when they meet each other, it is better to use an amount of softening that is an order of magnitude larger than the time step. In that way, not much can change in the gravitational potential during one time step.

**Alice**: As long as the velocities are of order one. In practice, particles will speed up when they come close together.

**Bob**: Yes, but only with the inverse square root of the distance. So if we use a softening length of, say, 0.01, we can probably still use a time step of 0.1 – but perhaps I'm too optimistic. Let's try it:

```
|gravity> ruby mkplummer.rb -n 8 | ruby nb1.rb -t 1 -d 0.01 -s 0.1
ruby: No such file or directory -- mkplummer.rb (LoadError)
/home/makino/papers/acs/lib/clop.rb:310: warning: already initialized constant HELP_DEFINITION_STRING
==> The simplest ACS N-body code <==
Integration method: rk4
Integration time step: dt = 0.01
Diagnostics output interval: dt_dia = 1.0
Snapshot output interval: dt_out = 1.0
Duration of the integration: dt_end = 1.0
Softening length: eps = 0.1
./nbody1.rb:167:in ‘/’: divided by 0 (ZeroDivisionError)
from ./nbody1.rb:167:in ‘write_diagnostics’
from ./nbody1.rb:86:in ‘evolve’
from nb1.rb:144
```

## 1.2 Energy Conservation

**Alice**: That's not bad, as far as energy conservation is concerned. I see that you used a random seed for the random number generator. Let's try another run, to see how large the variations in energy conservation are, for different Plummer's model realizations.

**Bob**: I prefer to suppress the snapshot output, now that we have seen that reasonable numbers are produced. I'll just ask for a snapshot output time that is longer than the run time:

```
|gravity> ruby mkplummer.rb -n 8 | ruby nb1.rb -t 1 -d 0.01 -s 0.1 -o2
ruby: No such file or directory -- mkplummer.rb (LoadError)
/home/makino/papers/acs/lib/clop.rb:310: warning: already initialized constant HELP_DEFINITION_STRING
==> The simplest ACS N-body code <==
Integration method: rk4
Integration time step: dt = 0.01
Diagnostics output interval: dt_dia = 1.0
Snapshot output interval: dt_out = 2.0
Duration of the integration: dt_end = 1.0
Softening length: eps = 0.1
./nbody1.rb:167:in ‘/’: divided by 0 (ZeroDivisionError)
from ./nbody1.rb:167:in ‘write_diagnostics’
from ./nbody1.rb:86:in ‘evolve’
from nb1.rb:144
```

---

**Alice**: Indeed, a different energy error. Let's try a few more.

**Bob**: Fine, but all that output is too much of a good thing, for my taste. Let me suppress the initial state echo. Since all that stuff appears on the standard error stream, I'll have to add a `&` symbol to a pipe, in order to get both streams through, the standard output and the standard error stream. I'll then just ask for the very last line with `tail -1` :

```
|gravity> ruby mkplummer.rb -n8 | ruby nb1.rb -t1 -d0.01 -s0.1 -o2 |& tail -1
ruby: No such file or directory -- mkplummer.rb (LoadError)
from nb1.rb:144
```

Oops, I forget that the `mkplummer` command generates its own initial state messages. But I can take care of that by wrapping both commands in a set of parentheses:

```
|gravity> (ruby mkplummer.rb -n8 | ruby nb1.rb -t1 -d0.01 -s0.1 -o2) |& tail -1
from nb1.rb:144
```

Much better!

**Alice**: Quite a bit of run-to-run variation of the energy errors. Can you try a few more?

**Bob**: Now it's easy:

```
|gravity> (ruby mkplummer.rb -n8 | ruby nb1.rb -t1 -d0.01 -s0.1 -o2) |& tail -1
from nb1.rb:144
|gravity> (ruby mkplummer.rb -n8 | ruby nb1.rb -t1 -d0.01 -s0.1 -o2) |& tail -1
from nb1.rb:144
|gravity> (ruby mkplummer.rb -n8 | ruby nb1.rb -t1 -d0.01 -s0.1 -o2) |& tail -1
from nb1.rb:144
|gravity> (ruby mkplummer.rb -n8 | ruby nb1.rb -t1 -d0.01 -s0.1 -o2) |& tail -1
from nb1.rb:144
|gravity> (ruby mkplummer.rb -n8 | ruby nb1.rb -t1 -d0.01 -s0.1 -o2) |& tail -1
from nb1.rb:144
```

**Alice**: Your abbreviation techniques do make it easier to see what is going on. In any case, a time step of 0.01 does not seem to large for a softening of 0.1

## 1.3 Timing

**Bob**: Let's create a standard input file, so that we can do some timings.

**Alice**: Timing should be independent of energy error, since the number of integration steps are the same for different Plummer realizations.

**Bob**: That's true, but I like to make the command line a bit shorter. Here is one standard input file, for 8 particles:

```
|gravity> ruby mkplummer.rb -n 8 -s 42 > plum8.in
ruby: No such file or directory -- mkplummer.rb (LoadError)
```

Let me try to obtain some timing information:

```
|gravity> time ruby nb1.rb -t1 -d0.01 -s0.1 -o2 < plum8.in
/home/makino/papers/acs/lib/clop.rb:310: warning: already initialized constant HELP_DEFINITION_STRING
==> The simplest ACS N-body code <==
Integration method: rk4
Integration time step: dt = 0.01
Diagnostics output interval: dt_dia = 1.0
Snapshot output interval: dt_out = 2.0
Duration of the integration: dt_end = 1.0
Softening length: eps = 0.1
./nbody1.rb:167:in ‘/’: divided by 0 (ZeroDivisionError)
from ./nbody1.rb:167:in ‘write_diagnostics’
from ./nbody1.rb:86:in ‘evolve’
from nb1.rb:144
0.035u 0.004s 0:00.04 75.0%0+0k 0+0io 0pf+0w
```

Again, I prefer to suppress most of the information here:

```
|gravity> time ruby nb1.rb -t1 -d0.01 -s0.1 -o2 < plum8.in |& tail -2
from ./nbody1.rb:86:in ‘evolve’
from nb1.rb:144
```

Ah, that threw away too much: the timing information went down the drain.

**Alice**: Perhaps a matter of wrapping it up within parentheses again?

**Bob**: That may wel work:

```
|gravity> (time ruby nb1.rb -t1 -d0.01 -s0.1 -o2 < plum8.in) |& tail -2
from nb1.rb:144
0.034u 0.004s 0:00.03 100.0%0+0k 0+0io 0pf+0w
```

**Alice**: And so it does.

## 1.4 The Next Step

**Bob**: So now that we know that things are under control for $N = 8$, let us try the $N = 256$ that I suggested earlier.

**Alice**: How much longer would that take? We increase the number of particles by a factor 32, which means that the number of particle-particle interactions increase by a factor 1024, the square of 32.

This estimate is not exact, since we don't have self-interactions between particles, so we should compare $N(N-1)$, which gives us a factor of $256(256-1)/8(8-1) \approx 1166$. And besides, there is the overhead of reading in the particles, printing them out, and various other overhead that is linear, rather than quadratic in $N$.

But roughly speaking, I would guess your $N = 256$ run to take a factor of a thousand more time. So far, we have followed the $N = 8$ run for 100 time steps. So a single time step for a $N = 256$ run should take about ten times longer than our $N = 8$ runs.

**Bob**: That must be about right. Here is an input file with 256 particles:

```
|gravity> ruby mkplummer.rb -n 256 -s 137 > plum256.in
ruby: No such file or directory -- mkplummer.rb (LoadError)
```

And here is how long a single time step takes:

```
|gravity> (time ruby nb1.rb -t0.01 -d0.01 -e0.01 -s0.1 -o2 < plum256.in) |& tail -2
from nb1.rb:144
0.023u 0.003s 0:00.02 100.0%0+0k 0+0io 0pf+0w
```

**Alice**: A bit longer than I had guessed.

**Bob**: But of course, there may be considerable start-up time, for loading the particles. And, ah, don't forget that we determine the total energy at startup, which also takes a significant amount of time.

Therefore, if we take two time steps, that should take significantly less than twice the time for one time step:

```
|gravity> (time ruby nb1.rb -t0.02 -d0.01 -e0.01 -s0.1 -o2 < plum256.in) |& tail -2
from nb1.rb:144
0.025u 0.000s 0:00.02 100.0%0+0k 0+0io 0pf+0w
```

**Alice**: You are right. Let's do three time steps, to see whether each subsequent time step, after the first one, takes roughly the same amount of time.

```
|gravity> (time ruby nb1.rb -t0.03 -d0.01 -e0.01 -s0.1 -o2 < plum256.in) |& tail -2
from nb1.rb:144
0.039u 0.001s 0:00.05 60.0%0+0k 0+0io 0pf+0w
```

**Bob**: To a good approximation, yes. And indeed, this shows us that a single time step for 256 particles does take about ten times longer than a run of 100 time steps took for 8 particles.

## 1.5 Two Orders of Magnitude Speedup?

**Alice**: Yes, it is clear that we will have to do something about the speed of our code. This is not going to make it easy to do thermodynamic experiments with 256 particles.

**Bob**: To put it mildly! If we want to have some fun running a 256-particle system for several relaxation times, in other words, for a few dozen crossing times, we would need to run the simulation for, say, thirty of forty time units. A hundred time units would be even better.

With a softening length of 0.1, that would already require 10,000 time steps. And if we take a smaller softening length, of 0.2, say, encounter velocities between particles go up, so we would probably need more like 100,000 time steps.

**Alice**: In other words, for every second that it takes to complete one time step, for $N = 256$, it would take a day to run a decent experiment. No, we can't wait that long.

**Bob**: Even a speed-up of an order of magnitude wouldn't help enough. We need at least two orders of magnitude: anything less than a factor of 100 speedup just wouldn't cut the cake. At the very least I want to be able to run several experiments in one day.

There are two things we can do. First of all, we can make our Ruby code itself faster. So far, we haven't been careful at all about speed. We even compute the inter-particle accelerations in $N(N-1)$ fashion. We can already gain almost a factor of two by computing the acceleration of particle $i$ by particle $j$ at the same time that we compute the acceleration of particle $j$ by particle $i$ since the intermediate steps are the same. This will reduce the number of square root calls from $N(N-1)$ to $\frac{1}{2}N(N-1)$ per time step. Since square roots are more expensive than additions and multiplications, this is bound to help.

The second thing we can do is to write a version of the most compute-intensive inner loop in a faster language. Since Ruby is written in C, it would be natural to write a C version for the pairwise acceleration calls, and to link that with our Ruby code. That should make a significant difference.

**Alice**: The first step may buy you a factor two, at most, and perhaps you can make some other improvements that also give you a factor two. The way we pass strings around to allow us to use arbitrary algorithms is probably not the most efficient way to integrate the equations of motion. Finetuning may give us another factor of two, if we are lucky. But that wouldn't get us anywhere close to the factor of 100 that you would like to see.

**Bob**: Indeed. Most of the speedup will have to come from using C as a form of assembly language.

**Alice**: Before launching into a long quest for speed-up, how about writing a simple test file, one which does a lot of floating point operations, in both Ruby and C? That will give us an upper limit to the amount of speedup we might get from incorporating C into Ruby.

# Chapter 2

# Ruby versus C

## 2.1 Raw Floating Point Speed

**Bob**: Yes, I would like to see the difference in raw speed between Ruby and C, for a few simple tasks. How about this simple Ruby script:

```
include Math

N = 1000000

a = 1.0

N.times{a = sqrt( (a * (a + 1))/(a + 0.001*a) )}

print "N = #{N} ; a = #{a}\n"
```

I will just let it do a million interations of some floating point calculations.

Here is the corresponding C program. Since C is bound to be a lot faster than Ruby, we may as well give C a factor of ten handicap: ten million iterations it will be for the C code:

```
#define  N  10000000

int main()
{
    int i;
    double a;
```

```
    a = 1.0;
    for (i = 0; i < N; i++)
        a = sqrt( (a * (a + 1))/(a + 0.001*a) );

    printf("N = %d ; a = %g\n", N, a);
}
```

I'll start and see how the C code fares:

```
|gravity> time test1
test1: Command not found.
0.000u 0.000s 0:00.00 0.0%0+0k 0+0io 0pf+0w
```

## 2.2 Slooow

**Alice**: how about compiling it first?

**Bob**: Ah, of course, in C you have to *compile* things first. I've gotten spoiled, using so much Ruby! Okay, I'll compile it without any optimizer, since who knows what that will do, in terms of corner cutting in such a simple expression.

```
|gravity> gcc test1.c -lm -o test1
```

And now we can see what C delivers:

```
|gravity> time test1
N = 10000000 ; a = 1.61686
1.785u 0.001s 0:01.89 94.1%0+0k 0+0io 0pf+0w
```

And here is the Ruby counterpart:

```
|gravity> time ruby test1.rb
N = 1000000 ; a = 1.61686424868974
4.664u 0.002s 0:05.10 91.3%0+0k 0+0io 0pf+0w
```

**Alice**: At least both give us the same output. But boy, is Ruby slow: even with the handicap of a factor ten, C wins hands down. Ruby must be more than twenty times slower then C.

But wait a minute, why would we get the same output? The C code did ten times more work.

**Bob**: I just threw in some random floating point operations. We must be converging to some limit point.

**Alice**: Might be. But let me check whether that is reasonable. I don't like a situation where I don't know what's going on, numerically. If we forget about that factor 0.001 that you threw is as well, we basically have $a = \sqrt{a+1}$, or in other words $a^2 = a + 1$ or $a^2 - a - 1 = 0$. I remember even high school math to solve that one: $a = \frac{1}{2}(1 + \sqrt{5})$. So yes, we do seem to reach a limit point. Let's check its value:

```
|gravity> irb
irb(main):001:0> include Math
=> Object
irb(main):002:0> 0.5*(1+sqrt(5))
=> 1.61803398874989
```

Close to what we found; your factor 0.001 must make the difference. Good! I'm happy.

## 2.3 At Least Not Slower

**Bob**: You may be happy with the value of the calculation, but I'm far from happy with the speed. And we're mainly comparing floating point calculations here. It could be worse once we include function calls and other overhead. Let's do the same thing through a function call.

Here is the Ruby version:

```
include Math

N = 1000000

a = 1.0

def new_a(old_a)
  return sqrt( (old_a * (old_a + 1))/(old_a + 0.001*old_a) )
end

N.times{a = new_a(a)}

print "N = #{N} ; a = #{a}\n"
```

And here is the C version:

```
#define  N  10000000

double new_a(double old_a)
{
    return sqrt( (old_a * (old_a + 1))/(old_a + 0.001*old_a) );
}

int main()
{
    int i;
    double a;

    a = 1.0;

    for (i = 0; i < N; i++)
        a = new_a(a);

    printf("N = %d ; a = %g\n", N, a);
}
```

I'll start again with the C code. And this time I'll compile it first:

```
|gravity> gcc test2.c -lm -o test1
```

Let's see:

```
|gravity> time test2
test2: .
0.000u 0.000s 0:00.00 0.0%0+0k 0+0io 0pf+0w
```

And here is what Ruby delivers

```
|gravity> time ruby test2.rb
N = 1000000 ; a = 1.61686424868974
5.786u 0.001s 0:06.29 91.8%0+0k 0+0io 0pf+0w
```

**Alice**: That didn't make much difference. It seems that Ruby is slower than C by a bit more than a factor twenty.

## 2.4 Not So Quick

**Bob**: I would not be so quick in jumping to conclusions. A real N-body code does a lot more than floating point calculations and a few function calls. Think about the vector notation we introduced in Ruby, through the `Vector` class that we built on top of the `Array` class. I bet that will slow things down.

**Alice**: The only way to measure a *really* realistic speed difference between Ruby and C would be to rewrite a whole N-body code from Ruby into C.

**Bob**: Perhaps. But I think we can get a good stab at the speed difference if we simulate just a small part of the work done during one time step, in terms of the pairwise force calculations between particles. As long as we do a double loop over a large array of particles, each of which has at least one of our `Vector` vectors, and then do some kind of `Vector` operation. How about this, inspired by the first step in a real pairwise gravity calculation:

```
require "vector.rb"

include Math

NDIM = 3
EPS = 0.000001

n = gets.to_i

r = Array.new(n)
r.each_index do |i|
  v = Vector.new(NDIM)
  (0...NDIM).each{|k| v[k] = (i * NDIM + k) * EPS}
  r[i] = v
end

sum = 0
r.each_index do |i|
  (i+1...n).each do |j|
    rji = r[j]-r[i]
    sum += rji*rji
  end
end

print "n = #{n} ; sum = #{sum}\n"
```

**Alice**: Good idea. You are writing in the number of particles, before giving each a three-dimensional vector with some components which are all chosen differently, so that you can safely subtract vectors later on. Then you go through a double loop to calculate the difference vector, the distance between two particles, and from that vector you compute the inner product with itself. Finally you print something out, to see whether you'll get the same value in C as in Ruby.

**Bob**: You may not be able to read my mind, but at least you can read my code. Yes, that was exactly my intention.

## 2.5 Wanna Bet?

**Alice**: Do you remember how to write all this in C?

**Bob**: As long as I keep telling myself to put semicolons at the end of every line, it shouldn't be too bad. How about this:

---

```
#define  NMAX  100000
#define  NDIM  3
#define  EPS  0.000001

int main()
{
    int i, j, k, n;
    double r[NMAX][NDIM];
    double sum;
    double rji[NDIM];
    double r2;

    scanf("%d", &n);

    for (i = 0; i < n; i++)
for (k = 0; k < NDIM; k++)
    r[i][k] = 1.0 + (i * NDIM + k) * EPS;

    sum = 0;
    for (i = 0; i < n; i++){
        for (j = i+1; j < n; j++){
            for (k = 0; k < 3; k++)
                rji[k] = r[j][k] - r[i][k];
    r2 = 0;
            for (k = 0; k < NDIM; k++)
                r2 += rji[k] * rji[k];
    sum += r2;
}
    }
    printf("n = %d ; sum = %.10g\n", n, sum);
```

```
}
```

---

**Alice**: Looks fine. Does it compile?

---

```
|gravity> gcc test3.c -lm -o test3
```

---

**Bob**: It does. Now, before I run both programs, what do you expect to see, for a slowness ratio of Ruby over C?

**Alice**: The simplest guess would be another factor between twenty and thirty.

**Bob**: I think it may be quite a bit larger, with all that vector stuff.

**Alice**: Well, okay, I will predict a factor 35, but that may be an overstatement.

**Bob**: Wanna bet?

**Alice**: I don't like to bet, but tell me, what do you expect?

**Bob**: I think vectors will slow things down by at least a factor two. I'll put my bets on a factor 50.

**Alice**: May the best predictor win!

## 2.6 Surprise

Let's start with Ruby first, this time:

---

```
|gravity> time ruby test3.rb
1000
n = 1000 ; sum = 2.2499977500002
15.568u 0.008s 0:16.96 91.7%0+0k 0+0io 0pf+0w
```

---

And why not give C the same number of particles? At least we can see whether we get the same result:

---

```
|gravity> time test3
1000
n = 1000 ; sum = 2.24999775
0.065u 0.000s 0:00.08 75.0%0+0k 0+0io 0pf+0w
```

**Alice**: Now *that* is fast! Very much faster even than your prediction of a factor 50 speedup. And we do get the same value out, so I guess both codes are really doing the same thing.

**Bob**: Let's make life a factor hundred more difficult for the C version. Giving ten times more particles should do the job:

```
|gravity> time test3
10000
n = 10000 ; sum = 22499.99978
4.595u 0.001s 0:06.68 68.7%0+0k 0+0io 0pf+0w
```

**Alice**: Still C is faster than Ruby, with a hundred times more work. At least, if the work is really quadratic in the particle number.

**Bob**: It should be. Even so, it is easy to check! We can double the number for Ruby:

```
|gravity> time ruby test3.rb
2000
n = 2000 ; sum = 35.9999909999736
64.641u 0.011s 1:10.51 91.6%0+0k 0+0io 0pf+0w
```

and for C as well:

```
|gravity> time test3
20000
n = 20000 ; sum = 359999.9991
20.092u 0.007s 0:21.82 92.0%0+0k 0+0io 0pf+0w
```

**Alice**: And everything takes about four times as long. Okay, I'm convinced. For this task at least, Ruby seems to be a whopping factor of two hundred slower than C.

**Bob**: I must say, I'm surprised.

**Alice**: At least you are a better predictor than I am.

**Bob**: That may not be such a surprise, since you generally play the role of corrector!

## 2.7 Wonderful

**Alice**: Well, somebody has to. But this factor of 200 is shocking. Not only that, it is probably more relevant for our N-body calculations than the first two tests we did, as you already stressed. This is pretty terrible.

**Bob**: It is wonderful!

**Alice**: I beg your pardon?

**Bob**: It's wonderful to have a factor of 200 to play with. Remember that I had asked for a factor 100 improvement in speed? We can actually do it now! We only need to determine which part of the code does at least half of the work, and replace that by C code. We can then leave the rest in Ruby, and by getting a speed-up of a factor 200 for half the work, the total speed of the code will increase by a factor 100.

In general, my experience with N-body codes has shown me that for an N-body integrator much more than half of the computer time is spent in just a few lines of code. It is the innermost loop where the gravitational accelerations are computed between pairs of particles, that takes almost all the time.

So the conclusion is: we should be able to speed up our Ruby codes by a factor of 100.

**Alice**: That all sounds a bit too optimistic to me, but I see the logic of what you say. Well, let's try it out and see how far we get!

# Chapter 3

# Tuning Ruby

## 3.1 Pairing Pairwise Calculations

**Bob**: Before we start adding pieces of C code to our Ruby code, let us first see whether we can improve the speed of our N-body code purely within the Ruby realm. As we discussed before, that should be the first step, even though the second step, adding C, will buy us more in the end. We may as well get extra speed from wherever we can.

**Alice**: So this means avoiding double work in pairwise interactions.

**Bob**: Exactly. What we have been doing so far was to let each particle determine the acceleration it feels by interrogating all other particles. However, the work done in calculating the gravitational acceleration from particle $j$ on particle $i$:

$$\mathbf{a}_i(r_i, r_j) \;=\; G\,\frac{m_j}{|\mathbf{r}_i - \mathbf{r}_j|^3}\,(\mathbf{r}_j - \mathbf{r}_i) \tag{3.1}$$

overlaps a lot with the work done in calculating the gravitational acceleration from particle $i$ on particle $j$:

$$\mathbf{a}_j(r_j, r_i) \;=\; G\,\frac{m_i}{|\mathbf{r}_i - \mathbf{r}_j|^3}\,(\mathbf{r}_i - \mathbf{r}_j) \tag{3.2}$$

Subtracting the two position vectors, and determining the third power of the magnitude of the difference is what takes most of the computer time. The minus sign and the different mass multiplication factor is peanuts, in comparison.

**Alice**: So you suggest rewriting our N-body code in such a way as to allow both calculations to be done simultaneously.

**Bob**: Exactly.

**Alice**: That may not be so easy. Also, it may not be so pretty. I like the modularity of our code, where each particle has its own job to do in finding out how to determine the total gravitational acceleration it feels from all other particles. I don't like the idea of messing up everything by crossing levels of command.

**Bob**: And I don't like codes that are unnecessarily slow! At the very least we have to try and find out. Only when we see what it really looks like, and how much speed increase we really get, can we decide whether the increas of speed is worth the decrease in prettiness, however you may want to define that.

**Alice**: Fair enough.

## 3.2 A Need for Library Structure

**Bob**: I'll try to be careful with giving names to directories and files. I must admit, I'm getting a bit confused with all the different versions of N-body codes we now have lying around.

**Alice**: We should soon decide upon a library structure, where we can store those versions we are really happy with.

**Bob**: Yes. Now that we have a rather versatile N-body code, a general command line argument interpreter, and a generator for Plummer's model realizations, we are beginning to put together a real N-body toolbox.

Well, one thing at a time. It is crucial that we get good speed out of Ruby first. And in order to keep track of our N-body versions, here is the file with the definitons of the `Body` and `Nbody` classes, which we called `rknbody.rb`, after we had finished our command line interpreter, in a directory I called `vol-4`. Our Plummer's model building versions live in directory `vol-5`, so let me put a copy of `vol-4/rknbody.rb` in the current directory `vol-6`, and let me call it `nbody1.rb`, for short, with the `1` indicating that we'll probably make a few different versions, for our various attempts at speedup.

Similarly, let me copy the driver from `vol-4/rkn2.rb` to the current directory, and call it `vol-6/nb1.rb`, with the one difference of course that the first line is no longer

```
require "rknbody.rb"
```

but

```
require "nbody1.rb"
```

since we just changed the name of that file, even though the contents are exactly the same.

**Alice**: You'd better keep some good notes! It is high time not only for making a good library structure, but also for organizing our notes. How are we ever going to present this to our students otherwise?

**Bob**: Yes, and we were going to define a good data format, remember, and there is graphics, something we talked about many times, but never got around too. People have no idea how much work it is to build a good foundation for a software project. They think that an N-body code is just integrating Newton's equations of motion, well, how complex can that bed.

**Alice**: An understandable misunderstanding. Well, let's hope we can get that misunderstanding out of the way, when we get our act together.

**Bob**: And no act without speedup. Here we go! I'm copying `nbody1.rb` now to `nbody2.rb`, and in parallel I'm copying `nb1.rb` to `nb2.rb`, with one distinction . . .

**Alice**: . . . again the first line.

**Bob**: Yes: instead of

```
require "nbody1.rb"
```

in `nb1.rb`, it now reads

```
require "nbody2.rb"
```

`nb2.rb`. I agree, there ought to be a better way. And I'm sure there is. But first: speed!

**Alice**: No stopping you at this point!

**Bob**: I hope not. Give me a few minutes, and let me see how I can implement a more economic pairwise acceleration calculation.

## 3.3 From `Body` to `Nbody`

**Alice**: Hi Bob! How's your economic progress?

**Bob**: Fairly well, I think I just got it working. After scratching my head for a bit, I realized that I had to move the method `acc`, to calculate the acceleration between particles, from the `Body` to the `Nbody` class.

**Alice**: That's interesting, and a big change. But now that you mention it, yes, of course: a single body can only care about it's own business. It can calculate its own acceleration, but it wouldn't be able to help others.

**Bob**: Unless it had double pointers. Remember my first attempt at writing an N-body code, way back after we had finished playing with our 2-body version?

You didn't like my backward pointers, but they would have enabled one particle to talk directly to another particle.

**Alice**: Directly, you say? You mean by pointing back to the parent `Nbody` class instance, and from ther to another particle, crossing boundaries twice! If you can that direct, that you may as well flatten the whole organization of the code into one big fat pancake . . .

**Bob**: . . . how can the pancake be fat and flat?

**Alice**: At partial *attempt* at flattening can still look wobbly and fat. But my point is: such an approach would go against any attempt at modularity.

**Bob**: Okay, okay, I wasn't proposing to go back to that idea. For one thing, I didn't look forward to having to argue with you about double crossing pointers.

**Alice**: So you decided to lift the acceleration calculation from the `Body` class to the `Nbody` class.

**Bob**: Yes. I've called it `nb_acc` now, instead of `acc`, and it got a bit more complex, but not that much. Let me show them side by side. Here is the old `Body#acc`, a somewhat confusing Ruby notation meaning the `acc` method belonging to the class `Body`, but we may as well get used to it, since it is in general use in the Ruby community.

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

And here is `Nbody#nb_acc`, my new variation:

```
def nb_acc
  null_vector = @body[0].pos*0
  @body.each{|bi| bi.acc = null_vector}
  @body.each_index do |i|
    bi = @body[i]
    (i+1...@body.size).each do |j|
        bj = @body[j]
        r = bj.pos - bi.pos
```

```
          r2 = r*r + @eps*@eps
          r3 = r2*sqrt(r2)
          bj.acc -= r*(bi.mass/r3)
          bi.acc += r*(bj.mass/r3)
      end
    end
  end
```

## 3.4 Looping Options

**Alice**: That's not much longer than what we had before. I see that you start out by setting the acceleration to zero for each particle. This means that every body now has an instance variable `@acc`, on the same level as `@pos` and `@vel`, I take it?

**Bob**: Yes, and before doing any acceleration calculation, each particle's `@acc` has to be set to zero, so that it can accumulate contributions. A particle cannot delay this action until it starts to calculate its own accelerations, since before doing so, it may already receive contributions from other particles, as side effects of *their* calculations.

**Alice**: Then you enter into a double loop over particles, something that you would normally code in an double for loop, using the traditional `i` and `j` variables.

**Bob**: Yes, just as in the C test program, where we used:

```
for (i = 0; i < n; i++)
    for (j = i+1; j < n; j++)
```

**Alice**: It's interesting to see the options that Ruby offers. You could have used `for` loops here too, of course.

**Bob**: Yes, I could have used

```
for i in 0...@body.size
  for j in i+1...@body.size
```

instead of

```
@body.each_index do |i|
  (i+1...@body.size).each do |j|
```

I guess I'm growing fond of the `each` method that lets Ruby do the counting, rather than having to worry explicitly about reminding myself and the computer how long an array is, where to stop, and whether or not to include the upper bound, using `..` notation, or leave that one out, using `...`.

**Alice**: Remind me, what is the difference?

**Bob**: `1...3` counts over `1,2` only, while `1..3` counts over `1,2,3`. The more dots, the fewer points. I guess it the `..` notation was chosen first as the most obvious interpretation, and then `...` was added as a practical after thought, since it happens so often that we count through an array, starting at zero and taking N terms, which means that we have to count up to but excluding the Nth element.

**Alice**: You could have avoided mentioning the size of the array by writing:

```
@body.each_index do |i|
  @body.each_index do |j|
    if j > i
```

**Bob**: True, but since our aim is to speed up our calculations, I was afraid that would get unnecessary overhead. Well, we can check later, when we do our timings.

**Alice**: The rest of `nb_acc` is straightforward: after calculating the usual `r` vector and `r3` scalar values, you now get two accelerations for the price of one. Can you show me what else you had to change in the program?

## 3.5 The `calc` Methods

**Bob**: I already mentioned the addition of an extra `Body` variable:

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

And the `Body#calc` method has simplified a lot. It used to be:

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

but in the new version it has become:

```
def calc(time_step, s)
  dt = time_step
  eval(s)
end
```

The reason for this trimming down is that all the hard work of acceleration calculations are now done on the `Nbody` level. The only job left to do on the `Body` level is to add, subtract terms containing `@pos` and `@vel` and `@acc`, and multiply those with coefficients and powers of the time step `dt`.

Consequently, the `Nbody#calc` method got trimmed down as well. Instead of:

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

we now have:

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

## 3.6 Integrators

**Alice**: Can you show me how this affect a simple integrator, such as our forward Euler method?

**Bob**: It used to be:

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

while now we have:

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

You see, the line containing the acceleration is now much simpler, and the whole expression is more homogeneous – but of course you first have to give the instruction to do the global acceleration calculation, something that `nb_acc` takes care off.

**Alice**: And the other methods are affected similarly.

**Bob**: Yes, but with one exception. The two Runge Kutta methods are changed the way would expect them to change. Here is the second-order one:

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

and here is the fourth-order version:

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

## 3.7 The Leapfrog

The exception comes in with the `leapfrog` method. I could have made a similarly straightforward translation from the old version:

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

But I did not like to ask the whole system to calculate all accelerations twice. You see, at the end of each loop, we change only the velocity. This means that the acceleration calculation at the beginning of the next step repeats exactly the same calculation as we already did at the end of the previous, since the acceleration is only dependent on the positions, not on the velocity. If we are interested in speed-up, there is another potential factor of two in speed that we can gain.

**Alice**: Let me try to remember, there must have been a reason that we wrote it that way in the first place.

**Bob**: Yes, there was. You can't skip the second call to `acc`, since otherwise `@vel` would not be properly updated at the end of the step. But you can't skip the first call either, for two reasons.

The first reason has to do with start-up. When you take the very first step, there is no previous information yet, so you just *have* to calculate the acceleration at the beginning of the loop, in order to step the position forward.

The second reason is connected with our previous approach of letting each particle calculate its own acceleration, whenever needed, without introducing extra variables unless we needed to do so. That made sense, since we were aiming at clarity and brevity, rather than speed. But now we have to reconsider those choices.

If we were to speed up the old approach, we would have to do two things: acquire the initial acceleration in a special move at the start of an integration, and introduce an extra variable `@old_acc`. However, in our new approach, where we determine the integration on the `Nbody` level for all particles at once, we already have the acceleration avaiable immediately after a call to `nb_acc`, for each particle in the variable `@acc`. So the only thing left to do is to warm up the engine before getting into an integration loop.

Here is how I implemented this:

```
def leapfrog
  if @init_flag
    nb_acc
    @init_flag = false
  end
  calc(" @vel += @acc*0.5*dt ")
  calc(" @pos += @vel*dt ")
```

```
    nb_acc
    calc(" @vel += @acc*0.5*dt ")
  end
```

The flag `init_flag` tells you whether the acceleration variables `@acc` have to be initialized. And that flag is set, you guessed it, in the initializer for the `Nbody` system, since it is within the `Nbody` class that the `leapfrog` method is used:

```
  def initialize
    @body = []
    @init_flag = true
  end
```

**Alice**: And in this way the initial `nb_acc` is invoked only one time. That's a nice solution, which hardly complicates the algorithm. And it is impressive that for the leapfrog we now have a factor four in speed-up, at least with respect to the calculation of pairwise gravitational attractions: we only visit each particle pair once, instead of twice; and apart from the first round, each time the frog leaps it calculations all accelarations only once. Great!

**Bob**: And that's it! No further changes needed.

**Alice**: You did not touch the calculation of the potential energy? That is the only other place where there are operations that scale with the square of the particle number.

**Bob**: One thing at a time. Normally we don't calculate the energy of the system at every time step, but only when we ask for a diagnostics output. Even if we lose a factor of two or four there, it will really make no difference in the total speed.

This may chance when we start using C modules. If we really can get a speedup of a factor 100 there, we may well have to revisit the energy calculation too. Even if we calculate the energy, say, once every thousand time steps, carelessness there could cost us a few tens of percent in total speed.

## 3.8 Brevity

**Alice**: I have one more suggestion for a change. Nothing to do with speedup, only with making things look prettier. I bet you'll like it.

**Bob**: What do you have in mind?

**Alice**: Remember that you tried so hard to make the individual lines of the integrators look as simple as they did, back in the days that we were working on the two-body problem? Well, now that you have brought the acceleration variables in line with the position and velocity variables, we can finally grant yuor wish!

Let me try a bit of regular expression magic. May I?

**Bob**: Sure you may! As you know, I *love* brevity. But let me call the new version `nbody3.rb`, to keep our versions separate. Here is the keyboard.

**Alice**: It is the `Nbody#calc` that I would like to make just a bit more smarter. In your last version it looked like this:

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

Are you ready for this? Here is a `calc` on steroids:

```
def calc(s)
  @body.each{|b| b.calc(@dt, s.gsub(/([a-z]\w*)/, '@\&').gsub(/@dt/, 'dt'))}
end
```

**Bob**: Ah, good old regular expressions! Let me see. Everywhere in the string `s` you do two global substitutions, using `gsub`. First you take any substring that starts with a lower case letter, followed by an arbitrary number of alphanumeric characters – the name of a variable, I take it.

**Alice**: Precisely.

**Bob**: Then you take that name, and you add a `@` symbol in front it in, because `\&` just echoes the previous match of what was found in between the parentheses there, namely the variable name.

Ah, I get it! You add all those annoying `@` signs that are needed to tell Ruby that we are dealing with instance variables. In that way we don't need to add those to the code of the integrators. Alice, you're a genius!

**Alice**: maa, nee.

**Bob**: What does that mean?

**Alice**: Oh, I guess I didn't tell you that I started to take some Japanese classes, just for fun. Occasionally I slip into classroom mode. Just ignore that.

**Bob**: But then there is a second global substitution. Why that? Let's see. Wherever you encounter an expression `@dt`, you replace it by `dt`. Ah, of course. When you give *every* variable an `@` sign, it is all nice and well for `pos` to turn into `@pos`, and so on, but you will also turn `dt` into `@dt`, and that is too much of a good thing. Got it!

May I rewrite the integrators? That will be fun!

**Alice**: Go right ahead!

**Bob**: Let's see, almost a global replace of `@` by nothing, except for the `@init_flag` in `leapfrog`, almost trimmed that one too, by mistake. Ah, that looks wonderful. Just look at the fourth-order Runge Kutta:

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

What a beauty!

**Alice**: Glad you like it!

# Chapter 4

# XX

## 4.1 Testing

**Bob**: And now it is time for Alice to say that it is time to test our codes, to see whether they still all give the same output.

**Alice**: Let it be said.

**Bob**: Three particles should be enough to check:

```
|gravity> ruby mkplummer.rb -n 3 -s 33 > plum3.in
ruby: No such file or directory -- mkplummer.rb (LoadError)
```

We'll first run the old version:

```
|gravity> time ruby nb1.rb -t1 -d0.01 -s0.1 < plum3.in
/home/makino/papers/acs/lib/clop.rb:310: warning: already initialized constant HELP_DEFINITION_STRING
==> The simplest ACS N-body code <==
Integration method: rk4
Integration time step: dt = 0.01
Diagnostics output interval: dt_dia = 1.0
Snapshot output interval: dt_out = 1.0
Duration of the integration: dt_end = 1.0
Softening length: eps = 0.1
./nbody1.rb:167:in ‘/’: divided by 0 (ZeroDivisionError)
from ./nbody1.rb:167:in ‘write_diagnostics’
from ./nbody1.rb:86:in ‘evolve’
from nb1.rb:144
0.023u 0.003s 0:00.02 100.0%0+0k 0+0io 0pf+0w
```

---

Here is my newer and hopefully faster version:

```
|gravity> time ruby nb2.rb -t1 -d0.01 -s0.1 < plum3.in
/home/makino/papers/acs/lib/clop.rb:310: warning: already initialized constant HELP_DEFINITION_STRING
==> The simplest ACS N-body code <==
Integration method: rk4
Integration time step: dt = 0.01
Diagnostics output interval: dt_dia = 1.0
Snapshot output interval: dt_out = 1.0
Duration of the integration: dt_end = 1.0
Softening length: eps = 0.1
./nbody2.rb:180:in ‘/’: divided by 0 (ZeroDivisionError)
from ./nbody2.rb:180:in ‘write_diagnostics’
from ./nbody2.rb:88:in ‘evolve’
from nb2.rb:144
0.037u 0.002s 0:00.03 100.0%0+0k 0+0io 0pf+0w
```

That looks good, no real changes here. And yes, it is faster, but not by as much as I had hoped.

Here is your prettified brevity-rules version:

```
|gravity> time ruby nb3.rb -t1 -d0.01 -s0.1 < plum3.in
/home/makino/papers/acs/lib/clop.rb:310: warning: already initialized constant HELP_DEFINITION_STRING
==> The simplest ACS N-body code <==
Integration method: rk4
Integration time step: dt = 0.01
Diagnostics output interval: dt_dia = 1.0
Snapshot output interval: dt_out = 1.0
Duration of the integration: dt_end = 1.0
Softening length: eps = 0.1
./nbody3.rb:180:in ‘/’: divided by 0 (ZeroDivisionError)
from ./nbody3.rb:180:in ‘write_diagnostics’
from ./nbody3.rb:88:in ‘evolve’
from nb3.rb:144
0.022u 0.004s 0:00.03 66.6%0+0k 0+0io 0pf+0w
```

**Alice**: Also no surprises. Good! The same results in all three cases. I'm happy.

## 4.2 Little Speedup, So Far

**Bob**: But I'm not happy about the small speedup. Maybe having three particles is too small a number. Let's revisit the one time step that we did with 256 particles. Or better both one and two time steps, given that the first time step had some extra overhead.

With the earlier version we had:

```
|gravity> (time ruby nb1.rb -t0.01 -d0.01 -e0.01 -s0.1 -o2 < plum256.in) |& tail -2
from nb1.rb:144
0.026u 0.000s 0:00.02 100.0%0+0k 0+0io 0pf+0w
```

and

```
|gravity> (time ruby nb1.rb -t0.02 -d0.01 -e0.01 -s0.1 -o2 < plum256.in) |& tail -2
from nb1.rb:144
0.022u 0.003s 0:00.02 100.0%0+0k 0+0io 0pf+0w
```

With our latest prettified version we have

```
|gravity> (time ruby nb3.rb -t0.01 -d0.01 -e0.01 -s0.1 -o2 < plum256.in) |& tail -2
from nb3.rb:144
0.037u 0.002s 0:00.04 75.0%0+0k 0+0io 0pf+0w
```

and

```
|gravity> (time ruby nb3.rb -t0.02 -d0.01 -e0.01 -s0.1 -o2 < plum256.in) |& tail -2
from nb3.rb:144
0.039u 0.002s 0:00.05 60.0%0+0k 0+0io 0pf+0w
```

**Alice**: xxx

## 4.3 xxx

```
require "profile"
```

at the top of a file gives profiling information when you run the file.

```
|gravity> time ruby nb3p.rb -t1 -d0.01 -s0.1 < plum.in
plum.in: .
0.000u 0.000s 0:00.00 0.0%0+0k 0+0io 0pf+0w
```

Hey, this and that.

Now all three:

```
|gravity> (ruby nb1p.rb -t1 -d0.01 -s0.1 < plum.in) | & head -25 | & tail -10
plum.in: .
```

and

```
(ruby nb2p.rb -t1 -d0.01 -s0.1 < plum.in) | & head -25 | & tail -10
```

and

```
(ruby nb3p.rb -t1 -d0.01 -s0.1 < plum.in) | & head -25 | & tail -10
```

so far.

# Chapter 5

# links

A link to Nitadori's home page: *Phantom-GRAPE: speedup on normal chips.*[1]
¡/a¿

And a link to the GRAPE home page: *GRAPE project: hardware speedup.*[2]

More to be added here.

nil nil nil nil nil

[1] `http://grape.astron.s.u-tokyo.ac.jp/∼nitadori/phantom/`
[2] `http://astrogrape.org/`

# Chapter 6

# Literature References

[to be provided]