Direct gravitational N-body simulations in R

Danail Obreschkow

2021-09-20

1 Overview

The **nbody** package contains routines to run and visualise direct gravitational N-body simulations. Its key features are:

- high-accuracy integration with customisable integrators (Euler, leapfrog, 4th and 6th order Yoshida)
- adaptive block timestep
- optional smoothing
- optional background potential and background particles
- optional periodic boundary conditions
- optimised for small numbers of particles (one to a few hundred)
- C++ backend for acceleration evaluations
- built-in interface to external simulators (e.g. GADGET-4, nbodyx)

2 Prerequisites

The **nbody** package is available on GitHub and must be installed using:

```
install.packages('devtools')
library(devtools)
install_github('obreschkow/nbody')
```

Load libraries:

```
library(nbody)
library(magicaxis)
library(png)
library(plotrix)
```

The **nbody** package comes with a built-in gravitational N-body simulator, but also allows the use of external simulators (e.g. GADGET-4 and `nbodyx`). In particular, the Fortran-based `nbodyx` code is a fast simulator, which replicates all the features of the default R-simulator, but is several orders of magnitude faster. `nbodyx` can also be used as a stand-alone tool. I recommend installing `nbodyx` from https://github.com/obreschkow/nbodyx (see README file for detailed instructions).

3 Getting started

The core of the **nbody** package is the routine **run.simulation()**. This routine takes a `simulation` class object as an input and returns a `simulation` class object as an output, with the additional sublist `output` that contains the simulation snapshots. A detailed description of the handling of this routine can be found in the package documentation (call `?run.simulation`).

3.1 First example

To get started, let us compute the motion of two gravitationally bound masses on an eccentric orbit. We can set up the initial conditions (ICs, i.e. masses, positions, velocities) and basic simulation parameters (e.g., gravitational constant, simulation time) by calling:

```
sim = setup.ellipse(e = 0.9, nperiods = 1)
```

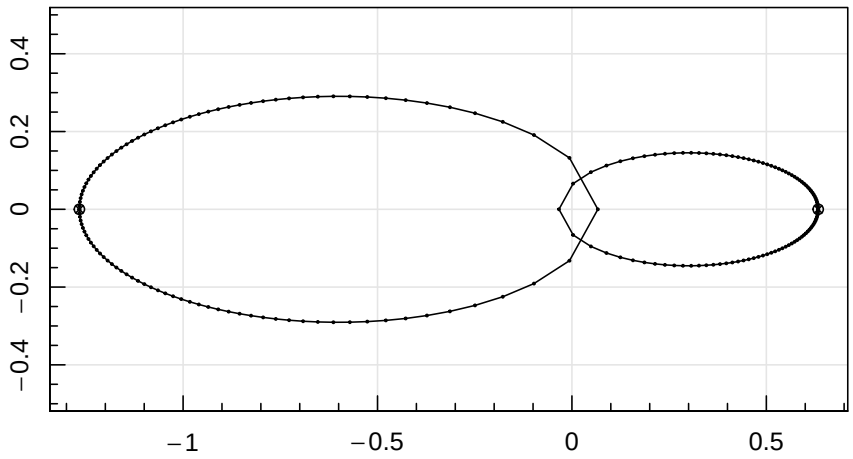
The variable `sim` is a structured list. For example, the vector of the masses is `sim$ics$m` and the gravitational constant is `sim$para$G`. To run the simulation, call:

```
sim = run.simulation(sim)
```

```
## Simulation successfully completed in 0.09s.
```

Now, the list `sim` contains an additional sublist `sim$output`, containing the snapshots (i.e. positions and velocities at discrete times) and a few simulation status variables (e.g. the total number of acceleration evaluations). To get a basic visualisation of the orbits, the standard **plot** function can be called:

```
plot(sim)
```



In this plot, points corresponds to discrete snapshots and straight line segments are used to connect adjacent snapshots. Note that the time interval of the output snapshots is different from the internal timestep, which can be much smaller and is dynamically adjusted as the simulation progresses.

3.2 Using external simulators

If you have installed the Fortran-based `nbodyx` simulator, you can now set this simulator as the default. To do so, the variable `fn` in the following code has to be set to your specific location of `nbodyx`:

```
fn = '~/Dropbox/Code/Fortran/nbodyx/nbodyx/nbodyx'
if (file.exists(fn)) default.code(list(name='nbodyx', file=fn))
```

If we now call **run.simulation** again, the simulation will be performed using the `nbodyx` code:

```
sim = run.simulation(sim)
```

```
## Simulation successfully completed in 0.04s.
```

The resulting orbits are exactly identical, but `nbodyx` generally runs much faster. In the case of this very basic two-body problem, the difference in computation time is barely noticeable. However, for many of examples below, the use of `nbodyx` vastly accelerates the computations.

In a similar manner, **nbody** can be linked to other external simulators (see built-in documentation for an updated list). For example, it is possible to interface with the powerful cosmological simulation code GADGET-4. This code is not optimal and normally very slow for direct N-body simulations, the GADGET-4 interface is sometimes useful for testing purposes.

3.3 Custom initial conditions

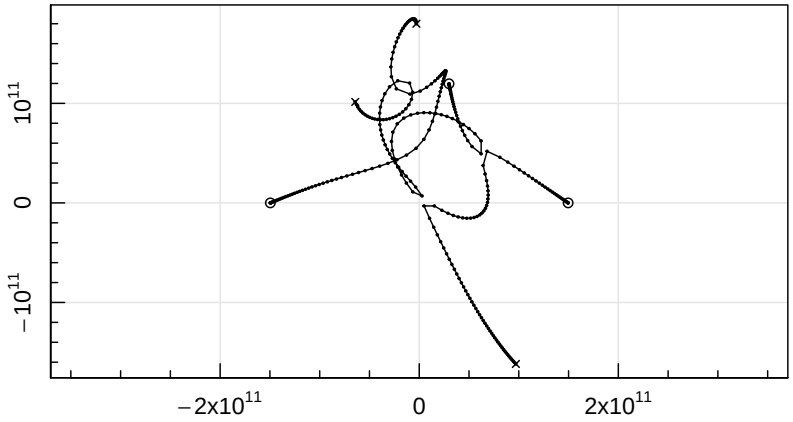
In the previous example, the ICs were generated by the routine **setup.ellipse**. Other routines to set up ICs for selected N-body problems can be found by calling `?setup`.

To define your own ICs, it suffices to create a list `ics` with three components: an N-element vector `m` containing the masses, an N-by-3 matrix `x` with the initial positions and an N-by-3 matrix `v` with the initial velocities. The list `ics` must then be made a sublist of the of a simulation object, here called `sim`. As an example, let us initialise three solar mass particles, initially at rest in the (x, y)-plane:

```
ics = list(m = rep(cst$Msun,3), # masses
           x = rbind(c(1,0,0),c(-1,0,0),c(0.2,0.8,0))*cst$AU, # positions
           v = array(0,c(3,3))) # velocities
sim = list(ics=ics)
```

To run the simulation and plot the result, call:

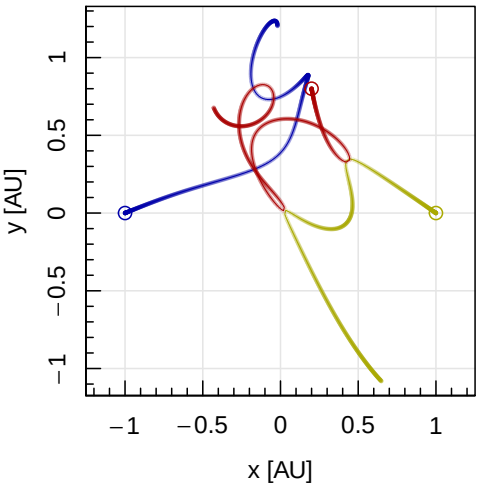
```
sim = run.simulation(sim, measure.time=FALSE)
plot(sim)
```



3.4 Custom visualisation

By specifying a smaller output time step (`dt.out`) in the simulation parameters and using the plotting options, the previous visualisation can be customised and prettified to your liking:

```
sim$para = list(dt.out=cst$hour*6)
sim = run.simulation(sim, measure.time=FALSE)
col = c('#aaaa00','#0000aa','#aa0000')
plot(sim, units=cst$AU, pty='s', xlab='x [AU]', ylab='y [AU]',
     col=col, alpha.snapshots = 0.2, show.fcs=FALSE, lwd=0.5, cex=0.4)
```



To view the full list of optional simulation parameters and plotting options, please refer to the built-in documentation by calling `?run.simulation` and `?plot.simulation`, respectively.

3.5 Note on scale-invariance and using natural units

Gravity is a scale-free phenomenon. More precisely, the solution of Newton's or Einstein's (with $\Lambda = 0$) law of gravity is invariant, if all masses, length scales and times are varied in such a way that the gravitational constant remains preserved.

Explicitly, if in the equation

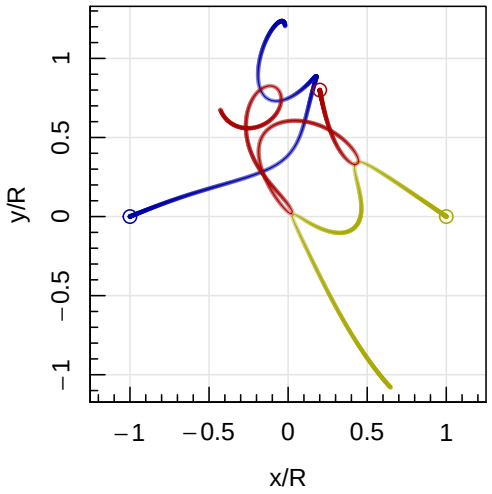
$$\frac{d^2x}{dt^2} = \frac{Gm}{x^2},$$

the three variables (x, m, t) are rescaled to $(\tilde{x}, \tilde{m}, \tilde{t})$, such that $x = f_x\tilde{x}$, $m = f_m\tilde{m}$ and $t = f_t\tilde{t}$, then the same differential equation (with the same constant G) remains vaild iff $f_t^2 f_m = f_x^3$. If this equality is violated, the gravitational constant has to be modified to $\tilde{G} = Gf_x^3/(f_t^2 f_m)$ for Newton's equation to remain valid.

This scale-invariance means that we can always choose natural units for an N-body system, in which $G = M = R = 1$, where M is a characteristic mass (e.g. the total mass $M = \sum m_i = 1$) and $R = 1$ is a characteristic length scale. Any N-body system can be computed in these natural units and rescaled to physical units at the end. In doing so, velocities must be rescaled by $\sqrt{GM/R}$ and times by $\sqrt{R^3/(GM)}$.

As an example, let's recompute the three stellar orbits in natural units:

```
G = M = R = 1
ics = list(m = rep(1/3,3)*M, v = array(0,c(3,3)),
           x = rbind(c(1,0,0),c(-1,0,0),c(0.2,0.8,0))*R)
sim = list(ics=ics, para=list(t.max=5.4*sqrt(R^3/G/M), dt.out=0.005, G=G))
sim = run.simulation(sim, measure.time=FALSE)
plot(sim, units=R, pty='s', xlab='x/R', ylab='y/R',
     col=col, alpha.snapshots = 0.2, show.fcs=FALSE, lwd=0.5, cex=0.4)
```



These orbits are geometrically identical to the previous ones, upon rescaling by R.

Incidentally, this 3-body problem is a good illustration of the generally chaotic orbits of 3-body systems. A few exceptional cases, of non-chaotic 3-body systems will be discussed in later examples.

4 Example: Pythagoras 3-body problem

The so-called Pythagorean 3-body problem starts with three unit masses placed at the vertices of a right triangle with side lengths 3, 4 and 5. The masses are initially at rest and the gravitational constant is unity. It turns out that this is a very challenging 3-body problem that can be used as a hard test for direct N-body integrators.

The set up the ICs and adequate simulation parameters of the Pythagorean 3-body can be set up by calling **setup.pythagoras()**. However, we here initialise the problem manually to see the problems that can arise, if the accuracy parameters are not adequately specified:

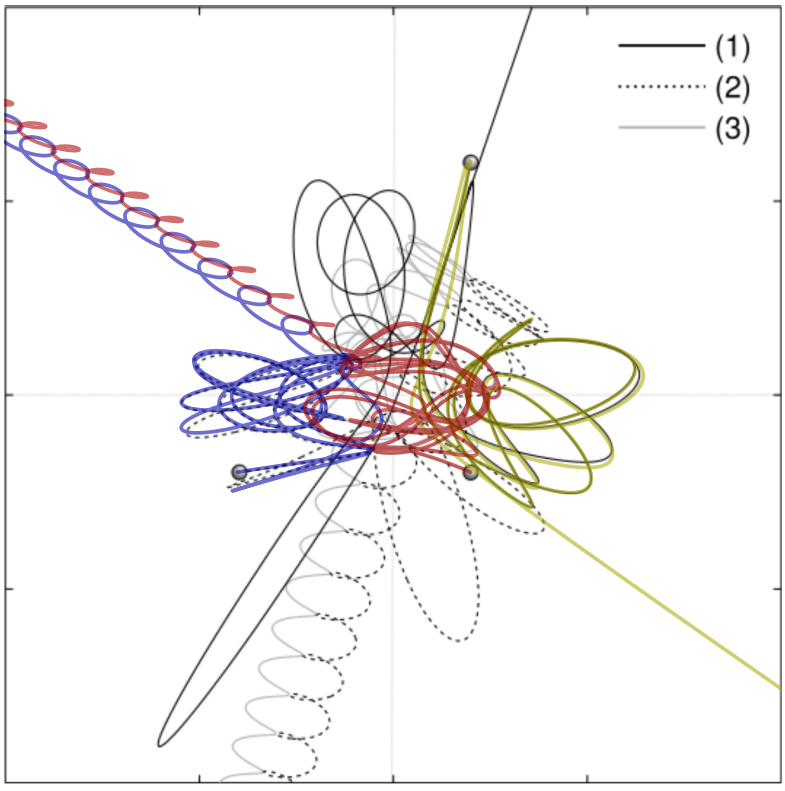
```
ics = list(m = c(3,4,5), # masses
           x = rbind(c(1,3,0),c(-2,-1,0),c(1,-1,0)), # positions
           v = rbind(c(0,0,0),c(0,0,0),c(0,0,0))) # velocities
para = list(G=1, t.max=68, dt.out=0.01) # grav constant, final time and time step
```

To run this simulation, we call:

```
sim = list(ics=ics, para=para)
sim = run.simulation(sim, measure.time=FALSE)
```

Let us plot our numerical solution on top of the correct reference solution (black lines)

```
par(mar=rep(0.3,4),pty='s')
plot(NA,xlim=c(-5,5),ylim=c(-5,5),xaxs='i',xaxt='n',yaxs='i',yaxt='n',xlab='',ylab='')
img = readPNG('../figures/pythagorean.png')
rasterImage(img,-5,-5,5,5)
for (i in seq(3)) lines(sim$output$x[,i,1:2],col=paste0(col[i],'90'),lwd=2)
```



It can be seen, that our orbit looks approximately correct for the first few loops, but then diverges with all three bodies ejected from the view into incorrect directions.

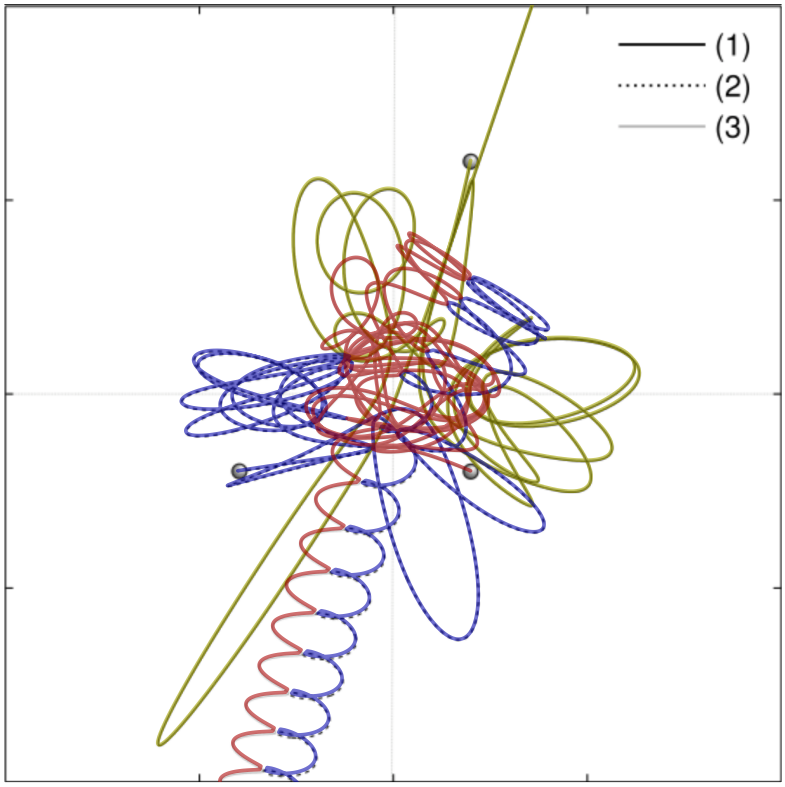
To get a more accurate result we can increase the accuracy by lowering the parameter **eta**, used to determine the adaptive timestep, and/or by choosing a more accurate integrator. The highest accuracy integrator is `yoshida6`. This is a 6th-order scheme, i.e. the position errors decresase to the 6th power of the number of internal timesteps. This extremely accurate. By comparison the standard leapfrog

integrator is only of 2nd order. Like leapfrog, the Yoshida integrators are symplectic. One way to get the correct solution is to choose this integrator in conjunction with a small value of **eta**:

```
sim$para$integrator = 'yoshida6'
sim$para$eta = 0.002
```

In this case the result agrees with the reference solution:

```
sim = run.simulation(sim, measure.time=FALSE)
par(mar=rep(0.3,4),pty='s')
plot(NA,xlim=c(-5,5),ylim=c(-5,5),xaxs='i',xaxt='n',yaxs='i',yaxt='n',xlab='',ylab='')
img = readPNG('../figures/pythagorean.png')
rasterImage(img,-5,-5,5,5)
for (i in seq(3)) lines(sim$output$x[,i,1:2],col=paste0(col[i],'90'),lwd=2)
```



It is possible to obtain similar solutions with lower order integrators, such as the 2nd order leapfrog integrator or the 4th order Yoshida integrator. However, in both these cases a smaller timestep parameter **eta** is needed.

NB: The Pythagorean 3-body is at limit of what can be solved using standard double precision floating point numbers (8 bytes per number). The solution shown in the plot above is therefore not fully converged and it becomes worse if the a smaller value of **eta** is chosen. To bypass this issue, it is possible to compile the `nbodyx` code in 16-byte mode. Please refer to the `nbodyx` README file for details.

5 Example: Gravity assist of Voyager 2

One of the most interesting, not necessarily intuitive processes in 3-body dynamics, are so-called *gravity assists*, also known as gravitational *slingshots* or *swing-bys*. A gravity assist normally refers to the situation of three bodies with vastly different masses, such as the Sun, a planet and a negligible mass, such as a spacecraft. As the name suggest, a gravity assist provides a net acceleration to the spacecraft during a flyby close to the planet. This net gain in kinetic energy results from the fact that the spacecraft experiences a higher acceleration falling towards the planet than deceleration when moving away from it. Gravity assists are typically used to save fuel, cost and time in space flight.

As an example of a well-engineered gravity assist, let's consider the Jupiter flyby manoeuvre of the iconic Voyager 2 space probe.

In the 1960ies, dynamic calculations revealed that it is possible to launch a spacecraft in the late 1970ies, such that it automatically visit all four giant outer planets, using the gravity of each planet to swing itself on to the next. The next such alignment would occur 176 years later. This realisation pushed NASA to build the two Voyager spacecrafts. Voyager 1 was intended to visit Jupiter and Saturn, while Voyager 2 was intended to visit all four gas giants. Voyager 2 was the first of the two probes to be launched; at 14:29:00 UCT on 20 Aug 1977. (NB: It's inspiring to have a look at the Voyager time-line at https://voyager.jpl.nasa.gov/mission/timeline/).

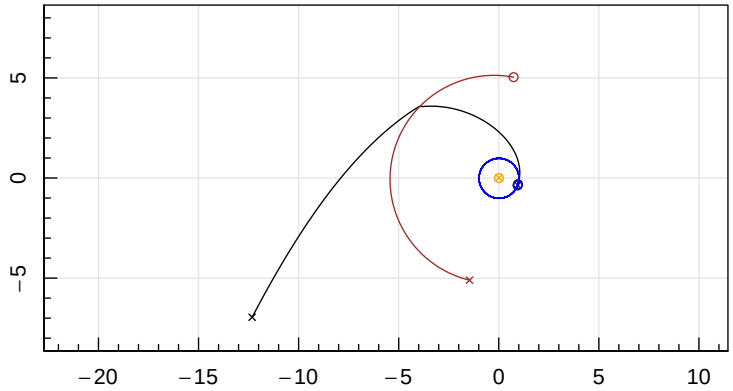
We start the calculation at 00:00:00 on 01 Sep 1977, when the probe was about 25 times farther from Earth than the moon. This starting point allows us to bypass complex calculations involving the launch sequence, the Earth's outer atmosphere and the Earth-Moon system.

The precise solar-system barycentric coordinates and velocityies can be found on the JPL's HORIZONS Web-interface (https://ssd.jpl.nasa.gov/?horizons). In **R**, these ICs read:

```
ics.voyager2 = function() {

  # data from the JPL's HORIZONS Web-interface on 1977-09-01 00:00
  # masses in [kg], positions in [AU], velocities in [AU/day]
  # all values rounded to six significant digits

  # Voyager 2
  m.voyager = 826
  x.voyager = c(+9.58244e-01,-3.09031e-01,+2.12014e-02)
  v.voyager = c(+7.24995e-03,+2.12674e-02,+1.84540e-03)

  # Sun
  m.sun = 1.988500e30
  x.sun = c(+2.33478e-03,-4.39729e-03,-6.31839e-05)
  v.sun = c(+8.05634e-06,-2.48712e-08,-2.19223e-07)

  # Earth-moon-barycentre
  m.earth = 5.97219e24+7.349e22
  x.earth = c(+9.42972e-01,-3.69925e-01,-7.65371e-05)
  v.earth = c(+5.95993e-03,+1.59722e-02,+5.94539e-07)

  # Jupiter
  m.jupiter = 1.89813e27
  x.jupiter = c(+7.40102e-01,+5.03779e+00,-3.73256e-02)
  v.jupiter = c(-7.55408e-03,+1.44595e-03,+1.63273e-04)

  # Combine all bodies
  m = c(m.voyager,m.sun,m.earth,m.jupiter) # [kg]
  x = rbind(x.voyager,x.sun,x.earth,x.jupiter)*cst$AU # [m]
  v = rbind(v.voyager,v.sun,v.earth,v.jupiter)*cst$AU/cst$day # [m/s]
  return(list(m=m, x=x, v=v))
}
```

We run the simulation for 6 years, producing an output every 10 days:

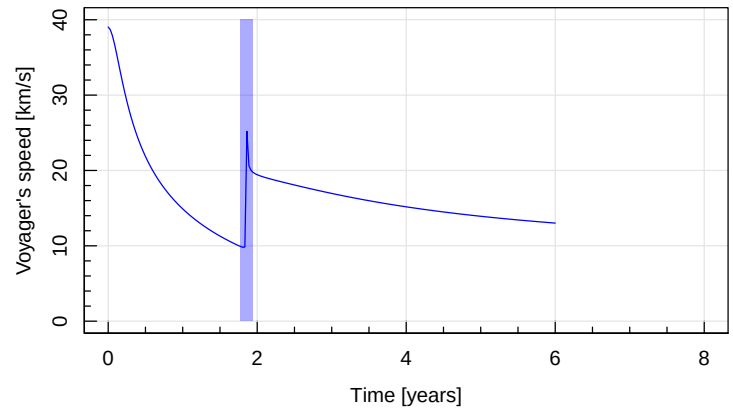
```
sim = list(ics=ics.voyager2(), para=list(t.max=6*cst$yr, dt.out=10*cst$day))
sim = run.simulation(sim, measure.time=FALSE)
plot(sim, units=cst$AU, ylim=c(-8,8),
     col=c('black','orange','blue','brown'), show.snapshots=FALSE)
```



During the close encounter with Jupiter, about 2 years into the mission (on 9 July 1979), the Voyager spacecraft experiences a gravity assist: approaching Jupiter from behind, it steals some of Jupiter's kinetic energy and changes is course significantly.

To appreciate the gain in kinetic energy caused by the gravity assist, let us plot the spacecraft's speed as a function of time:

```
t = sim$output$t/cst$yr # [yr]
v = sqrt(apply(sim$output$v[,1,]^2,1,'sum'))/1e3 # [km/s]
magplot(t,v,type='l',col='blue',xlim=c(0,8),ylim=c(0,40),
        xlab='Time [years]',ylab="Voyager's speed [km/s]")

# overplot region, where the force of Jupiter dominates
r.sun = sqrt(apply((sim$output$x[,1,]-sim$output$x[,2,])^2,1,'sum'))
r.jup = sqrt(apply((sim$output$x[,1,]-sim$output$x[,4,])^2,1,'sum'))
force.ratio = sim$ics$m[4]/r.jup^2/(sim$ics$m[2]/r.sun^2)
t1 = t[min(which(force.ratio>=1))] # time when Jupiter's force starts to dominate
t2 = t[max(which(force.ratio>=1))] # time when Jupiter's force ends to dominate
rect(t1,0,t2,40,col='#0000ff55',border=NA)
```



The duration of the actual encounter can be defined as the timespan where Jupiter's gravitational pull onto the spacecraft exceeds that from the Sun. This duration has been highlighted by the blue shading in the figure. During this encounter, Voyager 2 experienced a net gain in speed of almost 10 km/s!

Voyager 2 experienced three more gravity assists: with Saturn ($\Delta v \approx 8$km/s), with Uranus ($\Delta v \approx 1$km/s) and with Neptune ($\Delta v \approx -1$km/s). However, it is not straightforward to simulate these subsequent encounters, since they depend very sensitively on higher-order effects (due to moons, asteroids, etc.) and small artificial course corrections made using on-board thrusters.

6 Example: Lagrange points

The five Lagrange points (L_1-L_5) are the only points where a massless test particle can co-rotate with two celestial bodies (masses M_1 and $M_2 \leq M_1$) in circular orbit. Three of these points (L_1-L_3) lie on the connecting line between the two bodies. The other two (L_4, L_5) form an equilateral triangle with the two bodies. It is relatively straight-foward to determine the precise positions of L_1-L_5 as a function of the masses of the two bodies. It suffices to find the zero-gradient points (maxima, minima, saddle points) in the effective potential in the co-rotating frame of reference, that is the combination of the two gravitational potentials and the centrifugal potential (see Figure 1).

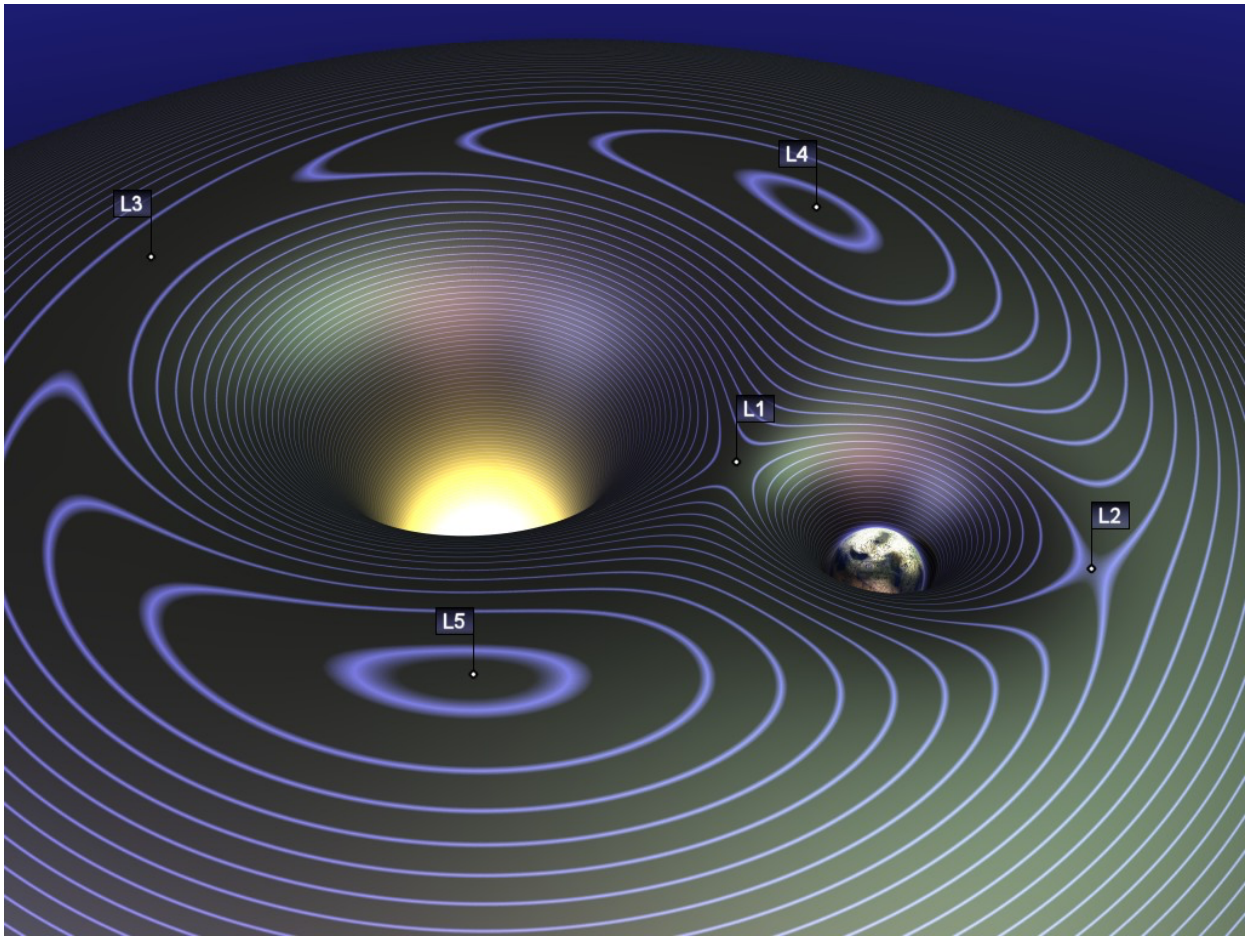


Figure 1: Effective potential in the co-rotating frame of reference of two massive bodies (source unknown).

Determining whether a test particle in a Lagrange point is stable or unstable is a complicated affair. In fact, all five points are saddle points (L_1-L_3) and local maxima ($L_{=}4$, L_5) in the effective potential and hence seemingly unstable. However, as soon as a particle starts moving relative to the co-rotating frame, it will experience a Coriolis acceleration, which cannot be represented by a potential (because it isn't curl-free). The Coriolis term turns out to make L_4 and L_5 stable if and only if

$$\frac{M_1}{M_2} > \frac{25 + \sqrt{621}}{2} \approx 24.95994. \tag{1}$$

In the solar system, this is the case for the Sun and any planet. The co-linear Lagrange points (L_1-L_3) are always unstable. This has practical consequences for the space probes placed near L_1 (e.g. SOHO, Lisa Pathfinder) and L_2 (e.g. Planck, Herschel, JWST, Gaia, Euclid, WFIRST) of the Sun-Earth system. These probes need to carry fuel and fire on-board engines every now and then to hold their position.

6.1 Testing the L_4 stability

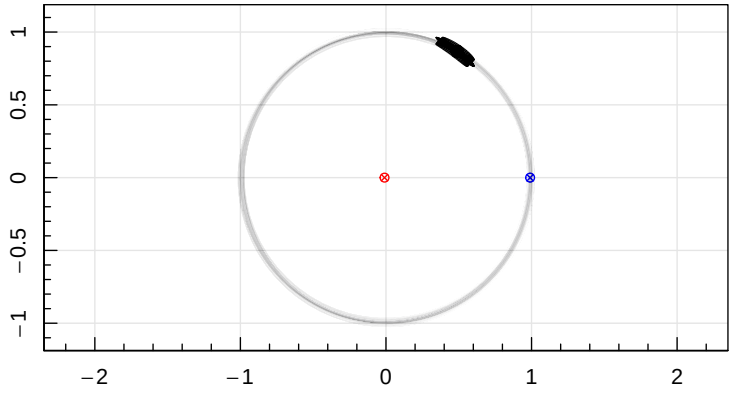
Let us now set up a 3-body simulation to test the stablity of L_4 (and L_5, by symmetry). The aim is to reproduce the result of eq. (1). If successful, we can then trust the simulation as a tool to study the Lagrange point stability in more realistic scenarios, e.g. in the presence of other planets, where no analytical solution exists.

For the simulation we conveniently choose non-dimensional units $G = M = R = 1$, where $M = M_1 + M_2$ is the total mass and R is their separation. In these units the orbital period is exactly 2π, which means that in physical units, the period is $P = 2\pi\sqrt{R^3/(GM)}$. The following routine sets up the ICs for these two masses and a bunch of mass-less test particles placed on a small circle of radius `dx` around L_4.

```
ics.L4 = function(mass.ratio, dx=0.01, n.test.particles=100) {
  # make masses, such that sum(m) = 1
  m = c(1/(1+1/mass.ratio),1/(1+mass.ratio),rep(0,n.test.particles))

  # make positions, such that distance = 1 and CG = (0,0,0)
  x.mass1 = c(-m[2],0,0)
  x.mass2 = c(+m[1],0,0)
  x.test  = c((m[1]-m[2])/2,sqrt(3)/2,0)
  alpha = seq(0,n.test.particles-1)/n.test.particles*2*pi
  x = rbind(x.mass1,x.mass2,
            t(replicate(n.test.particles,x.test))+dx*cbind(cos(alpha),sin(alpha),0))

  # make velocities, such that orbits are circular and v(CG) = (0,0,0)
  v.mass1 = c(0,x.mass1[1],0)
  v.mass2 = c(0,x.mass2[1],0)
  v.test  = c(-x.test[2],x.test[1],0)
  v = rbind(v.mass1,v.mass2,t(replicate(n.test.particles,v.test)))

  return(list(m=m, x=x, v=v))
}
```

Let us first run a simulation of five periods with a mass ratio $M_1/M_2 = 100$:

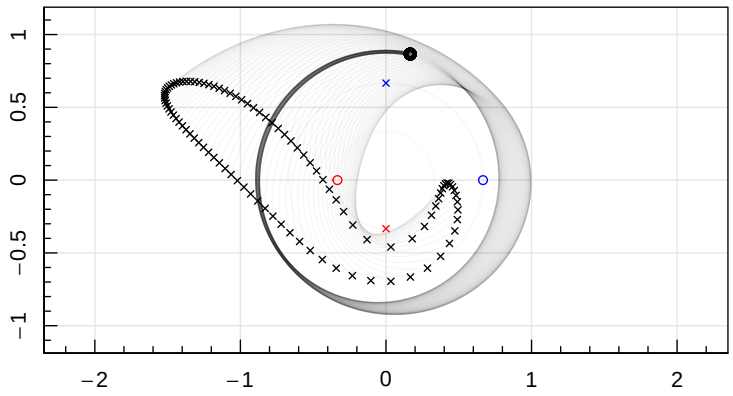
```
sim = list(ics=ics.L4(100), para=list(t.max=5*2*pi, dt.out=0.1, G=1))
sim = run.simulation(sim, measure.time=FALSE)
plot(sim, xlim=c(-1.1,1.1), ylim=c(-1.1,1.1),
     show.snapshots=FALSE, col=c('red','blue',rep('black',100)), alpha.orbits=0.005)
```



As expected, the more massive object (red) stays very close to the origin, while the less massive object (blue) performs five orbits of a radius close to 1. Our 10 test particles near L_4 revolve on a similar orbit and approximately remain on this orbit. In other words, L_4 (and thus L_5) appears to be stable, as expected for this high mass ratio.

Let us now reduce the mass ratio to $M_1/M_2 = 2$ and re-run the simulation for just 1.25 periods.

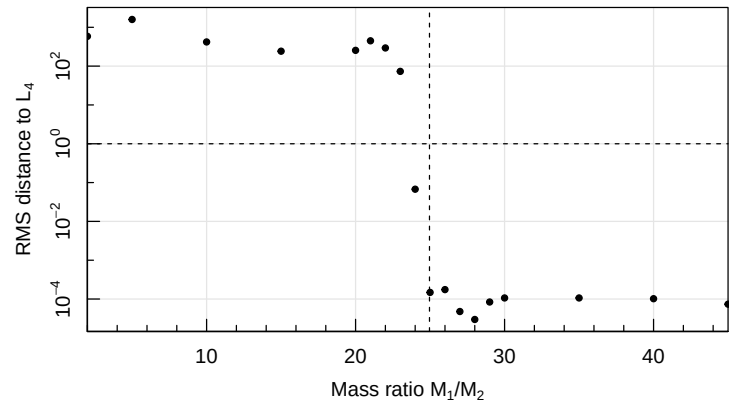
```
sim = list(ics=ics.L4(2), para=list(t.max=2.5*pi, dt.out=0.02, G=1))
sim = run.simulation(sim, measure.time=FALSE)
plot(sim, xlim=c(-1.1,1.1), ylim=c(-1.1,1.1),
     show.snapshots=FALSE, col=c('red','blue',rep('black',100)), alpha.orbits=0.04)
```



While the two massive bodies (red and blue) happily circle around their common centre of mass, the test particles near L_4 leave this Lagrange point rapidly. In other words, L_4 (and L_5) is clearly unstable, as expected from eq. (1) for this low mass ratio.

Next, we run a suite of simulations with mass ratios varying from 2 to 45. Each simulation is run for 50 orbits to make sure that even slight instabilities will manifest themselves. At the same time, we choose to place the particles very close to L_4 (with `dx=0.00001`) to make sure that very small stability regions can still be detected. We only output the final positions and measure the RMS distance of the mass-less test particles to L_4. If this Lagrange point is stable, the final RMS distance is expected to be small ($\ll 1$). If it is unstable, the RMS distance is expected to be large ($\gg 1$).

```
ratio = c(2,5,10,15,20,21,22,23,24,25,26,27,28,29,30,35,40,45)
val = array(NA,length(ratio))
for (i in seq(length(ratio))) {
  sim = list(ics=ics.L4(ratio[i],0.00001),
             para=list(t.max=50*2*pi, dt.out=50*2*pi, eta=0.1, G=1))
  sim = run.simulation(sim, measure.time=FALSE)
  x = sim$output$x[2,3:dim(sim$output$x)[2],] # final test particle positions
  val[i] = sqrt(sd(x[,1])^2+sd(x[,2])^2)
}
magplot(ratio,val,pch=20,xaxs='i',log='y',
        xlab=expression('Mass ratio M' [1]*'/M' [2]),
        ylab=expression('RMS distance to L' [4]))
abline(h=1,v=24.96,lty=2)
```



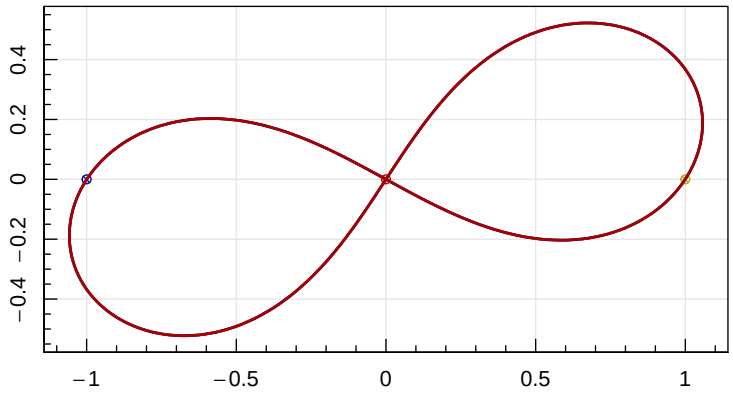
We find a clear transition of the RMS distance from $\gg 1$ to $\ll 1$ around the theoretical mass ratio of eq. (1), shown by the vertical dashed line. One point left of this line (at $M_1/M_2 = 24$) seems semi-stable with an RMS distance of $\sim 0.1 < 1$; however, given more orbits (> 1000), this point will also diverge.

7 Example: Stable periodic 3-body solutions

The gravity assist and Lagrange point problems discussed before are part of the *restricted* 3-body problem, in which one of the bodies is assumed to have neglegible mass. For several centuries it remained unknown whether periodic orbits for three comparable masses exist.

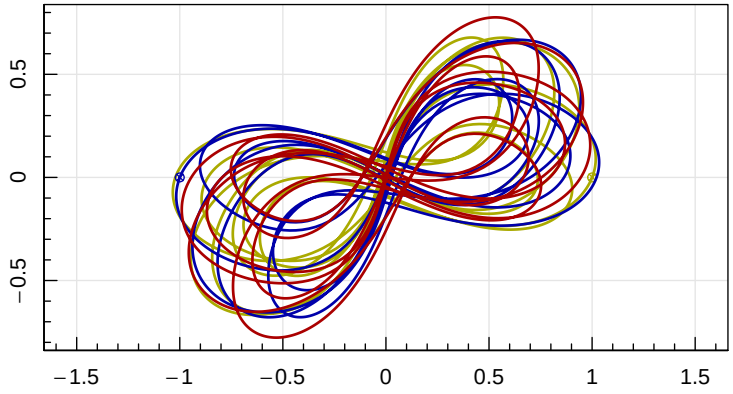
In 1993, Cris Moore discovered the first zero angular momentum solution with three equal masses. These masses are moving around a planar figure-eight shape as illustrated in the following simulation:

```
run.periodic.solution = function(a, b, period, m3=1, ...){

  # initial conditions
  m = c(1,1,m3)
  x = rbind(c(1,0,0),c(-1,0,0),c(0,0,0))
  v = rbind(c(a,b,0),c(a,b,0),c(-2*a/m3,-2*b/m3,0))
  ics = list(m=m,x=x,v=v)

  # simulation
  sim = list(ics=ics, para=list(G=1, t.max=period, ...))
  sim = run.simulation(sim, measure.time=FALSE)
  plot(sim,show.snapshots = F, lwd=2, col=c('#aaaa00','#0000aa','#aa0000'))

}

# orbital parameters
run.periodic.solution(0.3471128135672417, 0.532726851767674, 6.3250)
```



It was later demonstrated that this figure-eight solution is *stable* and several other, stable figure-eight-like solutions were found in its vicinity. Some of these require many several 'eights' for one periods and/or use a slightly different 'eight' for each body. A list of the orbital parameters can be found in Table 2 of Suvakov & Dmitrasinovic (2014), see http://suki.ipb.ac.rs/AJP_Suvakov_Dmitrasinovic.pdf. A nice example is this one:

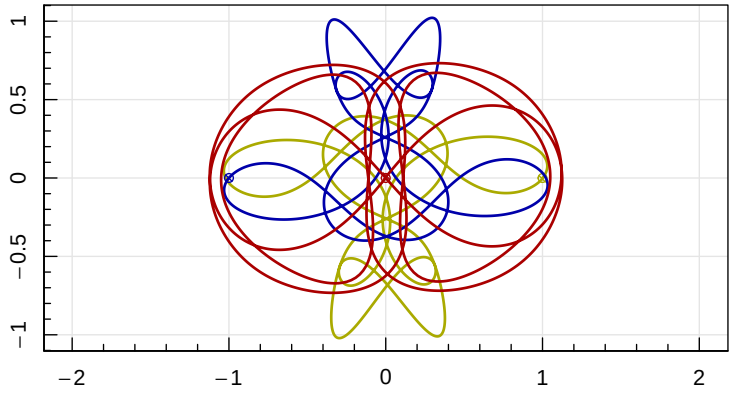
```
run.periodic.solution(0.2034916865234370, 0.5181128588867190, 32.850, dt.out=0.02)
```



Following on, many (>1000) more complicated planar zero angular momentum stable solutions for equal and non-equal masses have been found (e.g. https://arxiv.org/abs/1705.00527, https://arxiv.org/abs/1709.04775).

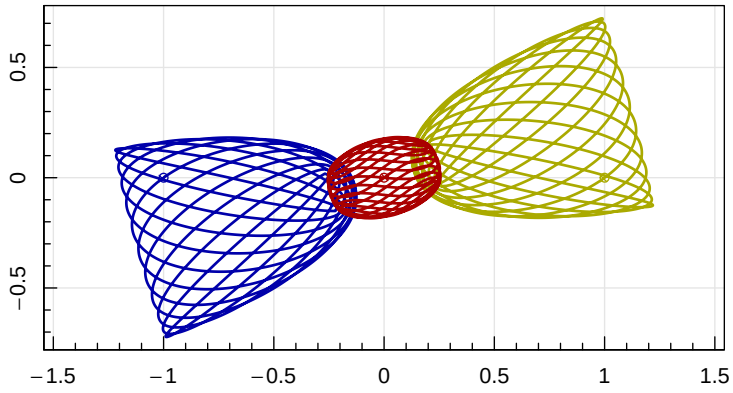
Here is an example with a mass ratio $2:2:1$ (less massive object in red).

```
run.periodic.solution(0.2009656237, 0.2431076328, 19.0134164290, 0.5, dt.out=0.01)
```



And a final example with a mass ratio $1:1:4$ (more massive object in red).

```
run.periodic.solution(0.99119812, 0.71194721, 17.65078078, 4, eta=0.005, dt.out=0.002)
```



8 Example: Collapsing star cluster

Let us now turn to a larger N-body problem, a small star cluster. In this case, two-body relaxation processes (i.e. three-body interactions allowing a pair to become more bound, while the third particle gets ejected) are important. We must avoid smoothing the particles to accurately capture these processes. In practice, we will still use a small smoothing radius to avoid singularities in close encounters, where so-called regularization techniques are needed, which we haven't discussed (e.g. see http://web.pd.astro.it/mapelli/Regularization.pdf). However, our smoothing radius ($\epsilon = 0.01$) is so small that a smoothed sphere will almost always contain just one particle, making the simulation effectively work like a collisional N-body simulation.

As an instructive example we consider the hypothetical formation of a globular star cluster from a uniform distribution of stars inside a sphere with no initial velocities. The continuous analogue of this example is the gravitational collapse of a spherical top-hat distribution, often used in first-order halo formation theories and in testing numerical simulations.

We will choose natural units $G = M = R = 1$, where M is the total mass of the stars and R is the radius of the initial sphere. The following routine sets up N particles (n in **R**), randomly placed inside the unit sphere with a total mass of one and zero velocities.

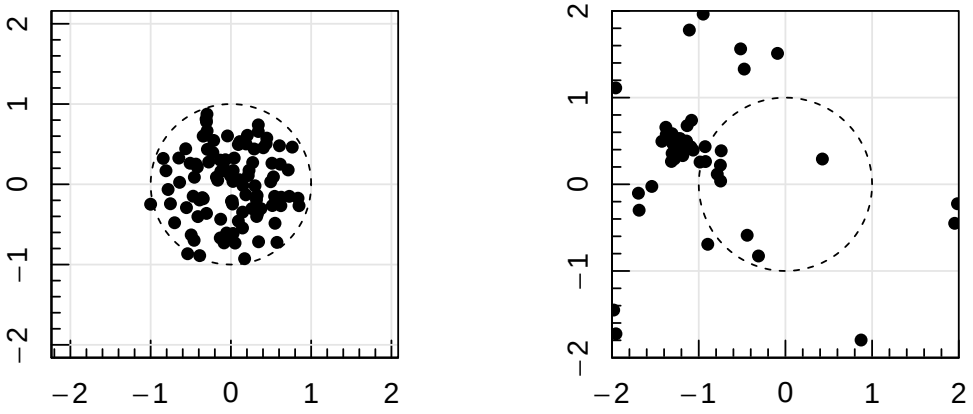
```
ics.cluster = function(n=100, seed=2) {
  set.seed(seed)
  m = rep(1/n,n)
  r = runif(n,0,1)^(1/3) # radius
  phi = runif(n,0,2*pi) # azimuth
  z = r*runif(n,-1,1) # z-coordinate
  x = cbind(sqrt(r^2-z^2)*cos(phi),sqrt(r^2-z^2)*sin(phi),z) # uniform density
  v = array(0,c(n,3)) # zero velocities
  return(list(m=m, x=x, v=v))
}
```

Let us evolve this cluster for 20 characteristic times:

```
sim = list(ics=ics.cluster(), para=list(eta=0.1,G=1,t.max=20,dt.out=0.1,rsmooth=1e-2))
sim = reset.cm(sim)
sim = run.simulation(sim, measure.time=FALSE)
```

This is what the initial and final states look like, with the dashed circle showing the initial sphere:

```
par(mfrow=c(1,2),pty='s')
magplot(sim$output$x[1,,1:2],ylim=c(-2,2),asp=1,pch=16,xlab='',ylab='')
draw.circle(0,0,1,lty=2)
magplot(sim$output$x[sim$output$n.snapshots,,1:2],xlim=c(-2,2),ylim=c(-2,2),
        asp=1,pch=16,xaxs='i',yaxs='i',xlab='',ylab='')
draw.circle(0,0,1,lty=2)
```



8.1 Relaxation and virialization

We would expect that the spherical cluster, initially at rest, will first undergo a global collapse before particle-particle interactions randomise the individual motions. This collapse phase can be approximated by the gravitational collapse of a collisionless gaseous sphere, characterised by the free-fall time

$$t_{\rm ff}(r) = \sqrt{\frac{3\pi}{32G\bar{\rho}(\leq r)}},$$

where $\bar{\rho}(\leq r)$ is the mean density enclosed inside the radius r. For a homogeneous sphere, this density is independent of r and equal to $3M/(4\pi R^3)$, which is just $3/(4\pi)$ in natural units. Hence, all shells share an identical free-fall time of

$$t_{\rm ff} = \frac{\pi}{\sqrt{8}} \approx 1.$$

This free-fall time is thus identical with the overall collapse time, also known as the *shell crossing time* of the system. During this time, we expect the kinetic energy of the system to grow from zero to a maximum point (to infinity for a truly homogeneous sphere). Past the collapse time, local N-body interactions are expected to randomise the motions and establish a virial equilibrium

$$V = -2T$$

between the total potential ($V \leq 0$) and kinetic ($T \geq 0$) energies. The time it takes for the virialisation to occur is known as the relaxation time $t_{\rm relax}$. This time can be thought of as the time it takes for the cluster to 'forget' an earlier phase-space configuration. Following the approximation by Binney & Tremaine,

$$t_{\rm relax} \approx \frac{N}{8 \ln N}\, t_{\rm cross}$$

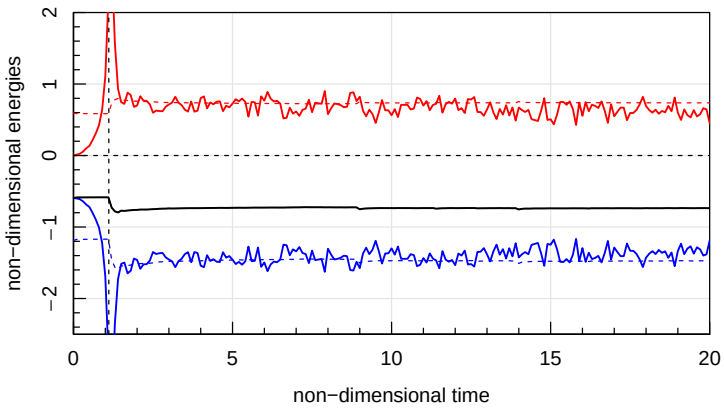
depends on the number of particles N and the average time $t_{\rm cross}$ it takes for a particle to travel a half-mass radius. In our case ($N = 100$ and $t_{\rm cross} \approx 1/3 - 1/2$), we expect $t_{\rm relax} \approx 1$.

In summary, we expect the cluster to collapse in a time $\Delta t = t_{\rm ff} \approx 1$ and subsequently virialise in a similar time span $\Delta t = t_{\rm relax} \approx 1$. To verify this prediction, we now evaluate the energies V (`Epot`) and T (`Ekin`) for each simulation snapshot. In testing the virial theorem, it is important that particles ejected from the system are excluded from the energy calculations because they do not take part in producing a dynamic equilibrium and are expected to carry an excess in kinetic energy. To divide particles into bound and unbound (ejected) ones, we use the practical definition that a particle i is bound if and only if its energy $E_i = T_i + V_i < 0$. Here, T_i is the kinetic energy of the particle with respect to the centre of mass and V_i is the potential energy of that particle with all the other particles (hence $V = \sum_i V_i/2 \neq \sum_i V_i$).

The following routine shows the time evolution of the total (black), kinetic (red) and potential (blue) energies of the bound particles.

```
m = dim(sim$output$x)[1] # number of snapshots (=sim$output$n.snapshot)
n = dim(sim$output$x)[2] # number of particles
nbound = rep(NA,m) # number of bound particles at each snapshot
Ekin = Epot = Emec = rep(NA,m) # energies at each snapshot
for (i in seq(m)) {
  E = energy(sim$ics$m,sim$output$x[i,,],sim$output$v[i,,],G=sim$para$G,
             rsmooth=sim$para$rsmooth)
  s = E$Ekin+E$Epot<0 # select bound particles
  nbound[i] = sum(s) # count bound particles
  Ekin[i] = sum(E$Ekin[s]) # kinetic energy of bound particles
  Epot[i] = sum(E$Epot[s])/2 # potential energy of bound particles
  Emec[i] = sum(E$Ekin[s])+sum(E$Epot[s])/2 # total energy of bound particles
}
magplot(sim$output$t,Emec,type='l',ylim=c(-2.5,2),xaxs='i',yaxs='i',
        xlab='non-dimensional time',ylab='non-dimensional energies')
lines(sim$output$t,Epot,col='blue',lwd=1.5)
lines(sim$output$t,Ekin,col='red',lwd=1.5)
lines(sim$output$t,Emec,col='black',lwd=1.5)
```

```
lines(sim$output$t,-Emec,col='red',lty=2)
lines(sim$output$t,2*Emec,col='blue',lty=2)
abline(h=0,v=pi/sqrt(8),lty=2)
```



The evolution matches the expectation, going through the shell crossing point at the predicted time (vertical dashed line) and subsequently settling onto the virial solution within a relaxation time, such that the equilibrium state is established at $t \approx 2$. The kinetic and potential energies predicted by the virial theorem are shown as dashed lines. The virial theorem works well, once the dynamic equilibrium has been reached.

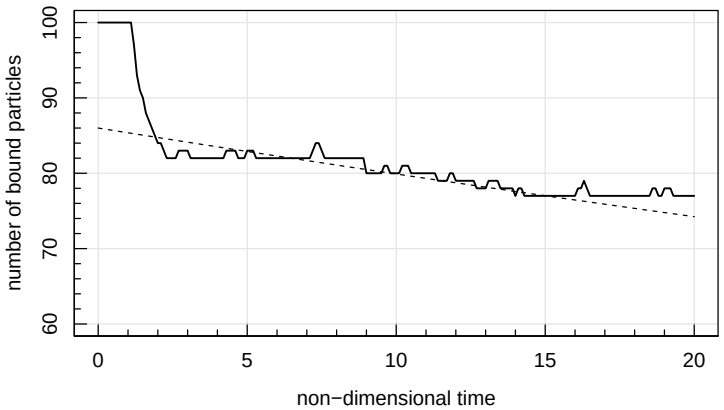
8.2 Evaporation

Due to 3-body (and higher) interactions some particles get ejected over time, a process known as *evaporation*, in analogy to the micro-physics of the evaporation of molecules from a fluid. The evaporation rate is normally modelled as $dN/dt \propto N$, where $N(t)$ denotes the number of bound particles. This solves to $N(t) \propto \exp(-t/t_{\rm evap})$, where $t_{\rm evap}$ is the characteristic evaporation time. Simplistic analytical models suggest that $t_{\rm evap} \approx 100$, more precisely $t_{\rm evap} = 136\, t_{\rm relax} \approx 136$ for a Maxwellian process.

The plot below shows the number of bound particles in our simulation. The exponential model with $t_{\rm evap} = 136$ is shown as dashed line for comparison. This model has been normalised, such that it traces the simulated evolution *after* virialisation has occurred ($t \gtrsim 2$).

The simulation agrees remarkably well with the model. This agreement is somewhat coincidentally, since the evaporation depends strongly on the ICs for such small clusters. In fact, we set the random seed to 2 instead of 1, because the latter produces an abnormally structured cluster with rather slow evaporation.

```
n.eff = 86
t.evap = 136
t.max = max(sim$output$t)
magplot(sim$output$t,nbound,xlim=c(0,t.max),ylim=c(60,100),type='l',lwd=1.5,
        xlab='non-dimensional time',ylab='number of bound particles')
curve(n.eff*exp(-x/t.evap),add=T,lty=2)
```



8.3 Mass segregation

This final example illustrates what happens if the particles have different masses. During a close encounter of two particles, there is a statistical tendency for the kinetic energies to equalise, a principle known as *equipartition*. As a result, more massive particles will migrate closer to the cluster centre, while less massive ones will travel outwards. This process, known as *dynamical mass segregation*, is efficient meaning that it occurs within less than a relaxation time $t_{\rm relax}$, which is about unity in our natural units. For a system made of just two species with masses ($m_2 > m_1$), the toy model by Lyman Spitzer predicts that

$$t_{\rm seg} = (m_1/m_2)\, t_{\rm relax}.$$

According to this model, the ratio $f = r_2/r_1$ between the half mass radii of the particles m_2 and m_1 is predicted to evolve approximately as

$$f(t) = f_{\rm eq} + (f_0 - f_{\rm eq}) e^{-(t-t_0)/t_{\rm seq}},$$

where f_0 is the initial value of f at time t_0 and $f_{\rm eq}$ is the long term equilibrium value of the segregated state. As an example, let us split our N (=100) particles into two species with a mass ratio $m_1/m_2 = 1/2$. The following line achieves this while keeping the total mass equal to unit:

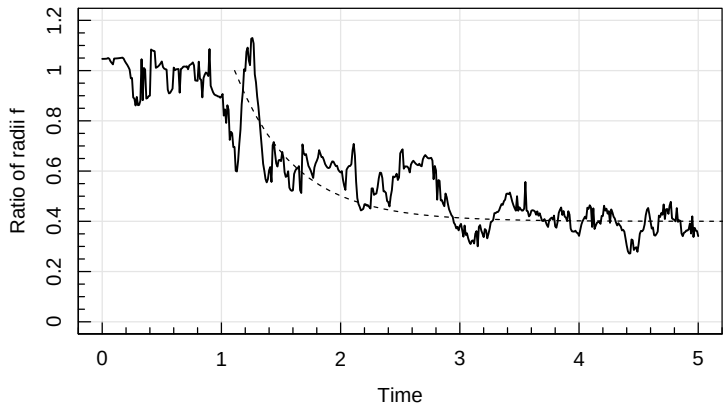
```
sim$ics$m = rep(c(2,1)*2/3/n,n/2)
```

WE simulate this cluster only to $t_{\rm max} = 5$, since there is no point continuing beyond:

```
sim$para$dt.out = 0.01
sim$para$t.max = 5
sim = run.simulation(sim, measure.time=FALSE)
```

Now let US analyse the segregation tracer $f(t)$ in this simulation:

```
f = array(NA,sim$output$n.snapshots)
for (i in seq(sim$output$n.snapshots)) {
  E = energy(sim$ics$m,sim$output$x[i,,],sim$output$v[i,,],G=sim$para$G,
             rsmooth=sim$para$rsmooth)
  k = which.min(E$Epot+E$Ekin) # index of the most bound particle
  x = t(t(sim$output$x[i,,])-sim$output$x[i,k,]) # centre onto the most bound particle
  f[i]= sqrt(median(rowSums(x[sim$ics$m>1/n,]^2))/median(rowSums(x[sim$ics$m<1/n,]^2)))
}
f0 = 1 # expected initial radius ratio
feq = 0.4 # final radius ratio (fitted to simulation by eye)
t0 = pi/sqrt(8) # time when segregation is expected to start (=shell crossing time)
tseg = 1/2 # expected segregation time (=m1/m2)
magplot(sim$output$t,f,type='l',lwd=1.5,
        xlim=c(0,sim$para$t.max),ylim=c(0,1.2),xlab='Time',ylab='Ratio of radii f')
curve(feq+(f0-feq)*exp(-(x-t0)/tseg),t0,10,lty=2,add=T)
```



This analysis shows that segregation occurs immediately after the collapse within a segregation time that roughly agrees with the Spitzer model (dashed line). As in the case of the evaporation, the good agreement between simulation and model is somewhat coincidental and depends on the ICs.