the N-body problem

Author: Martín Alejandre Chaya

Tutor: Renato

How to implement some methods to solve the N-body problem numerically, and how to analyze the results (error and longtime stability). We begin by defining some structures and usefull functions that will later help us implement the methods.

We will only use this essential two Julia packages

In [37]:

```
using LinearAlgebra
using Plots

gr()
```

Out[37]:

```
Plots.GRBackend()
```

Our bodies struct will hold all the necesary information of the bodies. Note that usually, in Hamiltonian Systems, the variables used are p and q, the momentum and position. But in our case we preffer to separe the momentum $p = mv$ into the masses and velocities of the bodies.

In [38]:

```
mutable struct Bodies
    m :: Array{Float64} # masses
    vel :: Matrix{Float64} # velocities
    q :: Matrix{Float64} # positions
    n :: Int64 # number of bodies
    dims :: Int64 # dimensions
end
```

Our Simulator struct will hold all the relevant information for a specific integrator method.

In [39]:

```
mutable struct Simulator
    bodies :: Bodies
    dt :: Float64
    G :: Float64
    method :: Function
end
```

Now we will go on to define some functions that will be usefull later. Recall some equations:

Hamiltonian of a system,

$$H(q,p) = K + P$$

Kinetic energy of a system,

$$K = \frac{1}{2}mv^2 = \frac{p^2}{2m}$$

Potential energy between two bodies,

$$P = -G\frac{mM}{r}$$

Angular momentum of a system,

$$L = \sum_{i=1}^{N} q_i \times p_i$$

In [40]:

```
# Returns the current hamiltonian of the system
function calculate_current_H(s::Simulator)
    KE = 0.0
    # Computation of Kinetic Energy
    for i in 1:s.bodies.n
        KE += 0.5 * s.bodies.m[i]*(norm(s.bodies.vel[i,:])^2)
    end
    # Computation of Potential Energy
    PE = computePE(s)
    # Kinetic energy + Potential energy
    return KE + PE
end
```

Out[40]:

```
calculate_current_H (generic function with 1 method)
```

In [41]:

```
# Returns the current Potential energy of the system
function computePE(s::Simulator)
    PE = 0.0

    for i in 2:s.bodies.n
        for j in 1:i-1
            r = norm(s.bodies.q[i,:] - s.bodies.q[j,:])
            PE -= s.bodies.m[i]*s.bodies.m[j]/r
        end
    end

    return s.G*PE
end
```

Out[41]:

```
computePE (generic function with 1 method)
```

In [42]:

```
# Calculates the current angular momentum
function calculate_current_L(s::Simulator)
    total_L = zeros(s.bodies.dims) # 3d vector
    for i in 1:s.bodies.n
        total_L += cross(s.bodies.q[i, :], s.bodies.vel[i, :]*s.bodies.m[i])
    end
    return total_L
end
```

Out[42]:

```
calculate_current_L (generic function with 1 method)
```

Now we will define some functions that will help us implement more easily the methods. Namely, *computeForce* will compute all the forces acting on all the bodies given a current state of the system.

Recall the equation for the gravity force between two bodies:

$$F_{ij} = -G\frac{m_i m_j}{r^2}u_{ij}$$

where u_{ij} is the unity vector that joins body i with body j

In [43]:

```
# Returns a matrix whose ith row is the force acting on body i
function computeForce(s::Simulator)

    force = zeros(s.bodies.n, s.bodies.dims) # (n_bodies, dims)

    for i in 2:s.bodies.n
        for j in 1:i-1
            # For every pair of bodies

            dij = s.bodies.q[i, :] - s.bodies.q[j, :] # distance vector
            r = norm(dij)

            fij = -s.G*s.bodies.m[i]*s.bodies.m[j]/(r*r*r)*dij

            force[i,:] += fij
            force[j,:] -= fij
        end
    end

    return force
end
```

Out[43]:

```
computeForce (generic function with 1 method)
```

In [44]:

```
# Runs a step of the simulation
function step!(s::Simulator)
    s.bodies.vel, s.bodies.q = s.method(s)
end

# Runs N steps of the simulation
function simulate(s::Simulator, n::Int64)
    for _ in 1:n
        step!(s)
    end
end
```

Out[44]:

```
simulate (generic function with 1 method)
```

Now we have set the framework to implement our methods. As you can see above we will implement the methods in such a way that, given the current state of the system, they return the veolicity and position of all the bodies in the next state.

Recall the different methods schemes, where h is the time-step and the equation we are trying to solve is

$$\frac{dq}{dt} = v$$

$$\frac{dv}{dt} = f(q)$$

Euler

$$q^{n+1} = q^n + hv^n$$

$$v^{n+1} = v^n + hf(q^n)$$

Euler-B

$$q^{n+1} = q^n + hv^{n+1}$$

$$v^{n+1} = v^n + hf(q^n)$$

Störmer-Verlet

$$v^{n+\frac{1}{2}} = v^n + \frac{h}{2}f(q^n)$$

$$q^{n+1} = q^n + hv^{n+\frac{1}{2}}$$

$$v^{n+1} = v^{n+\frac{1}{2}} + \frac{h}{2}f(q^{n+1})$$

Note that in the implementation, since we are using a function to compute the forces, $f(q^n)_i = F_i/m_i$ where F_i is the force computed for body i

In [45]:

```
# euler method
function euler(s::Simulator)

    force = computeForce(s)

    nq = s.bodies.q + s.dt*s.bodies.vel
    nvel = s.bodies.vel + force./s.bodies.m*s.dt

    return nvel, nq
end
```

Out[45]:

```
euler (generic function with 1 method)
```

In [46]:

```
# euler-b method
function euler_b(s::Simulator)

    force = computeForce(s)

    nvel = s.bodies.vel + force./s.bodies.m*s.dt
    nq = s.bodies.q + s.dt*nvel

    return nvel, nq
end
```

Out[46]:

```
euler_b (generic function with 1 method)
```

In [47]:

```
# Stormer-Verlet method
function stormer_verlet(s::Simulator)

    force = computeForce(s)

    vel_half = s.bodies.vel + force./s.bodies.m * (s.dt/2)
    nq = s.bodies.q + s.dt*vel_half

    s.bodies.q = nq
    force = computeForce(s)
    nvel = vel_half + force./s.bodies.m * (s.dt/2)

    return nvel, nq
end
```

Out[47]:

```
stormer_verlet (generic function with 1 method)
```

Now we have all the code we need to run the simulations, but we are going to first define a last function. Said function calculates the velocity of a Moon orbiting a certain planet. For this calculations we don't take into account any other planets since the magnitude of the forces they exert will be negligible if we plug in normal numbers.

Recall that the equation for the magnitud of the orbital velocity is

$$\|v\| = \sqrt{G\frac{M}{r}}$$

where M is the mass of the planet, and r the distance between the planet and the moon. For our computations we also have to take into account that this speed is relative to the planet, meaning in our system, the moon will have the speed of the planet plus the orbital speed. Bare in mind also that the orbital speed of the moon must be perpendicular to the velocity of the planet, and the vector that joins the planet and the moon.

In [48]:

```
# Calculates (in 3d space) the velocity of the moon
# of a planet at a certain distance from said planet
function make_moon3d(planet_mass, G, planet_pos, moon_pos, planet_vel)
    v = copy(planet_vel) # Copy

    # Compute orthogonal component (orbital speed)
    dist = moon_pos-planet_pos
    ov = normalize(cross(dist, v))
    ov *= sqrt(G*planet_mass/norm(dist))

    v += ov # Add to speed of planet
    return v
end
```

Out[48]:

```
make_moon3d (generic function with 1 method)
```

Simulations

Now we can finally run some simulations. First we will define some values to avoid having to redefine them every time. We normalized the system, so the sun starts in the origin of coordinates and has mass 1.

In [49]:

```
G = 0.00029591
```

Out[49]:

```
0.00029591
```

In [50]:

```
function get_solar_system(N_bodies, dims)
    if (N_bodies > 6)
        print("N_bodies must be <= 6")
        return
    end
    m = zeros(6)
    init_q = zeros(6, dims)
    init_v = zeros(6, dims)

    m[1] = 1. # sun
    m[2] = 0.000954786 # jupyter
    m[3] = 0.000285583 # saturn
    m[4] = 0.0000437273164546 # uranus
    m[5] = 0.0000517759138449 # neptune
    m[6] = 1/(1.3e8) # pluto

    init_v = [0.0 0.0 0.0;
              0.00565429 -0.00412490 -0.00190589;
              0.00168318 0.00483525 0.00192462;
              0.00354178 0.00137102 0.00055029;
              0.00288930 0.00114527 0.00039677;
              0.00276725 -0.00170702 -0.00136504]

    init_q = [0.0 0.0 0.0;
            -3.5023653 -3.8169847 -1.5507963;
            9.0755314 -3.0458353 -1.6483708;
            8.3101420 -16.2901086 -7.2521278;
            11.4707666 -25.7294829 -10.8169456;
            -15.5387357 -25.2225594 -3.1902382]

    bodies = Bodies(m[1:N_bodies], init_v[1:N_bodies, :], init_q[1:N_bodies, :], N_bodies, dims)
    return bodies
end
```

Out[50]:

```
get_solar_system (generic function with 1 method)
```

In [51]:

```
function get_solar_system_moons(N_bodies, N_moons)
    if (N_bodies > 6 || N_moons > 4)
        print("N_bodies must be <= 6 and N_moons must be <= 4")
    end
    dims = 3 # 2 dimensions
    m = zeros(10)
    init_q = zeros(10, dims)
    init_v = zeros(10, dims)

    # The masses of the moons must be small compared to the planets they orbit
    m[1] = 1. # sun
    m[2] = 0.000954786 # jupyter
    m[3] = 0.000285583 # saturn
    m[4] = 0.0000437273164546 # uranus
    m[5] = 0.0000517759138449 # neptune
    m[6] = 1/(1.3e8) # pluto
    m[7] = 0.000005 # jupyter moon
    m[8] = 0.00000005 # jupyter moons moon
    m[9] = 0.00002 # saturn moon
    m[10] = 0.00002 # saturn moon 2

    # The initial velocity of the moons will be defined later
    init_v = [0.0 0.0 0.0;
              0.00565429 -0.00412490 -0.00190589;
              0.00168318 0.00483525 0.00192462;
              0.00354178 0.00137102 0.00055029;
              0.00288930 0.00114527 0.00039677;
              0.00276725 -0.00170702 -0.00136504;
              0. 0. 0.;
              0. 0. 0.;
              0. 0. 0.;
              0. 0. 0.;]

    # The initial position of the moons is suposed to be very close to
    # the initial position of the planet it orbits
    init_q = [0.0 0.0 0.0;
              -3.5023653 -3.8169847 -1.5507963;
              9.0755314 -3.0458353 -1.6483708;
              8.3101420 -16.2901086 -7.2521278;
              11.4707666 -25.7294829 -10.8169456;
              -15.5387357 -25.2225594 -3.1902382;
              -3.5023653 -3.95 -1.5507963; # -3.5023653 -3.8169847 -1.5507963;
              -3.5023653 -3.95 -1.555;
              9.1355314 -3.0458353 -1.6483708; # 9.0755314 -3.0458353 -1.6483708;
              9.0755314 -3.2458353 -1.6483708;]

    init_v[7, :] = make_moon3d(m[2], G, init_q[2, :], init_q[7, :], init_v[2, :]) # Jupyter's moon
    init_v[8, :] = make_moon3d(m[7], G, init_q[7, :], init_q[8, :], init_v[7, :]) # Jupyer's moon's mon
    init_v[9, :] = make_moon3d(m[3], G, init_q[3, :], init_q[9, :], init_v[3, :]) # Saturn moon 1
    init_v[10, :] = make_moon3d(m[3], G, init_q[3, :], init_q[10, :], init_v[3, :]) # Saturn moon 2

    bodies = Bodies(m[[1:N_bodies; 7:(6+N_moons)]], init_v[[1:N_bodies; 7:(6+N_moons)], :], init_q[[1:N_bodies; 7
:(6+N_moons)], :], N_bodies+N_moons, dims)

end
```

Out[51]:

```
get_solar_system_moons (generic function with 1 method)
```

Now we can run some simulations. We will save the positions of all the bodies to a matrix, to later plot them and visualize the results of our integrators.

In [52]:

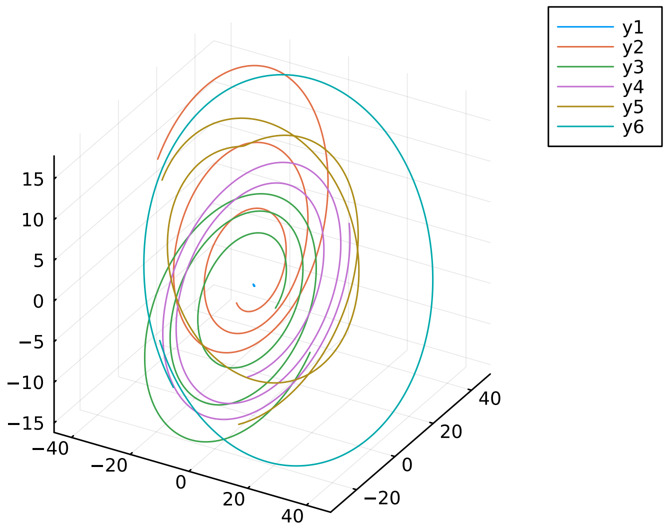
```
function solar_system(method, N_bodies, N, dt, N_moons = 0)
    if (N_moons == 0)
        bodies = get_solar_system(N_bodies, 3)
    else
        bodies = get_solar_system_moons(N_bodies, N_moons)
    end
    sim = Simulator(bodies, dt, G, method)

    positions = zeros(N, N_bodies+N_moons, 3);
    for i in 1:N
        step!(sim)
        positions[i, :, :] = sim.bodies.q
    end

    plt = plot()
    for i in 1:(N_bodies+N_moons)
        plot!(plt, positions[:, i, 1], positions[:, i, 2], positions[:, i, 3])
    end
    display(plt)
end
```

Out[52]:

```
solar_system (generic function with 2 methods)
```

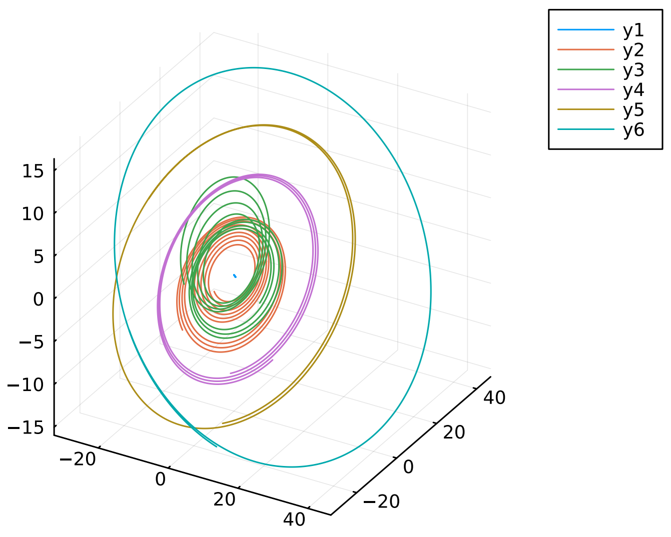
In [53]:

```
solar_system(euler, 6, 1000, 100)
```



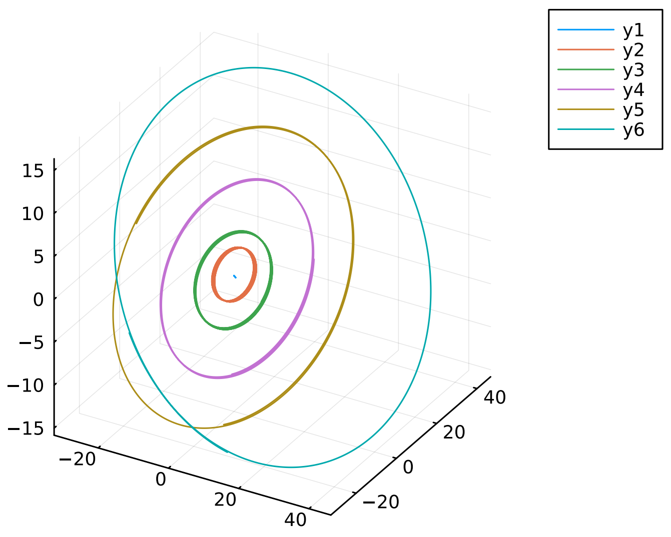
In [54]:

```
solar_system(euler, 6, 10000, 10) # Even with reduced step size it is not stable
```



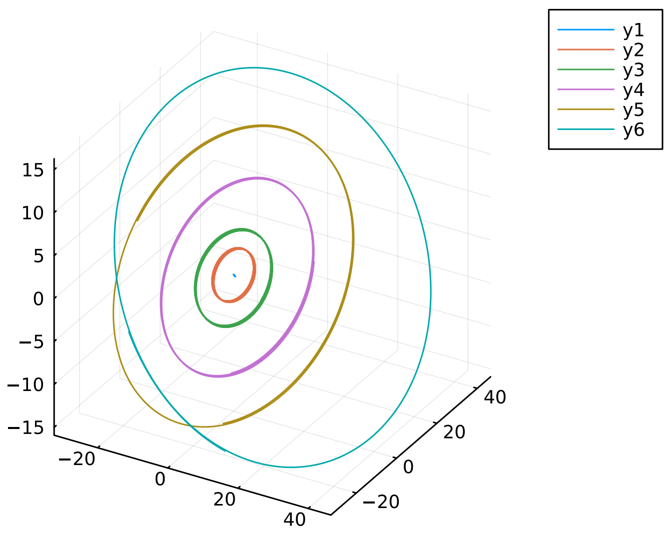
In [55]:

```
solar_system(euler_b, 6, 1000, 100)
```



In [56]:

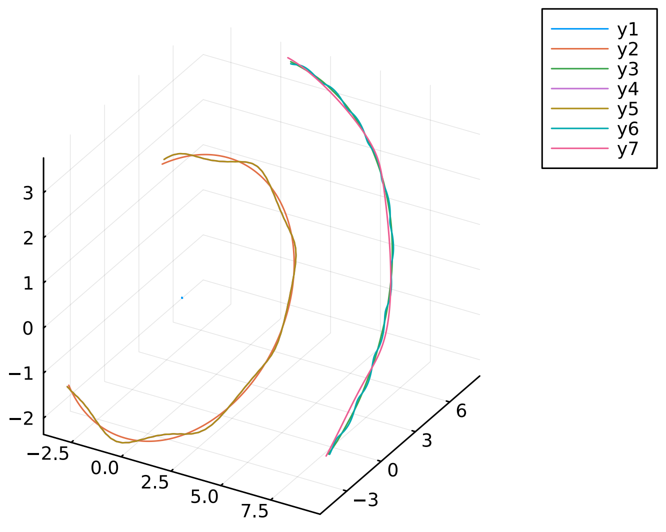
```
solar_system(stormer_verlet, 6, 1000, 100)
```



As you can see the Euler method is very unstable. Meanwhile the other two methods behave nicely. Now lets use the moons code.

In [57]:

```
solar_system(euler_b, 3, 3000, 1, 4)
```



You can observe there are two planets. The first one (y2) has a moon (y4), who itself has a moon(y5). And the second one (y3) has two moons (y6 and y-7) orbiting it at different distances and hence with different periods.

Longtime Stability

Now we will write some code to compare the longtime stability of these integrators. All the relevant details are in the memory, here only the code will be shown. We want to keep track of how well the integrators perserve their first integrals. Recall that first integrals are values that don't change through time, for example, the total energy of a closed system. We will use the functions that calculate the Hamiltonian and the total angular momentum of the system, which are first integrals of the N-body problem.

In [58]:

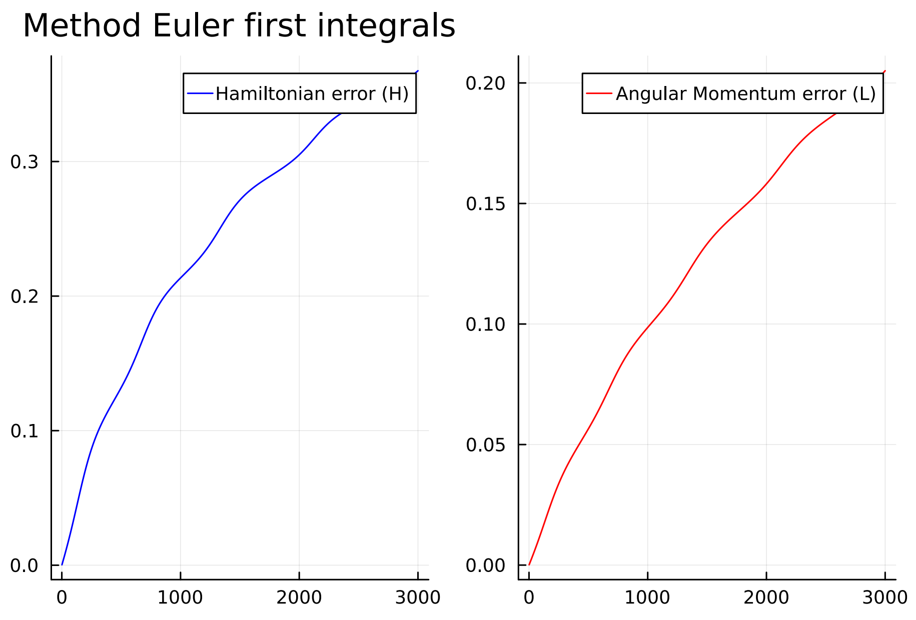
```
function get_H_L(method, dt, N, N_bodies = 6)
    bodies = get_solar_system(N_bodies, 3)
    sim = Simulator(bodies, dt, G, method)

    init_H = calculate_current_H(sim)
    init_L = calculate_current_L(sim)

    Hs = zeros(N)
    Ls = zeros(N)
    for i in 1:N
        step!(sim)
        Hs[i] = abs(calculate_current_H(sim) - init_H) / abs(init_H)
        Ls[i] = norm(calculate_current_L(sim) - init_L) / norm(init_L)
    end

    return Hs, Ls
end
```

Out[58]:

```
get_H_L (generic function with 2 methods)
```

In [59]:

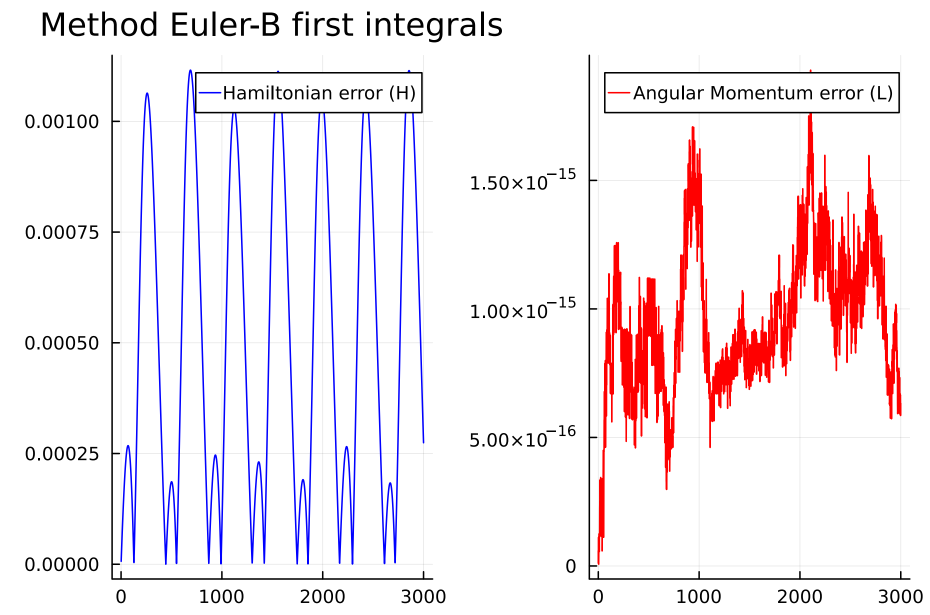
```
function to_str(method::Function)
    if (method == euler)
        return "Euler"
    elseif (method == euler_b)
        return "Euler-B"
    elseif (method == stormer_verlet)
        return "Stormer-Verlet"
    else
        return "None"
    end
end
function plot_H_L(method, dt, N, N_bodies = 6)
    Hs, Ls = get_H_L(method, dt, N)
    plt1 = plot(Hs, color = :blue, label = "Hamiltonian error (H)", title = "Method "*to_str(method)*" first inte
grals")
    plt2 = plot(Ls, color = :red, label = "Angular Momentum error (L)")
    p = plot(plt1, plt2, layout = (1, 2))
    display(p)
end

plot_H_L(euler, 10, 3000)
```

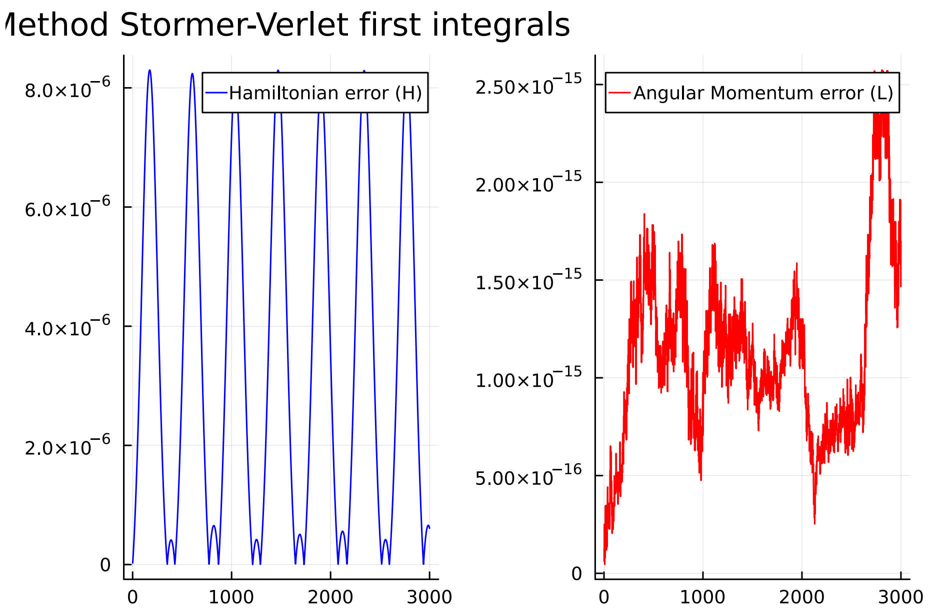


Here you can see for example that for Euler method the error increases linearly. We can try with the other methods

In [60]:

```
plot_H_L(euler_b, 10, 3000)

plot_H_L(stormer_verlet, 10, 3000)
```





As you can see, the Hamiltonian oscilates, so its not too interesting to see it plotted through time. We might be interested in seeing how different time steps affect the conservation of first integrals. So instead of plotting the Hamiltonian through time, we will plot the maximum error for different time steps

In [61]:

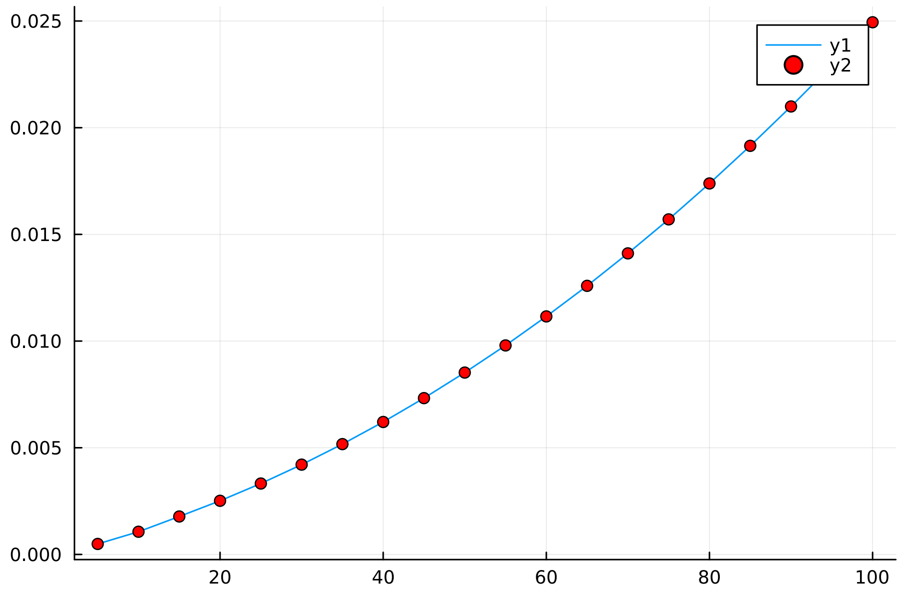
```
function get_maxH_dt(method, dts, T, N_bodies = 6)

    Hs = zeros(length(dts))

    for k in 1:length(dts)
        dt = dts[k]
        N = trunc(T/dt)
        bodies = get_solar_system(N_bodies, 3)
        sim = Simulator(bodies, dt, G, method)
        max_H = -1
        init_H = calculate_current_H(sim)
        for i in 1:min(N, 500)
            step!(sim)
            max_H = max(max_H, abs(calculate_current_H(sim)-init_H) / abs(init_H));
        end
        Hs[k] = max_H
    end

    return Hs
end
```

Out[61]:

```
get_maxH_dt (generic function with 2 methods)
```

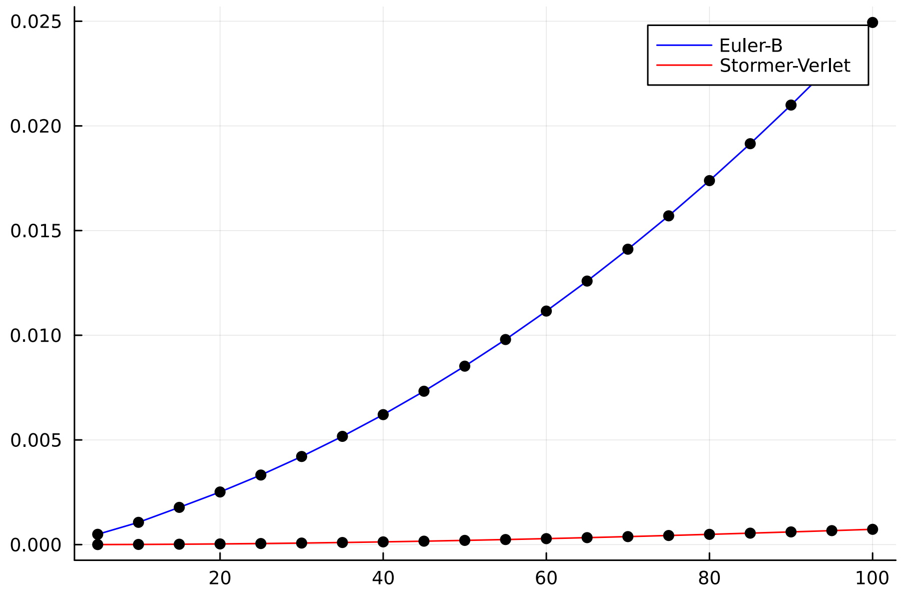
In [62]:

```
dts = [5*i for i in 1:20]
Hs = get_maxH_dt(euler_b, dts, 10000)

plt = plot(dts, Hs)
scatter!(plt, dts, Hs, color=:red)
display(plt)
```



In [63]:

```
dts = [5*i for i in 1:20]
Hs = get_maxH_dt(euler_b, dts, 10000)
Hs2 = get_maxH_dt(stormer_verlet, dts, 10000)

plt = plot(dts, Hs, color=:blue, label="Euler-B")
plot!(plt, dts, Hs2, color=:red, label="Stormer-Verlet")
scatter!(plt, dts, Hs, color=:black, label=false)
scatter!(plt, dts, Hs2, color=:black, label=false)
display(plt)
```



Now we can do a similar thing for the conservation of momentum. First thing we will check how the different integrators behave in longtime simulations

In [64]:

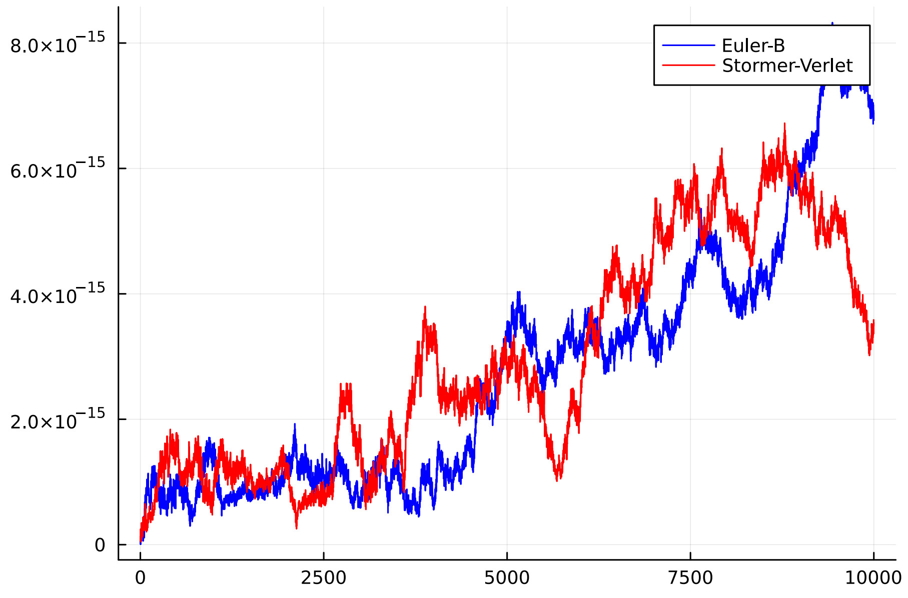
```
function get_L(method, dt, N, N_bodies = 6)
    bodies = get_solar_system(N_bodies, 3)
    sim = Simulator(bodies, dt, G, method)

    init_L = calculate_current_L(sim)
    Ls = zeros(N)

    for i in 1:N
        step!(sim)
        Ls[i] = norm(calculate_current_L(sim) - init_L) / norm(init_L)
    end

    return Ls
end
```

Out[64]:

```
get_L (generic function with 2 methods)
```

In [65]:

```
Ls1 = get_L(euler_b, 10, 10000)
Ls2 = get_L(stormer_verlet, 10, 10000)

plt = plot(Ls1, label="Euler-B", color=:blue)
plot!(plt, Ls2, label="Stormer-Verlet", color=:red)
display(plt)
```



Now rewritting the function we defined above:

In [66]:

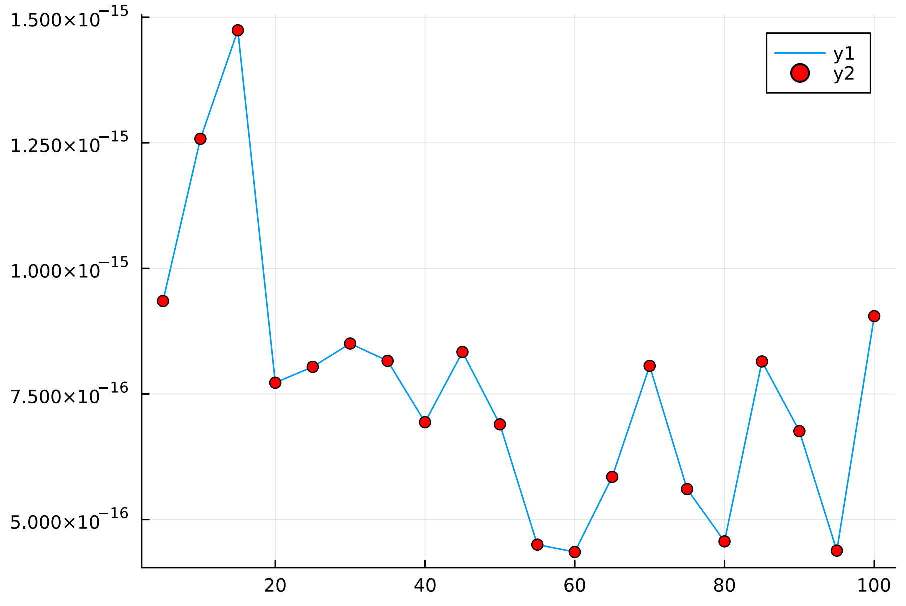
```
function get_max_error_dt(method, dts, T, N_bodies = 6)

    Hs = zeros(length(dts))
    Ls = zeros(length(dts))

    for k in 1:length(dts)
        dt = dts[k]
        N = trunc(T/dt)
        bodies = get_solar_system(N_bodies, 3)
        sim = Simulator(bodies, dt, G, method)
        max_H = -1
        max_L = -1
        init_H = calculate_current_H(sim)
        init_L = calculate_current_L(sim)
        for i in 1:min(N, 500)
            step!(sim)
            max_H = max(max_H, abs(calculate_current_H(sim)-init_H) / abs(init_H))
            max_L = max(max_L, norm(calculate_current_L(sim)-init_L) / norm(init_L))
        end
        Hs[k] = max_H
        Ls[k] = max_L
    end

    return Hs, Ls
end
```

Out[66]:

```
get_max_error_dt (generic function with 2 methods)
```

In [67]:

```
dts = [5*i for i in 1:20]
_, Ls = get_max_error_dt(euler_b, dts, 10000)

plt = plot(dts, Ls)
scatter!(plt, dts, Ls, color=:red)
display(plt)
```

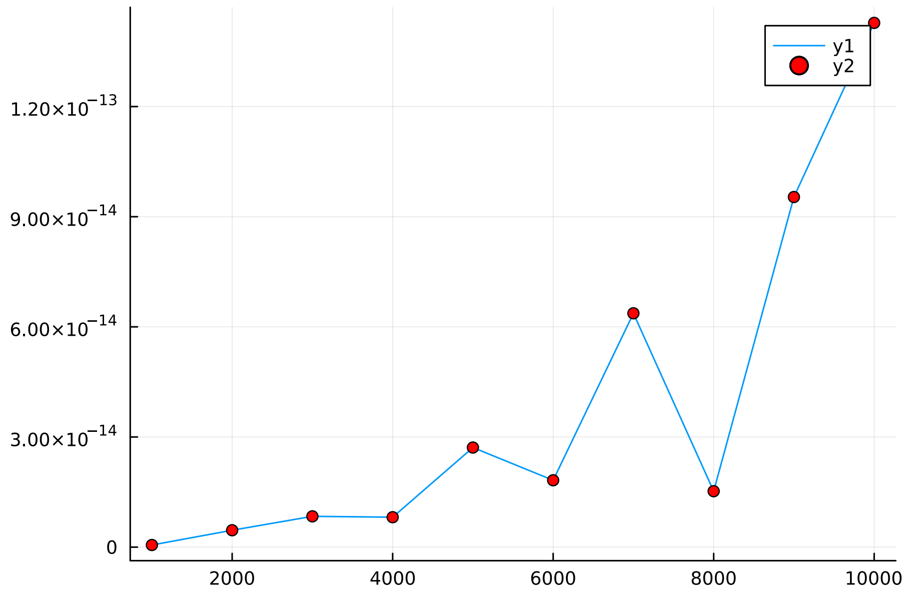


This suggests that the angular momentum is perfectly conserved, and there are just some little fluctuations and errors that appear because of the computer precission error. But again, all the analysis is written down in the document.

How big can we have dt and still conserve the angular momentum error in order 10^{-15}?

In [68]:

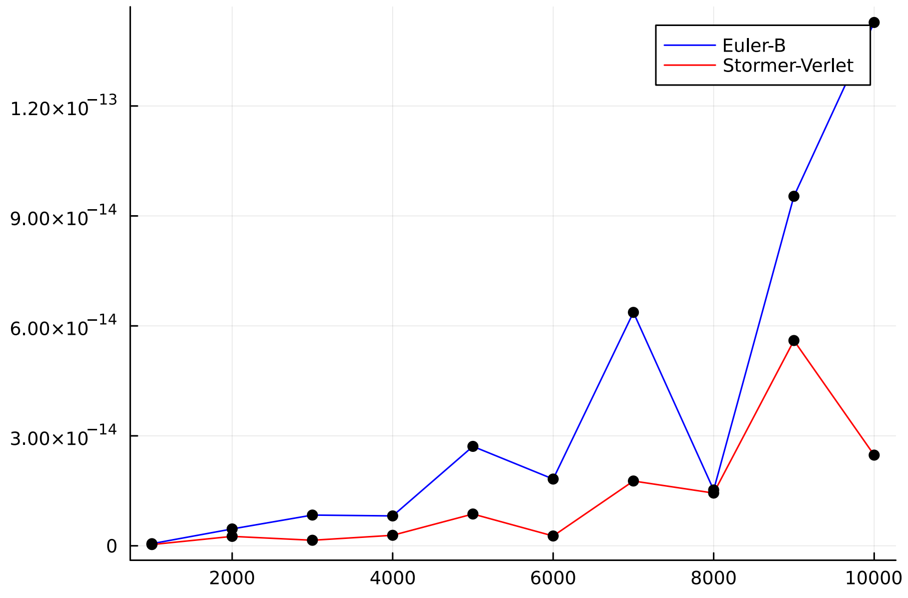
```
dts = [1000*i for i in 1:10]
_, Ls = get_max_error_dt(euler_b, dts, 10000)

plt = plot(dts, Ls)
scatter!(plt, dts, Ls, color=:red)
display(plt)
```



It looks like for Euler-B, at $dt = 5000$ we start to loose conservation of angular momentum.

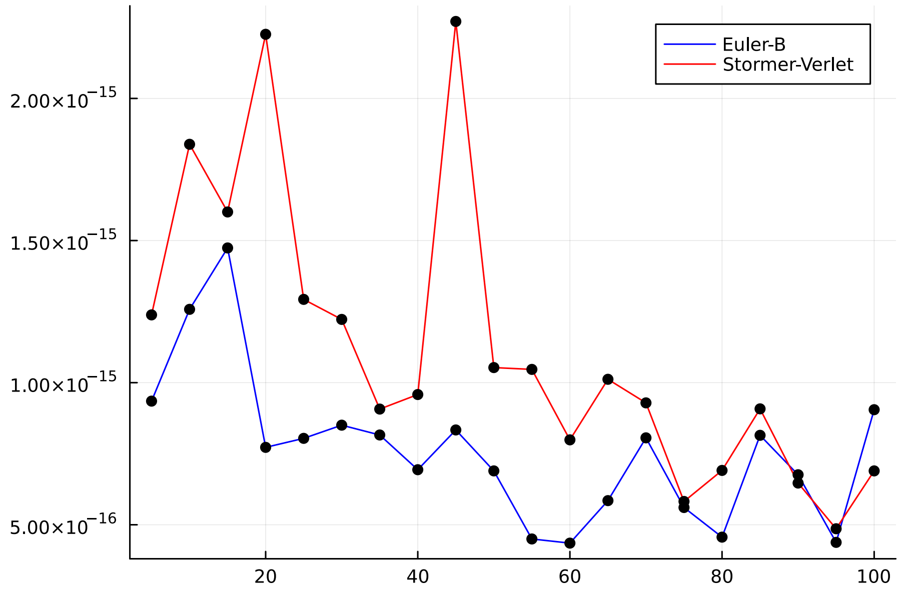
In [69]:

```
dts = [1000*i for i in 1:10]
_, Ls = get_max_error_dt(euler_b, dts, 10000)
_, Ls2 = get_max_error_dt(stormer_verlet, dts, 10000)

plt = plot(dts, Ls, label = "Euler-B", color =:blue)
plot!(plt, dts, Ls2, label="Stormer-Verlet", color=:red)
scatter!(plt, dts, Ls, color=:black, label = false)
scatter!(plt, dts, Ls2, color=:black, label = false)
display(plt)
```



In [70]:

```
dts = [5*i for i in 1:20]
_, Ls = get_max_error_dt(euler_b, dts, 10000)
_, Ls2 = get_max_error_dt(stormer_verlet, dts, 10000)

plt = plot(dts, Ls, label = "Euler-B", color =:blue)
plot!(plt, dts, Ls2, label="Stormer-Verlet", color=:red)
scatter!(plt, dts, Ls, color=:black, label = false)
scatter!(plt, dts, Ls2, color=:black, label = false)
display(plt)
```



It looks like the Euler-B method is a little better and conserving the angular momentum, but at such small scale both errors are essentially 0.

In [ ]: