# Astro 585: HW 2

Codename: The Maxwell-Jüttner Distribution

January 31, 2014

## 1 Common Function Benchmarks

```
#!/usr/bin/env julia

# 1
N = 10^7;
println("rand: ", 1./(@elapsed x = rand(N)));
println(".+:   ", 1./(@elapsed x.+x));
println(".*:   ", 1./(@elapsed x.*x));
println("./:   ", 1./(@elapsed x./x));
println("log:  ", 1./(@elapsed log(x)));
println("log10:", 1./(@elapsed log10(x)));
println("sin:  ", 1./(@elapsed sin(x)));
println("cos:  ", 1./(@elapsed cos(x)));
println("tan:  ", 1./(@elapsed tan(x)));
println("atan: ", 1./(@elapsed atan(x)));
round_down_to_power_of_ten(x) = 10.0.^floor(log10(x))
round_down_to_power_of_iten(x) = 10.0.^ifloor(log10(x))
println("rdtpot:  ", 1./(@elapsed round_down_to_power_of_ten(x)));
println("rdtopit: ", 1./(@elapsed round_down_to_power_of_iten(x)));
println("rdtpot:  ", 1./(@elapsed round_down_to_power_of_ten(x)));
println("rdtopit: ", 1./(@elapsed round_down_to_power_of_iten(x)));
```

Output:

```
rand: 5.467883391743815
.+:   0.7936923566738714
.*:   4.939859409921642
./:   2.0617212118723063
log:  0.7198937420711083
log10:0.7597877616464493
sin:  1.432453479819846
cos:  1.3486920641287916
tan:  0.670428735951879
atan: 0.7613800558965796
```

## 2 More Benchmarking

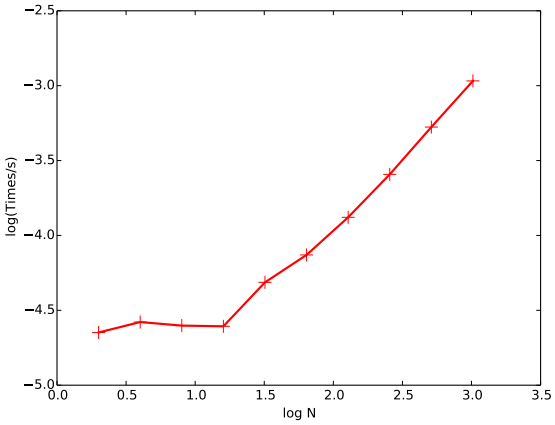
Figure of time versus number of times executed is a the end of the section.

```
#!/usr/bin/env julia
# To see the plots, this must be run interactively, i.e. include("filename")

# <codecell>

# 2
function calc_time_log_likelihood(Nobs::Int, M::Int = 1)
    srand(42);
    z = zeros(Nobs);
    sigma = 2. * ones(Nobs);
    y = z + sigma .* randn(Nobs);
    normal_pdf(z, y, sigma) = exp(-0.5 * ((y-z)/sigma)^2) / (sqrt(2*pi) * sigma);

    function log_likelihood(y::Array, sigma::Array, z::Array)
        n = length(y);
        @assert n == length(sigma);
        @assert n == length(z);
        sum = zero(y[1]);
        for i in 1:n
            sum += log(normal_pdf(y[i], z[i], sigma[i]));
        end;

        return sum;
    end;

    function m_times(y::Array, sigma::Array, z::Array, M::Int)
        sum = zero(y[1]);
        for i in 1:M
            sum += log_likelihood(y, sigma, z);
        end;

        return sum;
    end;

    return @elapsed m_times(y, sigma, z, M);
end;

function print_llik_MOPS(M::Int)
    println("log_likelihood ", M, " times: ",
                M * 1./calc_time_log_likelihood(10^6, M));
end;
```

```
print_llik_MOPS(1);
print_llik_MOPS(2);
print_llik_MOPS(10);

# <codecell>

# First time is always slower, probably since it must compile it first,
# but for subsequent runs it has the compilation cached.

# <codecell>

# 2c
#using PyCall
#pygui(true)
using PyPlot
n_list = [ 2^i for i = 1:10 ];
elapsed_list = map(calc_time_log_likelihood, n_list);
plot(log10(n_list), log10(elapsed_list), color="red", linewidth=2, marker="+",
                markersize=12);
xlabel("log N"); ylabel("log(Times/s)");

# <codecell>

# Initially, this is not linear for small array sizes, but for large
# ones it becomes fairly linear. For the future it is important to
# know what size of arrays will be expected.

# <codecell>

# 2e
println("log_likelihood 100 times without asserts: ",
                calc_time_log_likelihood(10^2, 10000));

# <codecell>

println("log_likelihood 100 times with asserts: ",
                calc_time_log_likelihood(10^2, 10000));

# <codecell>

# Barely a difference! (Even faster in this instance.)

# <codecell>

# 2f
```

```
normal_pdf(z, y, sigma) = exp(-0.5 * ((y-z)/sigma)^2) / (sqrt(2*pi) * sigma);

function calc_time_log_likelihood2(Nobs::Int, M::Int = 1, fn = normal_pdf)
    srand(42);
    z = zeros(Nobs);
    sigma = 2. * ones(Nobs);
    y = z + sigma .* randn(Nobs);

    function log_likelihood2(y::Array, sigma::Array, z::Array, fn)
        n = length(y);
        @assert n == length(sigma);
        @assert n == length(z);
        sum = zero(y[1]);
        for i in 1:n
            sum += log(fn(y[i], z[i], sigma[i]));
        end;

        return sum;
    end;

    function m_times2(y::Array, sigma::Array, z::Array, M::Int, fn)
        sum = zero(y[1]);
        for i in 1:M
            sum += log_likelihood2(y, sigma, z, fn);
        end;

        return sum;
    end;

    return @elapsed m_times2(y, sigma, z, M, fn);
end;

println("log_likelihood 100 times with function: ",
                calc_time_log_likelihood2(10^2, 10000, normal_pdf));

# <codecell>

# This is a little slower.
```

Time versus number of times executed:




## 3 Euler Leapfrog

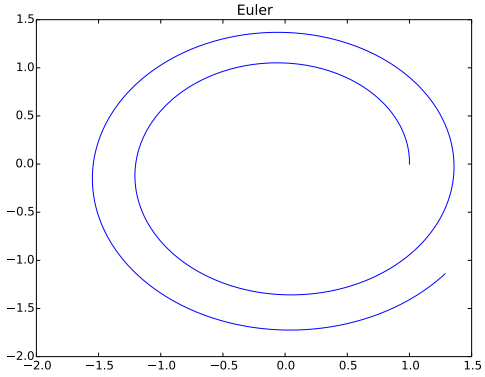
```
#!/usr/bin/env julia

# 3a)
function accel_grav(rvec)
    # assuming G*M = 1
    r = hypot(rvec...)
    return - rvec / r^3
end

function phase_space_speed(state)
    m = iround(length(state) / 2) # number of spatial dimensions
    ridx = 1:m
    vidx = (m+1):(2*m)

    speed = copy(state) # should probably cache this
    speed[ridx] = state[vidx]
    speed[vidx] = accel_grav(state[ridx])

    return speed
end

function integrate_euler!(state::Array, dt::Real, duration::Real)
    #assert(length(state) is even)

    n = iceil(duration / dt)
    m = length(state)
    all_states = Array(typeof(state[1]), n, m)
    all_states[1,:] = state
```

```
    for i in 2:n
        speed = phase_space_speed(state)

        state[:] += dt .* speed # one Euler step

        all_states[i,:] = state
    end

    return all_states
end


# 3b)
state = [ 1.0, 0.0, 0.0, 1.0 ]
all_states = integrate_euler!(state, 2*pi / 200, 3*2*pi)
function show_some_states(all_states)
    show(all_states[1:5,:])
    println("\n...")
    show(all_states[(end-4):end,:])
    println()
end
println("# 3b")
show_some_states(all_states)
# Seems completely wrong.


# 3c)
using PyPlot
x = all_states[:,1]
y = all_states[:,2]
figure()
plot(x, y)
title("Euler")
# The accuracy is, uhm, in need of improvement.


# 3d)
function integrate_leapfrog!(state::Array, dt::Real, duration::Real)
    #assert(length(state) is even)

    n = iceil(duration / dt)
    m = length(state)
    all_states = Array(typeof(state[1]), n, m)
    all_states[1,:] = state
```

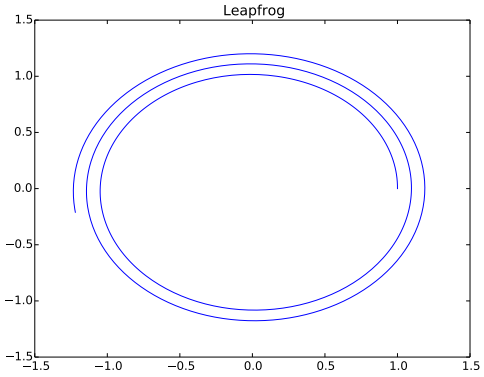
```
    m = iround(length(state) / 2) # number of spatial dimensions
    ridx = 1:m
    vidx = (m+1):(2*m)

    midstate = copy(state)

    for i in 2:n
        # leapfrog, the complicated way:
        midstate[ridx] = state[ridx] + dt/2 .* state[vidx]
        midstate[vidx] = state[vidx]
        state[vidx] += dt * accel_grav(midstate[ridx])
        midstate[vidx] = (midstate[vidx] + state[vidx]) / 2
        state[ridx] = midstate[ridx] + dt/2 .* midstate[vidx]

        all_states[i,:] = state
    end

    return all_states
end

# 3e)
state = [ 1.0, 0.0, 0.0, 1.0 ]
all_states = integrate_leapfrog!(state, 2*pi / 200, 3*2*pi)
println("\n# 3e")
show_some_states(all_states)
# Also quite wrong
x = all_states[:,1]
y = all_states[:,2]
figure()
plot(x, y)
title("Leapfrog")
# But the plot shows that it is much closer to the right answer, as the
# computed answer orbits about 2.5 times, not just under 2 times. Also, the
# energy is much closer to being conserved. I would have expected it to be much
# better conserved upon completion of an orbit, but since this is a homework, I
# am not inclined to continue debugging the algorithm.


# 3f)
println("\n# 3f")
state = [ 1.0, 0.0, 0.0, 1.0 ]
@time integrate_euler!(state, 2*pi / 200, 3 * 2 * pi)
# Takes only 0.002158235 seconds
time = @elapsed integrate_euler!(state, 2*pi / 200, 4.5e3 * 2 * pi)
println("Time for 4.5e3 orbits: ", time, " seconds")
```

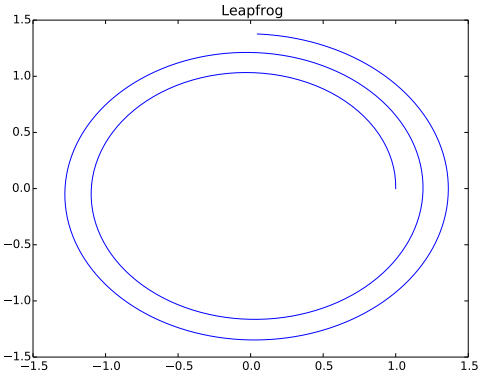
```
println("Time for 4.5e9 orbits: ", time * 1e6, " seconds", " = ",
                time*1e6 / 3600 / 24, " days")
# Takes only 6.76 seconds for 4.5e3 orbits, so for 4.5e9 I estimate 6.76e6
# seconds, or 78 days. Darn.


# 3g)
println("\n# 3g")
state = [ 1.0, 0.0, 0.0, 1.0 ]
all_states = integrate_leapfrog!(state, 2*pi / 100, 3*2*pi) # double dt from before
show_some_states(all_states)
# 2.25 orbits instead of 2.5.
x = all_states[:,1]
y = all_states[:,2]
figure()
plot(x, y)
title("Leapfrog")
```

Using Euler's method:




Using the leapfrog method:



# 4 Memory and Speed

**4a)** Memory is $2^{32}$ bytes, size of one float is $2^3$ bytes, so maximum is $2^{29} = 5.36870912 \times 10^8$floats, so largest matrix would be any rectangular one with that many entries, such as $2^{14} \times 2^{15}$. The largest square matrix would be $23170 \times 23170$, with a few bytes (176096 bytes) to spare for the program, and possibly an operating system...

**4b)**

$$n = 23170$$

Number of computations needed is

$$m = \frac{2}{3}n^3 = 8.29 \times 10^{12}$$

Time is approximately

$$t = \frac{m}{r} = \frac{8.29 \times 10^{12}}{20 \times 10^9 \,\text{flops/s}} = 414.626\,300\,433\,333\,3\,\text{s}$$

A few minutes.

**4c)** Memory seems to be a more stringent constraint, although I am not so sure that 20 Gflops/s is so realistic unless there is significant parallelization. Also, the floats need to be transferred from the memory to the CPU and back, which is likely going to be much slower than the 20 Gflops/s.

**4d)** The phase of the Moon!

**4e)** One could come up with an algorithm that operates only on parts of the original matrix at a time, such as splitting the matrix into blocks such that a few blocks fit into memory. Blocks that are not needed could be stored on disk, or on other nodes in a cluster. Either one will introduce extra overhead reading from disk or transferring blocks from one node to another.