CO61: Rocket Science

Gareth Everton – wadh7166
Date: 2025-03-05

1 AIM and Method

The main aim of this lab is to implement a basic numerical time integrator, allowing us to progress a simple satellite in a two-body system, in this case the Earth-Moon system, allowing us to predict the path of a satellite and find one that lands on the moon.

To improve the accuracy of and stability of our numerical integrator we implemented and improved Euler method. This increases computation time but greatly reduces the error introduced when using discrete timesteps. A further improvement to accuracy could be obtained by switching to and even more precise numerical method, such as the RK-4 algorithm.

Once this was implemented in the simulate_rocket function, we were asked to find a suitable launch angle for a situation in which we no longer simulate the moon as a static object; we consider the moon to be in a circular orbit. To do this, we first brute force simulate 5 different trajectories for launch angles ranging from 0 to π, and then we picked the trajectory with the lowest periapsis with respect to the moon. Once this was determined we sample a trajectory with similar initial conditions to our best guess, and use the values returned to approximate the rate of change of periapsis with respect to our initial conditions. We can then use this "gradient" to inform our next guess of initial conditions so that our periapsis will be smaller, and we simply iterate through this process until we find a suitable trajectory that lands on the moon.

This increase in the number of trajectories brings to light several performance considerations when writing the code. The return value of the simulate function is a large array of doubles storing the positions of the spacecraft at a given time. For large time integration regimes we observe that this can cause a large memory usage, and additionally the single-threaded nature of MATLAB code means that the approx. 200000 iterations per trajectory takes around 10 seconds to compute. This becomes especially apparent in the second section where we simulate multiple trajectories to find one which lands on the moon.

Finally, throughout the process of finding a suitable trajectory we plot the current calculated trajectory as a coloured line, with the colour of a point corresponding to the time at which the satellite was at that position. This has the advantage of communicating a path through 3D space-time in clear 2D way, and allows the user to see how close the code is to producing a suitable trajectory.

2 Results

Running the code with the moon in the circular orbit described in the lab script we obtain an optimal launch angle of 53 degrees, giving us a lunar periapsis of 0.0433 lunar radii; This is most likely not the best trajectory for a realistic case, given most craft want to perform and efficient decent to the lunar surface by inserting into an orbit around the moon with an insertion burn, and then perform a controlled decent from this orbit. However our trajectory performs a direct descent and will most likely be much more inefficient than a more time consuming insertion and controlled descent. One way in which we could improve the code is for it to factor the relative velocity of the satellite to the moon into its factors for evaluating what is a "good" orbit.

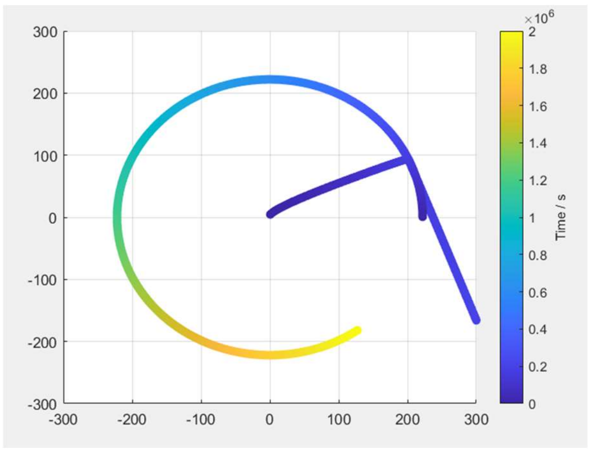

Figure 1. Final Trajectory produced by code, theta = 53 degrees

3 Performance

Performance of the code is not that great, a single satellite iteration running on my laptop (Intel i7-1195G7) takes between 7 and 10 seconds. If we assume a best case scenario, our code is able to simulate 2×10^5 iterations in 7 seconds, giving us a performance of about 3×10^4 iterations per second. By moving to a lower level programming language and simulating multiple satellites at once we can start to take advantage of the parallel computation abilities of modern hardware. In particular, if we design our code to run on the graphics processing unit (GPU) of a given machine, we can see huge performance gains. This is due to the design of a GPU die, graphics rasterization is a task that is inherently suited to parallel computation and floating point operations on pixels on screens, which is not too dissimilar to our task. By converting our simulation function to GLSL shader code and writing a C++ Vulkan Engine to support these shaders, I was able to simulate 4096 satellites over 1.2e06 seconds with a 10s timestep in ~22 seconds average, with a program startup time of ~2s as well. This is equivalent to around 22×10^6 iterations per second. This is over a 730x improvement in performance, and is due to us better utilising the computational power of our hardware[1].

4 Conclusion

Overall the code executed successfully and was able to find a good trajectory to land on the moon in reasonable time, although several performance improvements could be made by switching to a faster programming language such as C++. Further performance gains could be obtained by moving to a parallel programming model, and transitioning the simulation to run on a GPU using a modern graphics API such as DirectX or Vulkan.

1: Source code for Vulkan Engine can be found below, Accessed 05/03/2025 12:15
https://github.com/EvertonTGV2006/GravSim/tree/SolarSystemSplit

Appendix 1: MATLAB Code

```
function [ tout , pos, rMin ] = simulate_rocket ( init_pos ,
init_vel , moon_pos , t)
    % Author: Gareth , Date: 28/02/25
    % Simulate the rocket trajectory with the Earth and Moon
influence. The coordinate
    % used in this function is centred at Earth’s centre (i.e.
Earth centre at (0,0) )
    % and scaled in Moon-radius.
    % The simulation finishes when it simulates for the whole t, or
the rocket landed
    % on the Moon.
    % Input:
    % * init_pos: 2-elements vector (x, y) indicating the initial
position of the rocket.
    % * init_vel: 2-elements vector (vx, vy) of the initial
velocity of the rocket.
    % * moon_pos: a function that receives time, t, and return a
2-elements vector (x, y)
    % (see hint) indicating the Moon position relative to Earth.
    % * t: an N-elements vector of the time step where the position
of the rocket will be
    % returned.
    %
    % Output:
    % * tout: an M-elements vector of the time step where the
position is described,
    % if the rocket does not land on the Moon, M = N.
    % * pos: (M x 2) matrix indicating the positions of the rocket
as function of time,
    % with the first column is x and the second column is y.
    %
    % Example use:
    % >> init_pos = [0, 3.7];
    % >> init_vel = 0.0066 * [cosd(89.9), sind(89.9)];
    % >> moon_pos = @(t) [0, 222];
    % >> t = linspace(0, 10000, 1000);
    % >> [tout, pos] = simulate_rocket(init_pos, init_vel,
moon_pos, t);
    % >> plot(

    tic %Start iteration timer

    timesteps = length(t); %Number of timesteps is the count of the
number of times given in array t;

    pos(1,:) = init_pos; %set starting position in p array
    v_0 = init_vel; %Set v_0 as inital velocity, initialise values
for euler integrations

    G = 9.63e-7; %Constant value of G
    rEarth = [0, 0]; %Fix Position of Earth to be 0, ingnore effect
of moon's gravity on Earth
    MEarth = 83.3; %Mass of the Earth set to be 83.3
    MMoon = 1; %Mass of the moon set to be 1;

    rMin = 222; %Set large inital value of rMin so that it can be
beaten by better orbits

    tout = zeros(size(t)); %Initialize array of t values as 0s

    for i = 2:timesteps

        p_0 = pos(i-1,:); %Starting position for iteration

        %First Euler iteration
        r_1 = p_0 - rEarth; %Find Earth-Sat separation
        a_1 = -G * MEarth * r_1 / (sum(r_1.^2).^(3/2)); %Compute
acceleration due to Earth
        rMoon = moon_pos(t(i-1)); %Get Moon Position in space for
given time
        r_1 = p_0 - rMoon; %Get Moon-Sat Separation
        a_1 = -G * MMoon * r_1 / (sum(r_1.^2).^(3/2)) + a_1; %Add
acceleration due to Moon
        dt = t(i) - t(i-1); %Compute timestep dt
        dv_1 = a_1 * dt; %find dv_1
        v_1 = v_0 + dv_1; %Increment velocity
        dp_1 = v_1 * dt; %Find dp
        p_1 = p_0 + dp_1; %Increment Positions

        %2nd Euler iteration, same code as above but using v_1 and
p_1 and t(i) instead
        r_2 = p_1 - rEarth;
        a_2 = -G * MEarth * r_2 / (sum(r_2.^2).^(3/2));
        rMoon = moon_pos(t(i));
        r_2 = p_1 - rMoon;
        a_2 = -G * MMoon * r_2 / (sum(r_2.^2).^(3/2)) + a_2;
        dt = t(i) - t(i-1);
        dv_2 = a_2 * dt;
        v_2 = v_0 + dv_2;
        dp_2 = v_2 * dt;


        dv_t = 0.5* (dv_1 + dv_2); %Combine dv_x into total dv_t
        dp_t = 0.5* (dp_1 + dp_2); %Same for position

        p_t = p_0 + dp_t; %Update position
        v_t = v_0 + dv_t; %Update velocity


        v_0 = v_t; %Write new velocity to velocity for next
iteration
        pos(i,:) = p_t; %Store position

        tout(i) = t(i); %Write out timestep
        rMin = min(rMin, sqrt(sum(r_2.^2))); %Compute rMin for
return value

    end

    toc %Output elapsed time for iteration
end

function plotPath(t, p, moon_pos)
     %Author: Gareth     Date: 28/02/25
     %Input Paramaters:
     %t: M size array of double values indicating t values at
positions descriped in p
     %p: M x 2 size array of double values indicating x and y
positions in space at time t
     %moon_pos: function handle to function that takes input value
of t and returns x and y coordinates of the moon.

    hold off; %Hold off to clear screen for new plot
    x = p(:,1); %Initialize x and y arrays for plot
    y = p(:,2);
    col = t; %Set colour of line to time at that position
    scatter(x, y, [], col, 'fill'); %plot scatter of positions with
colour gradient showing dt for input position
    hold on;
    %Keep plot on screen


    %plot moon;
    mp = zeros(length(t),2); %Initialize array for position of moon
    for i = 1:length(t)
        mp(i,:) = moon_pos(t(i)); %Populate moon position vector mp
with values for given postions at time t
    end

    mx = mp(:,1); %Initialsie x and y values for the moon, col
still the same as used above
    my = mp(:,2);
    scatter(mx, my, [], col, 'fill'); %Same scatter plot but for
moon
```

```
    axis([-300 300 -300 300]) %Set axis of plot so that it is
scaled properly
    c = colorbar; %Add a colorbar to show timescale of orbits
    c.Label.String = 'Time / s'; %Label colorbar with Time values
    grid on; %Set the grid to be on
    shg; %Show the completed figure to the user.
end

moon_pos = @(t) [222*cos(2.6615e-6*t), 222*sin(2.6615e-6*t)];
% moon_pos = @(t) [0, 222];

function [t, p, theta] = findLaunchAngle(moon_pos)
    %Author: Gareth     Date: 28/02/25
    %Input parameters: moon_pos: handle to function that returns a
2D position vector for the moon for a given input time t;
    v0 = 0.0066; %Initial velocity
    thetas = 0:pi/4:pi; %Seed initial "guesses" for theta
    rMin = 222; %Set rMin closest approach to start value
    thetaRes = 0; %resultant theta
    for i = 1:length(thetas)
        v = [v0 * cos(thetas(i)), v0 * sin(thetas(i))]; %Convert v0
and theta to vector
        [t, p, rMr] = simulate_rocket([0, 3.7], v, moon_pos,
0:10:2000000); %Simulate trajectory using simulate_rocket function
as above
        plotPath(t, p, moon_pos); %Plot the resultant path so that
the user can visualise the currently calculated trajectory;
        thetas(i) %Print out current theta
        rMr %Print out minimun radius obtained from the current
simulation iteration
        if rMr < rMin %If this closes approach is better than
previous ones, save the value of theta for use later, and update
new closes approach
            thetaRes = thetas(i);
            rMin = rMr;
        end
    end
    dtheta = 0.1 %Print seed value of dtheta
    theta2 = thetaRes + dtheta; %Now next guess of theta is based
on previous guess and dtheta
    drdt = 0; %Rate of change of r wrt theta, where r is closest
approach to moon.
    for i = 1:10 % set a limit of 10 iterations
        v = [v0 * cos(theta2), v0 * sin(theta2)]; %Calculate seed
velocity from v0 and theta
        [t, p, rMr2] = simulate_rocket([0, 3.7], v, moon_pos,
0:10:2000000); %simulate trajectory
        plotPath(t, p, moon_pos); %Plot path for user
        drdt = (rMr2 - rMr )/dtheta %Print rate of change of
closest approac wrt theta for user information
```

```
        dtheta = -rMr2 / drdt %calculate new dtheta for drdt
        if (dtheta > 0.1)
            dtheta = 0.1
            dtheta = dtheta * rMr2; %Do some bounds checks so that
dtheta is not too large, this adds stability to the simulation
        end
        if (dtheta < -0.1)
            dtheta = -0.1
            dtheta = dtheta * rMr2;
        end

        theta2 = theta2 + dtheta %Print new theta
        rMr = rMr2 %Update disrtance of closes approach
        if(rMr < 0.1) %Check if we have met our success criteria of
being within 0.1 radii of the moon.
            break
        end
    end
    theta = theta2; %Return best value of theta that was found

end


[t, p, theta] = findLaunchAngle(moon_pos); %Find best launch angle
for rocket.

"Best launch angle: " %Print out best launch angle
theta_deg = theta * 180 / pi
```

Appendix 2: GLSL Shader Code

```
//Requires GPU that supports the Vulkan shaderFloat64 Feature.
#version 460

struct Planet{
    //0-th is position at current time, 1-st is position at T+.5dt,
2-nd is position at T+dt
    dvec4 pos[3];
    dvec4 vel[3];
    dvec4 axis;
    dvec4 padding;
};
struct Satellite{
    dvec4 pos;
    dvec4 vel;
};
struct LineVertex{
    vec4 pos;
    vec4 base;
    vec4 inter;
    vec4 fin;
};
struct LineInfo{
    float eccentricity;
    float apoapsis;
    float periapsis;
    float cost;
};
struct SatInfo{
    dvec4 pos;
    dvec4 vel;
    double relPos;
    double relVel;
    double tApproach;
    double score;
};


layout(constant_id = 0) const double G = 6.67e-11;
layout(constant_id = 1) const uint MAX_PLANETS_ARRAY_SIZE = 2;
layout(constant_id = 2) const uint SATELLITE_COUNT = 256;
layout(constant_id = 3) const uint LINE_VERTEX_COUNT = 1024;
layout(constant_id = 4) const uint COMPUTE_STEPS_PER_FRAME = 256;
layout(constant_id = 5) const uint MESH_STEPS_MAJOR = 16;
layout(constant_id = 6) const uint MESH_STEPS_MINOR = 8;
layout(constant_id = 7) const uint SATELLITES_PER_SHADER = 1;
layout(constant_id = 8) const uint STEPS_PER_SHADER = 1;
layout(constant_id = 9) const uint MESH_PER_SHADER = 1;

layout(binding = 0) readonly buffer UniformBufferObject{
    Planet planets[MAX_PLANETS_ARRAY_SIZE * COMPUTE_STEPS_PER_FRAME
* STEPS_PER_SHADER];
} ubo;

layout(binding = 1) readonly buffer inSSBO{
    Satellite inSats[SATELLITE_COUNT];
};
layout(binding = 2) buffer outSSBO{
    Satellite outSats[SATELLITE_COUNT];
};
layout(binding = 3) buffer lineSSBO{
    LineVertex vertices[SATELLITE_COUNT * LINE_VERTEX_COUNT];
};
layout(binding = 4) buffer lineInfoSSBO{
    LineInfo infos[SATELLITE_COUNT];
};
layout(binding = 5) buffer meshSSBO{
    LineVertex mesh[MESH_STEPS_MINOR * MESH_STEPS_MAJOR * 2];
};
```

```
layout(binding = 6) buffer satDataSSBO{
    SatInfo satInfos[SATELLITE_COUNT];
};
layout(push_constant) uniform pc {
    double dt;
    double tElapsed;
    uint targetPlanetIndex;
    uint pOffset;
} constants;
layout(local_size_x = 1024) in;
//FOR SATELLITE INTEGRATION

uint STEP_INDEX = 0;
// vec3 evaluatePlanetAccelEuler(Satellite sat, Planet pl, uint
index){ //OLD CODE, OBSOLETE REPLACED BY evaluateAccelBase;
//     vec3 sep = sat.pos.xyz - pl.pos[index].xyz;
//     float aMult = G * pl.pos.w / dot(sep, sep);
//     vec3 a = aMult * -normalize(sep);
//     return a;
// }
dvec3 evaluateAccelBase(Satellite sat, uint index){
    dvec3 a = vec3(0);
    uint planetIndexStart = constants.pOffset *
MAX_PLANETS_ARRAY_SIZE * STEPS_PER_SHADER + STEP_INDEX *
MAX_PLANETS_ARRAY_SIZE;
    uint planetIndexStop = constants.pOffset *
MAX_PLANETS_ARRAY_SIZE * STEPS_PER_SHADER + (STEP_INDEX + 1) *
MAX_PLANETS_ARRAY_SIZE;
    for(uint j = planetIndexStart; j < planetIndexStop; j++){
        dvec3 sep = sat.pos.xyz - ubo.planets[j].pos[index].xyz;
        double aMult = G * ubo.planets[j].pos[2].w / dot(sep, sep);
        a += aMult * - normalize(sep);
    }
    return a;
}

// Satellite evaluatePlanetAccelEulerImproved(Satellite sat, Planet
pl){ //OBSOLETE, REPLACED BY updateSatellitePositionEulerImproved;
//     //requires rethink  of planet positions, need planet
position data to be available for several timesteps;
//     //but for now we take planet position data as separate
inputs
//     //
//     vec3 dv_1 = evaluateAccelBase(sat, 0) * constants.dt;
//     vec3 dx_1 = sat.vel.xyz * constants.dt;
//     Satellite sat_1 = sat;
//     sat_1.pos.xyz = sat.pos.xyz + dx_1;
//     vec3 dv_2 = evaluateAccelBase(sat_1, 2) * constants.dt;
//     vec3 dx_2 = (sat.vel.xyz + dv_1) * constants.dt;
//     vec3 dv = 0.5f * dv_1 + 0.5f * dv_2;
//     vec3 dx = 0.5f * dx_1 + 0.5f * dx_2;
//     Satellite sat_2 = sat;
//     sat_2.pos.xyz += dx;
//     sat_2.vel.xyz += dv;
// }

Satellite updateSatellite_Euler(Satellite sat_0){
    Satellite sat_1 = sat_0;

    dvec3 dv = evaluateAccelBase(sat_0, 0) * constants.dt;
    dvec3 dx = sat_0.vel.xyz * constants.dt;

    sat_1.vel.xyz = sat_0.vel.xyz + dv;
    sat_1.pos.xyz = sat_0.pos.xyz + dx;

    return sat_1;
}

Satellite updateSatellite_EulerImproved(Satellite sat_0){
    Satellite sat_1 = sat_0;
    Satellite sat_2 = sat_0;

    dvec3 dv_1 = evaluateAccelBase(sat_0, 0) * constants.dt;
    dvec3 dx_1 = sat_0.vel.xyz * constants.dt;
    sat_1.vel.xyz = sat_0.vel.xyz + dv_1;
    sat_1.pos.xyz = sat_0.pos.xyz + dx_1;

    dvec3 dv_2 = evaluateAccelBase(sat_1, 2) * constants.dt;
    dvec3 dx_2 = (sat_0.vel.xyz + dv_1) * constants.dt;

    dvec3 dv = (dv_1 + dv_2) / 2;
    dvec3 dx = (dx_1 + dx_2) / 2;

    sat_2.vel.xyz = sat_0.vel.xyz + dv;
    sat_2.pos.xyz = sat_0.pos.xyz + dx;

    return sat_2;
}

Satellite updateSatellite_RK4(Satellite sat_0){
    Satellite sat_1 = sat_0;
    Satellite sat_2 = sat_0;
    Satellite sat_3 = sat_0;
    Satellite sat_4 = sat_0;

    dvec3 dv_1 = evaluateAccelBase(sat_0, 0) * constants.dt;
    dvec3 dx_1 = sat_0.vel.xyz * constants.dt;
    sat_1.vel.xyz = sat_0.vel.xyz + 0.5 * dv_1;
    sat_1.pos.xyz = sat_0.pos.xyz + 0.5 * dx_1;
    dvec3 dv_2 = evaluateAccelBase(sat_1, 1) * constants.dt;
    dvec3 dx_2 = (sat_0.vel.xyz + 0.5 * dv_1) * constants.dt;
    sat_2.vel.xyz = sat_0.vel.xyz + 0.5 * dv_2;
    sat_2.pos.xyz = sat_0.pos.xyz + 0.5 * dx_2;

    dvec3 dv_3 = evaluateAccelBase(sat_2, 1) * constants.dt;
    dvec3 dx_3 = (sat_0.vel.xyz + 0.5 * dv_2) * constants.dt;
    sat_3.vel.xyz = sat_0.vel.xyz + dv_3;
    sat_3.pos.xyz = sat_0.pos.xyz + dx_3;

    dvec3 dv_4 = evaluateAccelBase(sat_3, 2) * constants.dt;
    dvec3 dx_4 = (sat_0.vel.xyz + dv_3) * constants.dt;

    dvec3 dv = (dv_1 + 2 * dv_2 + 2 * dv_3 + dv_4) / 6;
    dvec3 dx = (dx_1 + 2 * dx_2 + 2 * dx_3 + dx_4) / 6;

    sat_4.vel.xyz = sat_0.vel.xyz + dv;
    sat_4.pos.xyz = sat_0.pos.xyz + dx;

    return sat_4;
}


void main(){
    uint i = gl_GlobalInvocationID.x;
//     vec3 a = vec3(0); //OLD CODE, REPLACED BY CODE BELOW
//     for(uint j = 0; j < MAX_PLANETS_ARRAY_SIZE; j++){
//         a += evaluatePlanetAccelEuler(inSats[i],
ubo.planets[j]);
//     }
//     outSats[i] = inSats[i];
//     outSats[i].pos.xyz = inSats[i].pos.xyz + inSats[i].vel.xyz *
constants.dt;
//     outSats[i].vel.xyz = inSats[i].vel.xyz + a * constants.dt;

    for(uint s = 0; s < SATELLITES_PER_SHADER; s++){
        uint satIndex = SATELLITES_PER_SHADER * i + s;
        if(satIndex < SATELLITE_COUNT){
            //EULER:
            //outSats[satIndex] =
updateSatellite_Euler(inSats[satIndex]);

            //EULER IMPROVED
            //outSats[satIndex] =
updateSatellite_EulerImproved(inSats[satIndex]);

            //RK-4
            //outSats[satIndex] =
updateSatellite_RK4(inSats[satIndex]);
```

```
            //RK-4 multipleStep
            Satellite tempSats[2];
            tempSats[0] = inSats[satIndex];
            uint satIndexIn = 0;
            uint satIndexOut = 1;
            for(uint i = 0; i < STEPS_PER_SHADER; i++){
                tempSats[satIndexOut] =
updateSatellite_RK4(tempSats[satIndexIn]);
                satIndexIn = (satIndexIn + 1) % 2;
                satIndexOut = (satIndexOut + 1) % 2;
                STEP_INDEX++;
            }
            outSats[satIndex] = tempSats[satIndexIn];

        }
    }
}
```