**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.

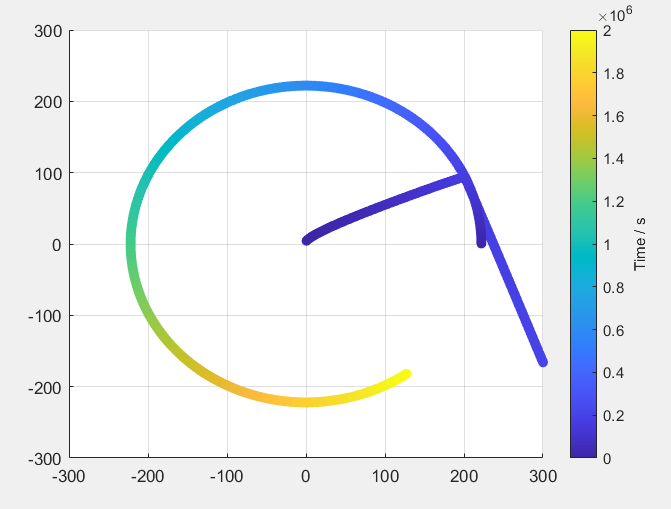
Once this was implemented in the simulate\_rocket function, we were asked to find a suitable launch angle for a situation in which we no longer simulate the moon as a static object; we consider the moon to be in a circular orbit. To do this, we first brute force simulate 5 different trajectories for 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 iterations in 7 seconds, giving us a performance of about 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 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];

}

}

}