A Summer Vacation Project for IB Engineering

Assignment Report

By Lorcan Nicholls, 2021

Assignment 1

Numerical Dynamics in One Dimension

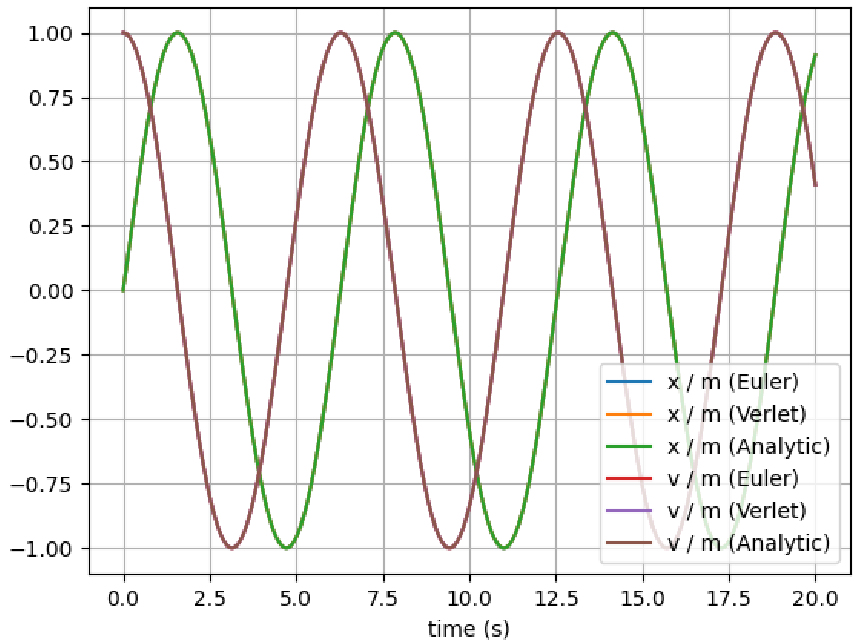
Implement Verlet integration on the following system of differential equations:

$$\dot{v} = -\frac{k}{m}x \ , \qquad \dot{x} = v$$

This was done using the Euler step for the first iteration (already initialised in the code below) and Verlet for all subsequent steps. Since the velocity is one index behind displacement, use the given approximation to find the final value.

```
x, v, a = [x, x + dt * v], [v], [a, -k * (x + dt * v) / m]

for i, t in enumerate(t_array[2:], start=2):

    x.append(2 * x[i-1] - x[i-2] + dt**2 * a[i-1])
    v.append((x[i] - x[i-2]) / (2 * dt))
    a.append(-k * x[i] / m)

else:
    v.append((x[i] - x[i-1]) / dt)
```

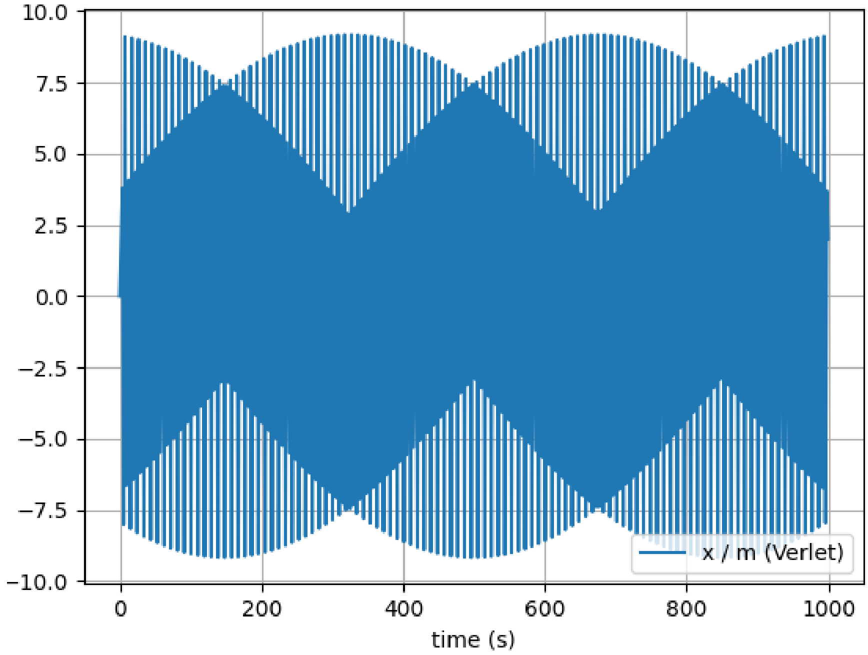
For small enough values of `t`, using `dt = 1e-4`, the curves all overlap well:



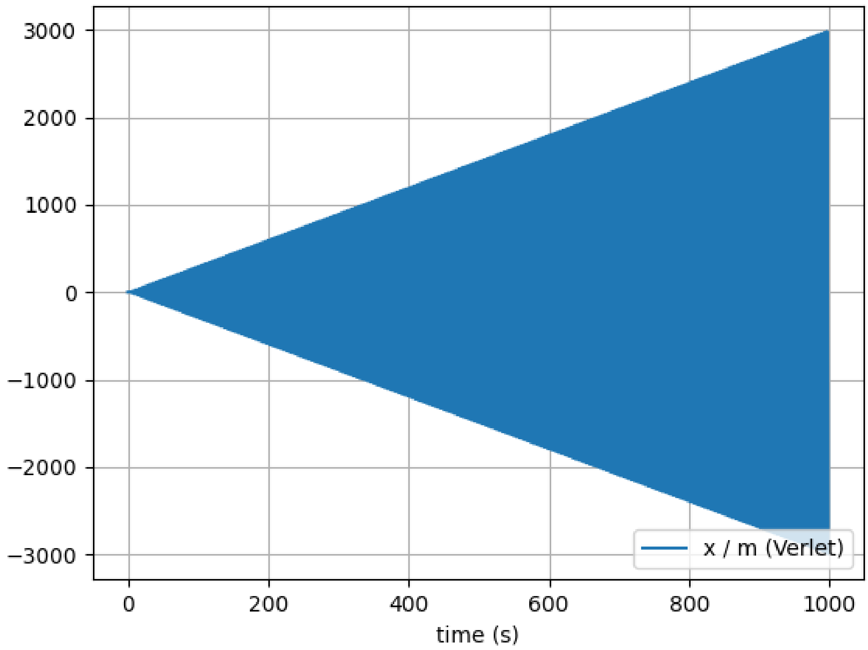
When `t_max = 1000`, the Verlet solution is stable for all `dt` < 2, becoming unstable for `dt` ≥ 2.

This was verified by graphing and agrees with analysis [1].

When `dt = 1.9`:



When `dt = 2`:



Assignment 2

Numerical Dynamics in Three Dimensions

Extend the Euler and Verlet integrators to three dimensions. Code for Euler:

```
def euler_integration(a_func: callable = None, t_max=10000, dt=1,
    x_0=(0, 1, 0), v_0=(1, 0, 0), a_0=(0, 0, 0),
    return_x_only: bool = True, as_arrays: bool = False):

    import numpy as np

    # initialise arrays
    x_euler, v_euler = [], []
    t_array = np.arange(0, t_max, dt)
    x, v, a = np.array(x_0, dtype=np.dtype('f8')), np.array(v_0,
              dtype=np.dtype('f8')), np.array(a_0, dtype=np.dtype('f8'))

    # initialise constants and define force-per-unit-mass (acceleration) field
    G, M = 6.67408e-11, 6.42e23
    a_field = lambda x: ((-1 * G * M) / np.linalg.norm(x)**3) * x if a_func is None else a_func

    # apply the Euler integration steps
    for t in t_array:

        x_euler.append(x)
        v_euler.append(v)

        a = a_field(x)
        x = x + dt * v
        v = v + dt * a

    # convert to arrays and return
    if as_arrays:
        x, v = np.array(x_euler), np.array(v_euler)
    else:
        x = [list(i) for i in x_euler]
        v = None if return_x_only else [list(i) for i in v_euler]

    return x if return_x_only else (x, v)
```

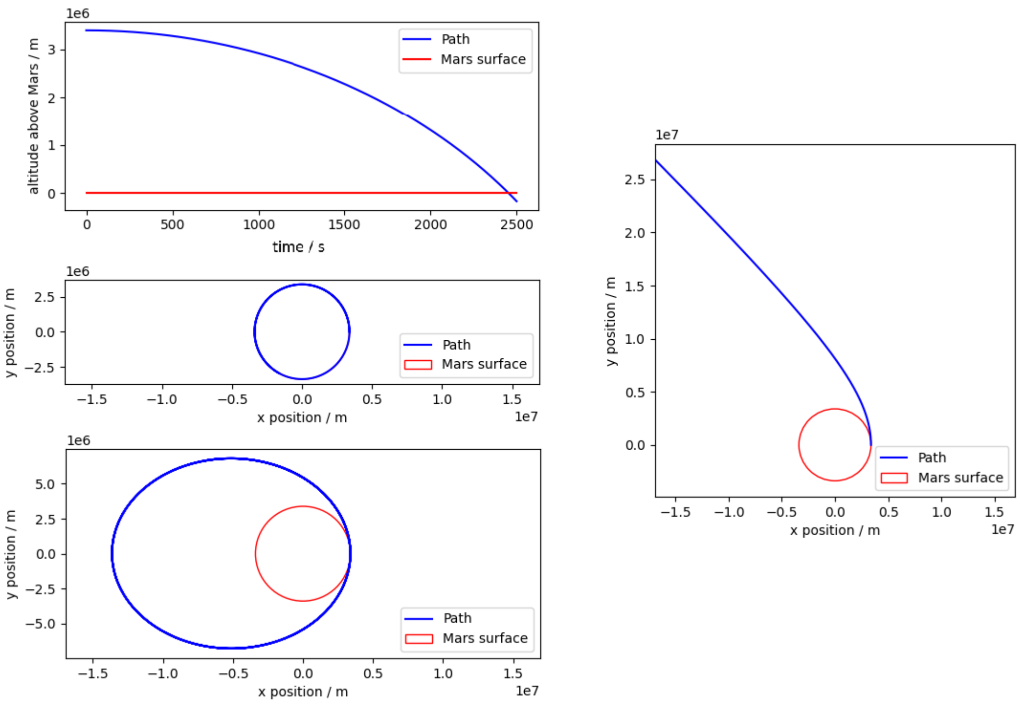
Code for Verlet:

```
def verlet_integration(a_func: callable = None, t_max=10000, dt=1,
    x_0=(0, 1, 0), v_0=(1, 0, 0), a_0=(0, 0, 0),
    return_x_only: bool = True, as_arrays: bool = False):

    import numpy as np

    # initialise arrays
    x_verlet, v_verlet = [], []
    t_array = np.arange(0, t_max, dt)
    x_0, v_0, a_0 = np.array(x_0, dtype=np.dtype('f8')), np.array(v_0,
         dtype=np.dtype('f8')), np.array(a_0, dtype=np.dtype('f8'))
    x, v, a = [x_0, x_0 + dt * v_0], [v_0], [a_0, a_0]

    # initialise constants and define force-per-unit-mass (acceleration)
    field
    G, M = 6.67408e-11, 6.42e23
    a_field = lambda x: ((-1 * G * M) / np.linalg.norm(x)**3) * x if a_func
              is None else a_func

    # apply the Verlet integration steps
    for n, t in enumerate(t_array[2:]):

        i = n + 2
        x.append(2 * x[i-1] - x[i-2] + dt**2 * a[i-1])
        v.append((x[i] - x[i-2]) / (2 * dt))
        a.append(a_field(x[i]))

    else:
        v.append((x[i] - x[i-1]) / dt)

    # return
    if not as_arrays:
        x = [list(i) for i in x]
        v = None if return_x_only else [list(i) for i in v]

    return x if return_x_only else (x, v)
```

General code for graphing:

```
MARS_RADIUS = 3.3895e6

results = verlet_integration(x_0=(MARS_RADIUS, 0, 0), v_0=(0, 10000, 0))
x, y, z = zip(*results)

plt.figure(1); plt.clf(); plt.grid(); ax = plt.gca(); ax.cla()
ax.set_xlim((-5 * MARS_RADIUS, 5 * MARS_RADIUS))
ax.set_aspect('equal', adjustable='box')

plt.xlabel('x position / m')
plt.ylabel('y position / m')

mars_outline = plt.Circle((0, 0), MARS_RADIUS, color='r', fill=False,
                          label='Mars surface')
ax.add_patch(mars_outline)

plt.plot(x, y, color='b', label='Path')
plt.legend(loc='lower right')
plt.show()
```

Plots - replacing `R = MARS_RADIUS` :



Top left: straight-down descent `v_0 = (0, 0, 0)` `x_0 = (2*R, 0, 0)`
Middle: circular orbit `v_0 = (0, 3555, 0)` `x_0 = (R, 0, 0)`
Bottom left: elliptical orbit `v_0 = (0, 4500, 0)` `x_0 = (R, 0, 0)`
Right: hyperbolic escape `v_0 = (0, 5000, 0)` `x_0 = (R, 0, 0)`

Assignment 3

Familiarisation with C++

Timings for Python were measured using the `timeit` module with 3 loops, while timings for C++ were read from the program output.

Optimisation was done using the `-O3` flag.

Tests were done on the Euler integrator for `t_max = 100; dt = 0.00001`, with low background usage of CPU and RAM, with specs:

- Intel i7-8565U @ 1.8 GHz; 8 GB RAM
- Windows 10, compiler: GNU GCC

Program Language	Average Time / s	Best Time / s
C++, unoptimised	0.600	0.583
C++, optimised	0.270	0.266
Python	4.455	4.390

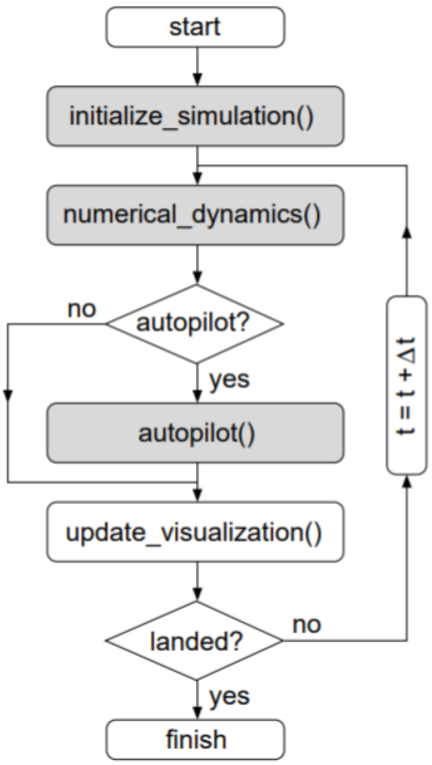
Verlet integration in C++:

```
t_list.push_back(0); t_list.push_back(dt);
x_list.push_back(x); x_list.push_back(x + dt * v);
v_list.push_back(v);
a_list.push_back(a); a_list.push_back(-k * x / m);

int i = 2;

for (t = 2 * dt; t <= t_max; t += dt) {
  t_list.push_back(t);
  x_list.push_back(2 * x_list[i-1] - x_list[i-2] + dt*dt * a_list[i-1]);
  v_list.push_back((x_list[i] - x_list[i-2]) / (2 * dt));
  a_list.push_back(-k * x_list[i] / m);
  i += 1;
}

v_list.push_back((x_list[i-1] - x_list[i-2]) / dt);
```

Assignment 4

Landing Craft Simulator

Implement the Euler/Verlet integrator into the lander project under the `numerical_dynamics` subroutine.



The new acceleration field includes the gravitational pull, the thrust and the atmospheric drag on the lander craft and the parachute (if deployed):

$$m\mathbf{a} = \text{thrust} + \text{gravity} + \text{atmospheric drag} + \text{parachute drag}$$

$$\mathbf{a} = \frac{\mathbf{T}}{m} - \frac{GM}{|\mathbf{r}|^2}\hat{\mathbf{r}} - \frac{1}{2m}\rho C_{D1} A_1 |\mathbf{v}|^2 \hat{\mathbf{v}} - \frac{1}{2m}\rho C_{D2} A_2 |\mathbf{v}|^2 \hat{\mathbf{v}}$$

The lander is assumed to have a constant projected area where the radius is `LANDER_SIZE`, i.e. independent of whether or not the lander is oriented parallel to its velocity. This could be improved by using $\hat{\mathbf{v}} \cdot \hat{\mathbf{d}}$ as the unit velocity vector, where $\hat{\mathbf{d}}$ is the unit orientation (surface normal) vector although it was considered unnecessary.

The lander has 5 parachutes, each modelled as squares with side length `2 * LANDER_SIZE`, again with constant projected area.

This is implemented using the various predefined constants:

```
lander_mass = (UNLOADED_LANDER_MASS + fuel * FUEL_CAPACITY * FUEL_DENSITY);
acceleration  = thrust_wrt_world() / lander_mass;  // thrust
acceleration -= (GRAVITY * MARS_MASS / position.abs2())
                * position.norm();  // gravity
acceleration -= 0.5 * DRAG_COEF_LANDER * atmospheric_density(position)
                * M_PI * LANDER_SIZE * LANDER_SIZE * velocity.abs2()
                * velocity.norm() / lander_mass;  // drag on lander
if (parachute_status == DEPLOYED) acceleration -= // drag on parachute
                10 * DRAG_COEF_CHUTE * atmospheric_density(position)
                * LANDER_SIZE * LANDER_SIZE * velocity.abs2()
                * velocity.norm() / lander_mass;
```

The Euler and Verlet integration schemes are then performed:

```
static vector3d previous_position;
vector3d new_position;

if (simulation_time == 0.0) {  // First iteration - use Euler
      new_position = position + velocity * delta_t;
      velocity += acceleration * delta_t;
}

else {  // Main iterations - use Verlet
      new_position = 2 * position - previous_position
                   + delta_t * delta_t * acceleration;
      velocity = (new_position - previous_position) / (2 * delta_t);
}

previous_position = position;
position = new_position;
```

Landing the craft in scenario 1 was surprisingly challenging but fun!

In scenario 4, the Verlet integrator shows a decaying elliptic orbit due to drag when passing through the atmosphere, while the Euler integrator does not decay, continuing in a steady elliptic orbit.

Assignment 5

Autopilot with Proportional Control

The autopilot was programmed for vertical descents only, without the parachute. The controller output took into account the thrust required to balance the variable weight and a target constant deceleration to a safe speed of 0.5 ms^{-1} as the craft landed.

The order-of-magnitude of K_h should be around 10^{-2} since it roughly determines the altitude at which the thruster kicks in and the target velocity at that point: initially set to 1000 ms^{-1} at 100 km and 50 ms^{-1} at 5 km.

Smaller values of K_h will trigger the thruster to fire earlier, risking running out of fuel. Larger values of K_h will delay the thruster, risking a crash landing during deceleration. A balance is needed to establish a safe landing.

The controller gain, K_p, sets the sensitivity of the autopilot, firing with more thrust when K_p is larger. Its value was not found to be particularly important, although it should be on the order of 1.

The autopilot subroutine was implemented as follows:

```
void autopilot (void)
  // Autopilot to adjust the engine throttle
{
    error_term = -(0.5 + K_h * altitude + velocity * position.norm());
    p_out = K_p * error_term;
    delta = (UNLOADED_LANDER_MASS + fuel * FUEL_CAPACITY * FUEL_DENSITY)
        * (GRAVITY * MARS_MASS / position.abs2()) / MAX_THRUST;
            // delta = mg, normalised to 0-1

    if (p_out <= -1 * delta) {
        throttle = 0;
    }
    else if (p_out < 1 - delta) {
        throttle = delta + p_out;
    }
    else {
        throttle = 1;
    }
}
```

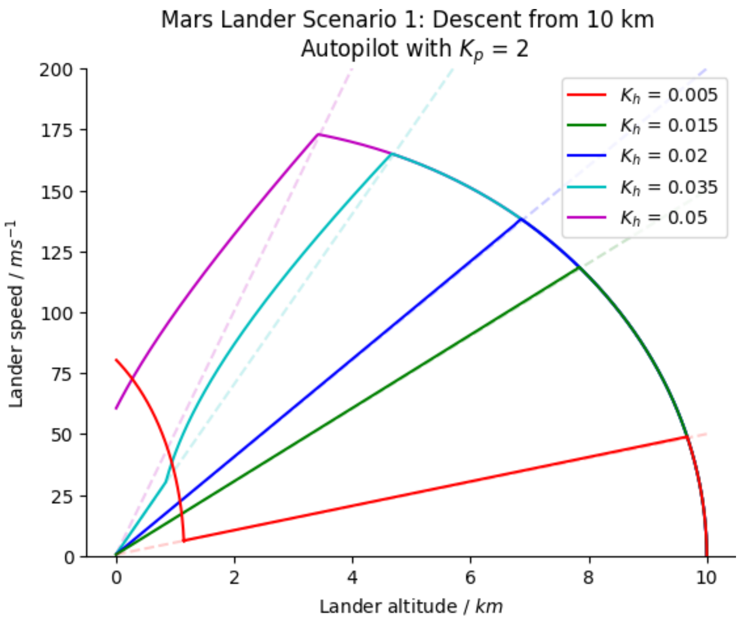
The data was then collected and exported to a `.txt` file by recording the altitude and speed as the `numerical_dynamics` subroutine executed:

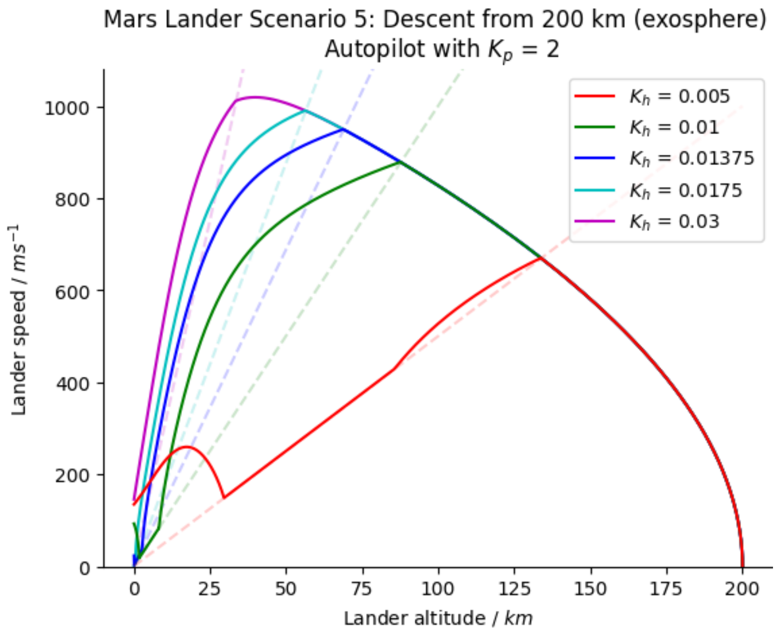
```
h_list.push_back(altitude);
v_list.push_back(velocity.abs());

// (rest of numerical dynamics code unchanged)

if (altitude < 100) { // choose a cut-off point to avoid messy crashing
                         data showing up or missing the endpoint
      ofstream fout;
      fout.open("trajectories_for_scenario_5_with_Kh=" + to_string(K_h)
                + ".txt");
      if (fout) {
          for (int i = 0; i < h_list.size(); i++) {
              fout << h_list[i] << ' ' << v_list[i] << endl;
          }
          exit(0);

      }
      else { // file did not open successfully
          cout << "Could not open trajectory file for writing" << endl;
      }
  }
```

Repeating with different values of K_h, these datasets were plotted in Python:

```
import os
import numpy as np
from matplotlib import pyplot as plt

# set scenario data to plot, the colours to use and the data directory
SCENARIO = 1
COLOURS = ['r', 'g', 'b', 'c', 'm']
FILEDIR = os.path.realpath(r'C:\Users\lnick\source\repos\lander\lander')

# find the text files
files = filter(lambda f: f.endswith('.txt') and 'scenario_' + str(SCENARIO)
               in f, os.listdir(FILEDIR))

# for each file, open it and plot the data on the same axis
for i, filename in enumerate(files):
    with open(os.path.join(FILEDIR, filename), 'r') as f:
        K_h = round(float(filename.split("=")[-1].split(".txt")[0]), 5)
        data = np.loadtxt(f)
        plt.plot(data[:, 0] / 1000, data[:, 1], label=f'$ K_h $ = {K_h}',
                 color=COLOURS[i], zorder=5-i)  # actual speed

        plt.plot([0, 10], [0.5, K_h * 10000], '--',
                 color=COLOURS[i], zorder=5-i, alpha=0.2)  # ideal speed

# graph properties
plt.title(f'Mars Lander Scenario {SCENARIO}: Descent from 10 km
            \n Autopilot with $ K_p $ = 2')
plt.xlabel('Lander altitude / $ km $')
plt.ylabel('Lander speed / $ ms^{-1} $')

# graphical improvements
plt.ylim((0, 200))
plt.legend(loc='upper right')
plt.gca().spines['right'].set_visible(False)
plt.gca().spines['top'].set_visible(False)

# save and display
plt.savefig('output_1', bbox_inches='tight', pad_inches = 0)
plt.show()
```

Autopilot descent plots - solid lines represent the actual plot, dashed lines represent the target descent speeds, for various values of K_h.





Scenario 1 was the easier of the two to get a successful landing, with a wide range of K_h allowing touchdown. The range for scenario 5 was much narrower, with the safe interval being approximately $0.015 < K_h < 0.020$.

This autopilot also worked well for Scenario 3, but not for Scenario 4 as the orientation of the craft in the decaying-elliptic path varied, so firing the thruster actually made it faster instead of slowing it down. With the given attitude stabilisation however, the vertical component of the velocity at crashing was brought down to 0.5 ms^{-1}.

Extensions

Some simpler additions

Modelling an areostationary orbit

Scenario 6 was set up to execute an areostationary (Martian geostationary) circular orbit. The radius of the orbit is found by equating centripetal acceleration with gravitational acceleration:

$$\frac{GM}{r^2} = r\omega^2 \quad \Rightarrow \quad r = \sqrt[3]{\frac{GM}{\omega^2}} = \sqrt[3]{\frac{GMT^2}{4\pi^2}}$$

The velocity was set perpendicular to the initial displacement, with $v = \omega r$.

```
case 6:
  // areostationary (martian geostationary) circular orbit
  altitude = cbrt(GRAVITY * MARS_MASS / (4 * (M_PI * M_PI) /
                  (MARS_DAY * MARS_DAY))) - MARS_RADIUS;
  position = vector3d(altitude + MARS_RADIUS, 0.0, 0.0);
  velocity = vector3d(0.0, 2 * M_PI / MARS_DAY * (altitude +
                      MARS_RADIUS), 0.0);
  orientation = vector3d(0.0, 90, 0.0);
  delta_t = 0.1;
  parachute_status = NOT_DEPLOYED;
  stabilized_attitude = true;
  autopilot_enabled = false;
  break;
```

Allow autopilot to use the parachute and land in all scenarios

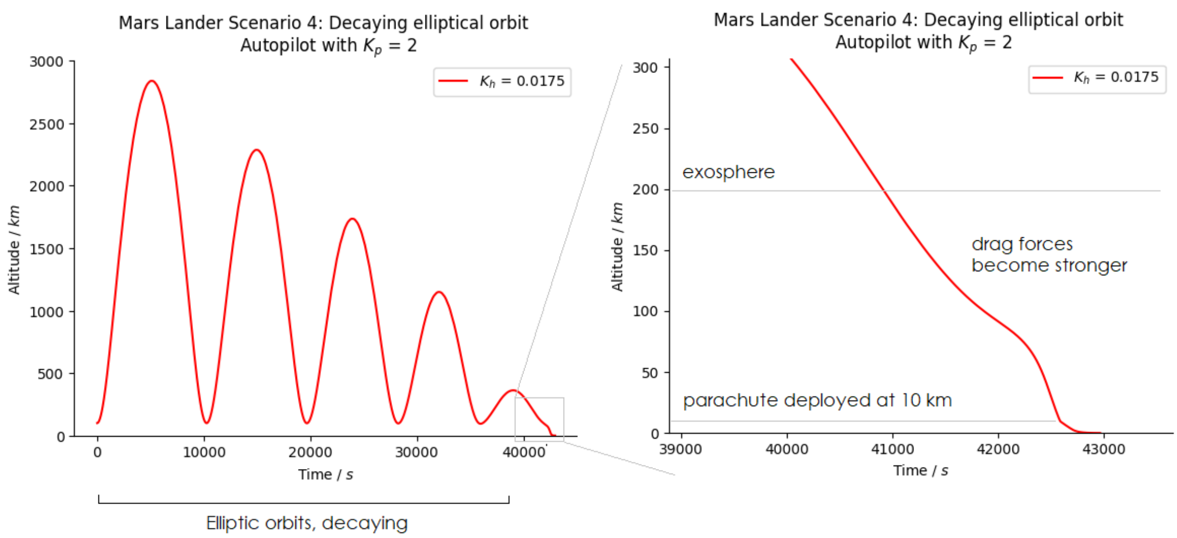
In order to land the craft in Scenario 4 successfully, a simple attitude stabilisation was implemented, and the ability to deploy the parachute from autopilot was added.

```
if (altitude > 3000 and altitude < 10000 and velocity * position < 0
   and safe_to_deploy_parachute) {
      parachute_status = DEPLOYED;
}
```

The attitude stabilisation was changed to point the lander away from its velocity, so that all thrust exerted acts directly against the velocity, giving it maximum momentum change, saving fuel.

```
void attitude_stabilization (void)  // lander.h: line 1576
  // Three-axis stabilization
{
  vector3d up, left, out;
  double m[16];
  // this is the direction we want the lander's nose to point in
  if (velocity * position < 0) {
      up = -1 * velocity.norm();  // travelling downwards (landing):
  }                               // point against velocity
  else {
      up = velocity.norm();       // travelling upwards (taking off):
  }                               // point with velocity
```

This also ensured the autopilot did not break Scenarios 1, 3 and 5 as their descents are purely vertical, and allowed a successful autopilot landing in Scenario 4:



Allowing manual attitude control

The Z and X keys were used to rotate

Adding the gravitational effects the Sun and Mars' moons

Near Mars (below 5,000 km altitude) , the error in assuming the only gravitational force on the lander is due to the planet is less than 0.5%. However, at further orbits, Mars' gravity no longer dominates and the Sun's gravity becomes comparable.

The long-term behaviour of an areostationary orbit (Scenario 6, altitude 17,000 km) is also influenced by the presence of Phobos in the form of orbital resonance. The autopilot for Scenario 6 uses attitude stabilisation to counter the tiny changes to the circular orbit, which is entirely unnoticeable for practical times unless the mass of Phobos is manually overridden to be a larger value.

Gravitational force contributions at representative altitudes:

Altitude	100 km	1,000 km	10,000 km
Mars	99.97%	99.88 %	98.94 %
Sun	0.03 %	0.12 %	1.06 %
Phobos	~ 10^{-9} %	~ 10^{-7} %	0.00076 %
Deimos	~ 10^{-10} %	~ 10^{-8} %	~ 10^{-7} %

This was done by finding the orbital planes and speeds of Deimos and Phobos relative to Mars, and of Mars relative to the Sun. The positions (modelled as elliptical orbits) were recorded at each time step of all their positions relative to Mars at (0, 0, 0). Their gravity terms were then added to the acceleration equation in `numerical_dynamics`.

Improving the integrator algorithm using the Gauss-Jackson method

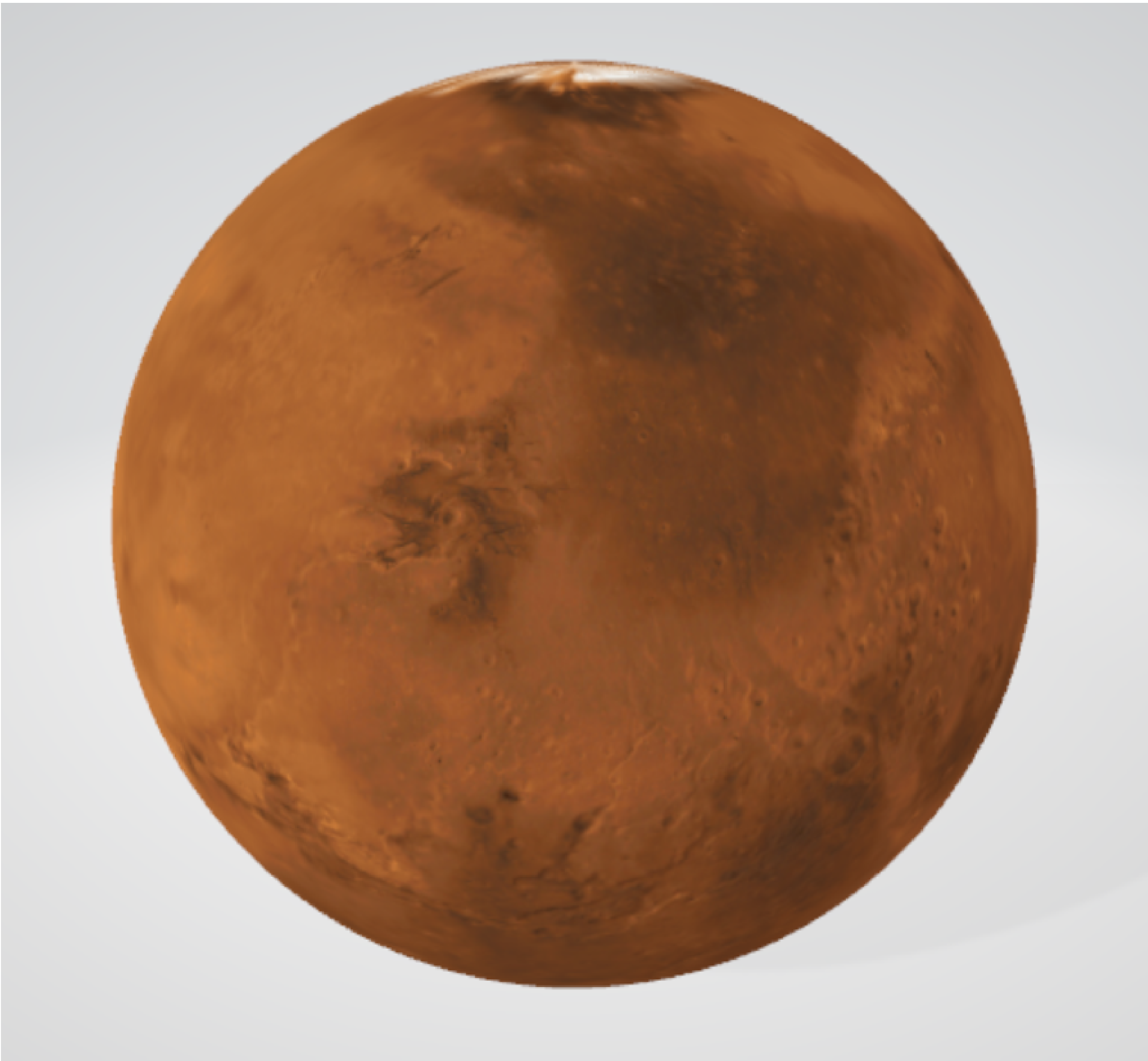
Improving the controller using PID-control

Extensions

Some more challenging additions

Improving the graphics using a 3D model

Inspired by the winning project from 2018 [2] as well as NASA's official simulation [9], I looked at how to replace the plain wireframe graphic of Mars with a realistic 3D model.



With the help of several great resources, I was *almost* able to do it. Many of these tools are fairly recently made - students attempting to do this 5 years ago would have found it practically impossible!

Some of these resources were:

- Tiny glTF Loader by Syoyo Fujita: [10]

  A very convenient tool for loading in 3D models and rendering them in OpenGL directly.

- OpenGL Tutorial by Victor Gordan: [5]

  This was an amazing tutorial on how to create a model object loader from scratch, specifically video #13 in the playlist. I learned a lot here!

- StackOverflow: [11]

  As always, StackOverflow proved to be full of answers, with one question in particular being a solution to a very specific issue I faced.

- GLUT Red Book (Online): [12]

  An ancient-looking website, but the only place I could find a clear description of the various GL functions in lander_graphics.cpp to understand the program flow.

- 3D Models from NASA and the Microsoft open-source community: [6]

  Some of the only free 3D models of Mars and its moons publicly available - the two moons, Deimos and Phobos, provided by NASA, and Mars from the Microsoft 3D Viewer library (which turned out to be even better than NASA's Mars model).

- 3D Viewer: [7]

  I used this online tool to visualise the various models as well as check their sizes in order to debug a camera issue I initially faced.

- glTF Shell Extensions: [8]

  A very convenient tool to extract the JSON, binary and texture files from a package GLB file, which is necessary to load into OpenGL.

I was able to eventually load a 3D Mars model into an empty C++ project, but I was unfortunately not able to combine it with the Mars Lander. It seems that a lot of the trouble came from the fact that lander_graphics.cpp used various old-style GL functions while the glTF loader used new functions.

Despite failing this goal, I still learned a lot about graphics along the way.

References

[1] Stability of Verlet integrator, Stack Exchange, 2013
scicomp.stackexchange.com/questions/10329/stable-time-step-limits-for-velocity-verlet-integration

[2] Mars Lander Project Page, Tom Wyllie, 2018
wyllie.dev/mars/

[3] PID Controller, Hernán Foffani, 2017
github.com/hfoffani/PID-controller

[4] Implementation of Gauss-Jackson Integration for Orbit Propagation, Matthew Berry & Liam Healy, 2004
drum.lib.umd.edu/bitstream/handle/1903/2202/2004-berry-healy-jas.pdf

[5] OpenGL Tutorial, Victor Gordan, 2021
youtube.com/playlist?list=PLPaoO-vpZnumdcb4tZc4x5Q-v7CkrQ6M-

[6] Deimos and Phobos, NASA, 2019:
solarsystem.nasa.gov/resources/2434/deimos-3d-model/
solarsystem.nasa.gov/resources/2358/phobos-3d-model/
Mars, freely available from within the 3D Library of Microsoft 3D Viewer:
microsoft.com/en-us/p/3d-viewer/9nblggh42ths

[7] Online 3D Viewer:
3dviewer.net/

[8] glTF Shell Extensions:
github.com/bghgary/glTF-Shell-Extensions

[9] Mars 2020 Entry Descent Landing, NASA, 2020:
eyes.nasa.gov/apps/mars2020/#/home

[10] Tiny glTF Loader, Syoyo Fujita, 2018:
github.com/syoyo/tinygltf

[11] StackOverflow answer, Rabbid76, 2019
https://stackoverflow.com/a/55324961/8747480

[12] OpenGL Red Book, 1996
https://www.glprogramming.com/red/appendixd.html

[13] Space Flight Handbooks Volume I, p.212, NASA, 1963
https://ntrs.nasa.gov/citations/19630011221

PID help
https://www.csimn.com/CSI_pages/PIDforDummies.html