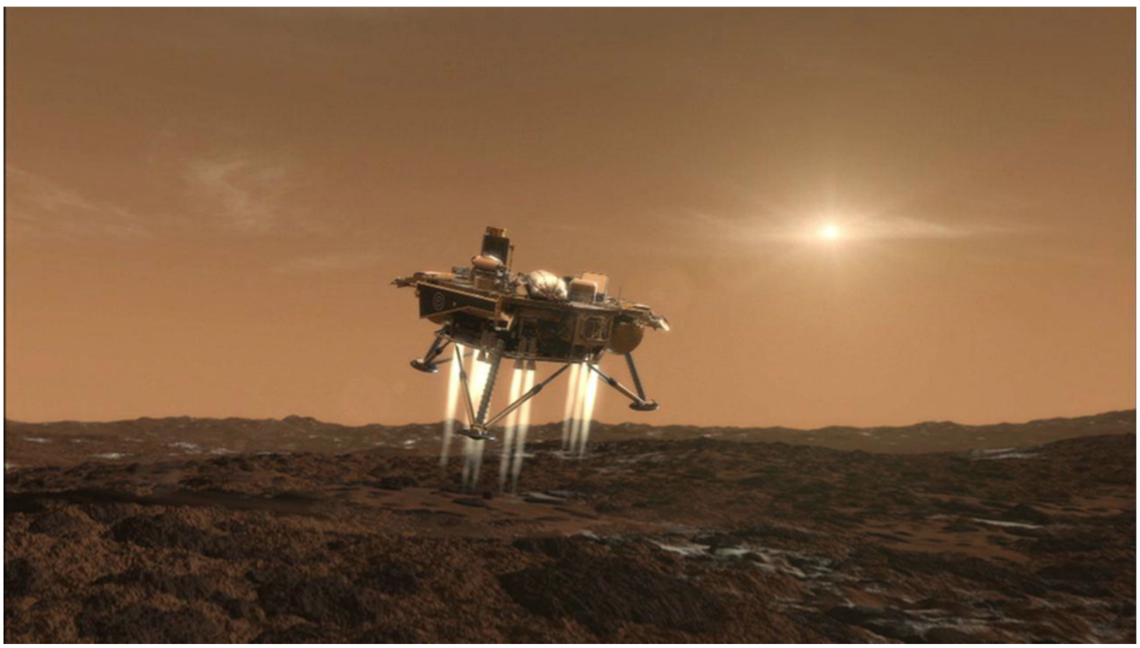
MARS LANDER

Andrew Gee and Gábor Csányi

October 17, 2023



Contents

1 Introduction

The aim of this exercise is to develop a computer program that simulates the landing of a spacecraft on Mars. The following sections break this task down into stages, starting from the motion of a simple harmonic spring (in Python, with which you should be familiar) and ending with the realistic three-dimensional trajectory of the landing craft, complete with thruster and parachute (in C++, which is probably new to you). You must also write a simple function that automatically *controls* the thrust to land the craft safely.

In order to visualize the system, the exercise is accompanied by a 2000–line program that constitutes a complete graphical interface to flight instruments and controls, including two external views of the descending craft itself. The code does *not* contain the mechanical simulation, i.e. how to update the trajectory of the craft given the instantaneous forces acting on it: that is your job. Without your contribution, the code can be compiled and run, but the craft just hangs stationary in space. A detailed description of the code, its key functions and variables is in Appendix B.

Assignments 1–5 are **not optional** and their completion will be assumed in the Part IB Control course. It should take you no more than a few days to complete these assignments.

Aims and objectives

- To appreciate the various aspects of dynamical simulation, from modelling forces, through integrating the equations of motion to visualizing the results.
- To model the principal forces (gravity, engine thrust and drag) on a landing craft as it orbits and then enters the atmosphere of a planet.
- To compare the forward Euler and Verlet techniques for the numerical integration of dynamical equations of motion.
- To introduce basic control theory, by implementing simple strategies for the automatic control of the landing craft.
- To gain experience and confidence developing software in Python and C++ on one's own personal computer.

2 The Moodle course page

The **1CW: Mars Lander** Moodle course page contains links to supporting material, including:

- This handout, and also a brief, one-page overview of the exercise.
- Supplied source code in a single zip file: `spring.py` for Assignments 1–2, `spring.cpp` for Assignment 3, `lander.cpp`, `lander.h`, `lander_graphics.cpp` and `Makefile` for Assignments 4–5.
- "Getting started" instructions for Windows, Linux and Mac OS X.
- A list of common problems, with answers!
- Support forums, where you can ask specific questions to demonstrators and other students.
- Details of the Airbus prizes, and commendations of past winners.
- Some extensions for Parts IB and IIA of the Engineering Tripos.

3 Assignment 1: numerical dynamics in one dimension

This and the next assignment are best done in Python. The first step is to learn to simulate something simple. Consider a mass m at the end of a simple harmonic (massless) spring of strength k in one dimension. The energy of the system is given by the sum of the kinetic and potential energies

$$E = \frac{1}{2}kx^2 + \frac{1}{2}mv^2$$

where x is the displacement and v is the velocity of the mass, which are functions of time. The force on the mass is the negative of the derivative of the potential energy with respect to position, so the equations of motion are

$$\begin{aligned} \dot{v} &= F/m = -\frac{1}{m}\frac{\partial E}{\partial x} = -kx/m \\ \dot{x} &= v \end{aligned}$$

In this first assignment, you have to write a program that integrates these equations from an initial condition (e.g. $x(0) = 0, v(0) = v_0$). The integration will have to be done approximately, using a discrete time step Δt. You will have to adjust the value of Δt empirically. If it is too small the program takes too long to run, and if it is too large the integration becomes numerically unstable due to accumulating inaccuracies, and the dynamical variables will take on unfeasible values (e.g. the amplitude of the oscillations will grow in time, even though no energy is supplied to the system). You should investigate two different ways of discretizing the trajectory. To obtain the first one, let us expand the dynamical variables in Taylor series around a particular point t in time, e.g. for x

$$\begin{aligned} x(t+\Delta t) &= x(t) + \Delta t\frac{\partial x}{\partial t} + \mathcal{O}(\Delta t^2) \\ &= x(t) + \Delta t v(t) + \mathcal{O}(\Delta t^2) \end{aligned}$$

and similarly for v

$$\begin{aligned} v(t+\Delta t) &= v(t) + \Delta t\frac{\partial v}{\partial t} + \mathcal{O}(\Delta t^2) \\ &= v(t) + \Delta t F(t)/m + \mathcal{O}(\Delta t^2) = v(t) - \Delta t k x(t)/m + \mathcal{O}(\Delta t^2) \end{aligned}$$

Using these equations, you can obtain the values of the dynamical variables at time $t + \Delta t$ from their values at time t, and this algorithm is called the Euler method. At each step, the error in your integration will be of the order Δt^2, or alternatively, the trajectory will be accurate to order Δt: this is a "first order" integrator and is rather inaccurate. If we truncate the Taylor expansion beyond the second order term, we get a better Euler method, which is accurate to second order. However, with a little trick, we can do even better without increasing the complexity. Let us again expand the dynamical variables in Taylor series, but now both forwards and backwards in time:

$$\begin{aligned} x(t-\Delta t) &= x(t) - \Delta t\frac{\partial x}{\partial t} + \frac{\Delta t^2}{2!}\frac{\partial^2 x}{\partial t^2} - \frac{\Delta t^3}{3!}\frac{\partial^3 x}{\partial t^3} + \mathcal{O}(\Delta t^4) \\ x(t+\Delta t) &= x(t) + \Delta t\frac{\partial x}{\partial t} + \frac{\Delta t^2}{2!}\frac{\partial^2 x}{\partial t^2} + \frac{\Delta t^3}{3!}\frac{\partial^3 x}{\partial t^3} + \mathcal{O}(\Delta t^4) \end{aligned}$$

Adding the two expansions, we get

$$x(t-\Delta t) + x(t+\Delta t) = 2x(t) + \Delta t^2\frac{\partial^2 x}{\partial t^2} + \mathcal{O}(\Delta t^4)$$

from which an update rule can be extracted:

$$x(t+\Delta t) \approx 2x(t) - x(t-\Delta t) + \Delta t^2 F(t)/m$$

This is the Verlet integrator, which uses *two* previous values of the position to estimate the next one. Notice that it does not make use of the velocity directly. For analysis purposes, an estimate of the velocity can be obtained from the trajectory:

$$v(t) = \frac{1}{2\Delta t}\left[x(t+\Delta t) - x(t-\Delta t)\right] + \mathcal{O}(\Delta t^2)$$

This velocity estimate is one step behind the position estimate. It is possible to estimate $v(t+\Delta t)$, thus synchronizing the two estimates, but only at the expense of accuracy:

$$v(t+\Delta t) = \frac{1}{\Delta t}\left[x(t+\Delta t) - x(t)\right] + \mathcal{O}(\Delta t)$$

Specific tasks

1. Write a program to execute the Euler and Verlet algorithms for the mass on the spring, and plot the trajectories of the dynamical variables as functions of time. Convince yourself that the programs work by solving the equation of motion analytically. To get you started, a Python script for the Euler method is provided on the Moodle course page.

2. Investigate the numerical solution for various different values of the time step Δt. Compare the performance of the two algorithms over several thousand oscillations, both with each other and also with the exact analytical solution. For example, if you take $m = k = v_0 = 1$ and calculate the trajectory from $t = 0$ to $t = 1000$, you should find that the Verlet integrator is stable for $\Delta t = 1$ but not for $\Delta t = 2$.

3. Find, by trial and error, the critical value of Δt for the Verlet case, above which it becomes unstable. The first order Euler integrator is never completely stable, but gets better for smaller values of Δt.

4 Assignment 2: numerical dynamics in three dimensions

Now consider the dynamics of a body in the gravitational field of a planet. The equations of motion have to change, but the integrators remain the same. The gravitational force is given by

$$\mathbf{F}_G = -\frac{GMm}{|\mathbf{r}|^2}\hat{\mathbf{r}}$$

where G is the universal gravitational constant, M is the mass of the planet (6.42×10^{23} kg for Mars), m is the mass of the moving body and $\mathbf{r}$ is its position relative to the centre of the planet.

The most important change is that you now have to work with vectors in three-dimensional space instead of the one dimension of the simple harmonic oscillator. There are six dynamical variables (the x, y and z positional coordinates of the body, and the corresponding velocity components) and you have to use vector arithmetic inside the integrators. Arrange the variables in three-element NumPy arrays called `position` and `velocity`. Do not use six separate, scalar variables: this is clumsy and does not generalise well for Assignment 4.

Specific tasks

After modifying your Euler and Verlet programs as described above, check them by simulating:

1. Straight down descent
2. Circular orbit **Gravity = Centreptal**
3. Elliptical orbit **All Just before escape is elliptical**
4. Hyperbolic escape **All Just after escape speed**

For simplicity, take the centre of the planet to be the origin of the coordinate system, and the direction of the initial velocity to be perpendicular to the initial position vector of the body for Scenarios 2–4. Use zero initial velocity for Scenario 1. You will obtain different orbits by adjusting the magnitude of the initial velocity. It might help if you first calculate the required speed for a circular orbit, and the escape velocity (use the principle of conservation of energy). For Scenario 1, plot altitude as a function of time. For Scenarios 2–4, plot the trajectory in the orbital plane.

5 Assignment 3: familiarisation with C++

Read through Appendix A for a quick introduction to C++. This includes a worked C++ example of an Euler simulation of a mass on a spring, which you can compare and contrast with the Python equivalent in Assignment 1. Measure the execution time of the Python and C++ simulations, both with and without C++ compiler optimizations. Modify the provided C++ code to perform Verlet instead of Euler integration.

6 Assignment 4: landing craft simulator

Drawing on your solution to Assignment 2, edit the supplied C++ `lander.cpp` program to update the lander's position and velocity using both Euler and Verlet integrators. Note that the supplied code includes a `vector3d` class to store and manipulate 3-vectors, and that you do not have to code the main loop over time: see Appendix B. In addition to gravity, you now have to consider extra forces due to the engine's thrust and any atmospheric drag. The code supplies you with a function to work out the thrust vector from the current orientation of the lander, as well as an attitude stabilizer function that maintains the orientation with respect to the planet vertical. The aerodynamic drag is given approximately by

$$\mathbf{F}_d = -\frac{1}{2}\rho C_d A|\mathbf{v}|^2\hat{\mathbf{v}}$$

where ρ is the atmospheric density, C_d is the drag coefficient, A is the projected frontal area and $\mathbf{v}$ is the lander's velocity. If the parachute is deployed, you need to sum the two separate drag expressions for the lander and the parachute. A correct Verlet-based simulator should:

- Crash the lander after 83.5 s with a final descent rate of 176.1 ms^{-1} in Scenario 1.
- Crash the lander after 42629.6 s with a final descent rate of 172.0 ms^{-1} in Scenario 4.
- Crash the lander after 361.7 s with a final descent rate of 328.7 ms^{-1} in Scenario 5.

There might be some small discrepancies, depending on how you chose to deal with the first iteration of the Verlet integrator.

Once your program is working and you get plausible trajectories given the supplied initial conditions (try all of them by pressing the corresponding number keys in the simulator, and press the 'h' key to see all the keyboard and mouse functions), use the simulator to land the craft on the surface without breaking it. Start with Scenario 1, straight descent from 10km: it is not easy! To bring the craft down from orbit, you first need to initiate the re-entry procedure by decelerating. The easiest way to do this is to disengage the attitude stabilizer, wait until the engine is pointing backwards with respect to the direction of motion (press the 'h' key to see when this is), then fire the thruster. Once inside the atmosphere, you can deploy the parachute to increase drag, but make sure you do not do this while travelling too fast (the fabric will vaporize) or when the drag load is too high (the tethers will snap). The landing is considered successful if neither the descent rate nor the ground speed exceed 1 m/s.

Finally, compare the performance of the Euler and Verlet integrators. Scenario 4 is particularly illuminating: you should see a big difference in the way the trajectories evolve with time. Make sure you understand why this is.

Better options: RK4.
More in IB mechanics
AND 3A3 (shockwaves CFD)
And 4A2 (CFD!!)

7 Assignment 5: autopilot

Now that you have practised landing the craft manually, think about the strategy you used to control its rate of descent. How did you decide how much thrust to use, and when? Why did you deploy the parachute when you did? We would like to encapsulate a suitable strategy in an autopilot, whose job is to adjust automatically the lander's controls (thrust, parachute, attitude stabilizer) to bring it safely to the surface. There is a powerful engineering discipline, **control theory**, that provides us with the necessary tools to do this. Control theory is introduced next year in Part IB Paper 6, so for now we will take a less-than-totally-rigorous look at a straightforward technique called **proportional control**.

To keep things simple, we will consider only Scenarios 1 and 5 (simple descent), forget about the parachute, leave the attitude stabilizer engaged (so the engine is always pointing towards the surface) and use just the engine's thrust to control the rate of descent. Suppose our strategy is that the descent rate should decrease linearly as the lander approaches the surface,

$$\mathbf{v}.\mathbf{e_r} = -(0.5 + K_h h)$$

where $\mathbf{e_r}$ is the unit vector in the radial (vertical) direction, h is the lander's altitude and K_h is a positive constant. This seems reasonable, since the descent rate should then approach 0.5 m/s as the lander touches down, safely within the 1 m/s limit. Define an error term

$$e = -(0.5 + K_h h + \mathbf{v}.\mathbf{e_r})$$

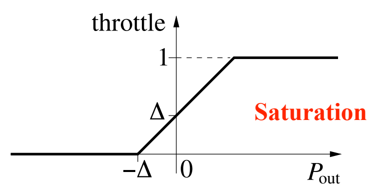
which is positive if the lander is descending too quickly and negative if the lander is descending too slowly. The instantaneous output of a proportional controller is then

$$P_{\text{out}} = K_p e$$

where K_p is a positive constant known as the **controller gain**. Ideally, we would use P_{out} to control directly the engine's throttle, but here we must diverge a little from perfect proportional control theory, since whereas P_{out} might take any real value representing arbitrary forward and reverse thrust, the lander's engine is only capable of delivering a limited amount of forward thrust corresponding to throttle values in the range 0 to 1. There is also the lander's weight to consider: even when the error e approaches zero, we still need a certain amount of thrust to balance the weight. This suggests the following autopilot:

feedforward action..

$$\text{throttle} = \begin{cases} 0 & P_{\text{out}} \le -\Delta \\ \Delta + P_{\text{out}} & -\Delta < P_{\text{out}} < 1 - \Delta \\ 1 & P_{\text{out}} \ge 1 - \Delta \end{cases}$$



Implement this scheme in the autopilot function and experiment with different values of K_h, K_p and Δ. In Part IB Paper 6, you will see how to set the controller gains analytically, but for now you will need to resort to trial and error when tuning K_h and K_p. Initially, you might want to give yourself an effectively infinite fuel supply (see Appendix B), though you should eventually manage to find suitable values for K_h that bring the lander safely down with the supplied amount of fuel. Make sure you understand the effect of K_h: too high and the lander crashes, too low and it runs out of fuel, but why? Adapt the program to output values of h and $\mathbf{v}.\mathbf{e_r}$ at each time step (use the `ofstream` method introduced in Appendix A). Read these values into Python, and use `matplotlib` to plot graphs of actual and target descent rate against altitude for various values of K_h.

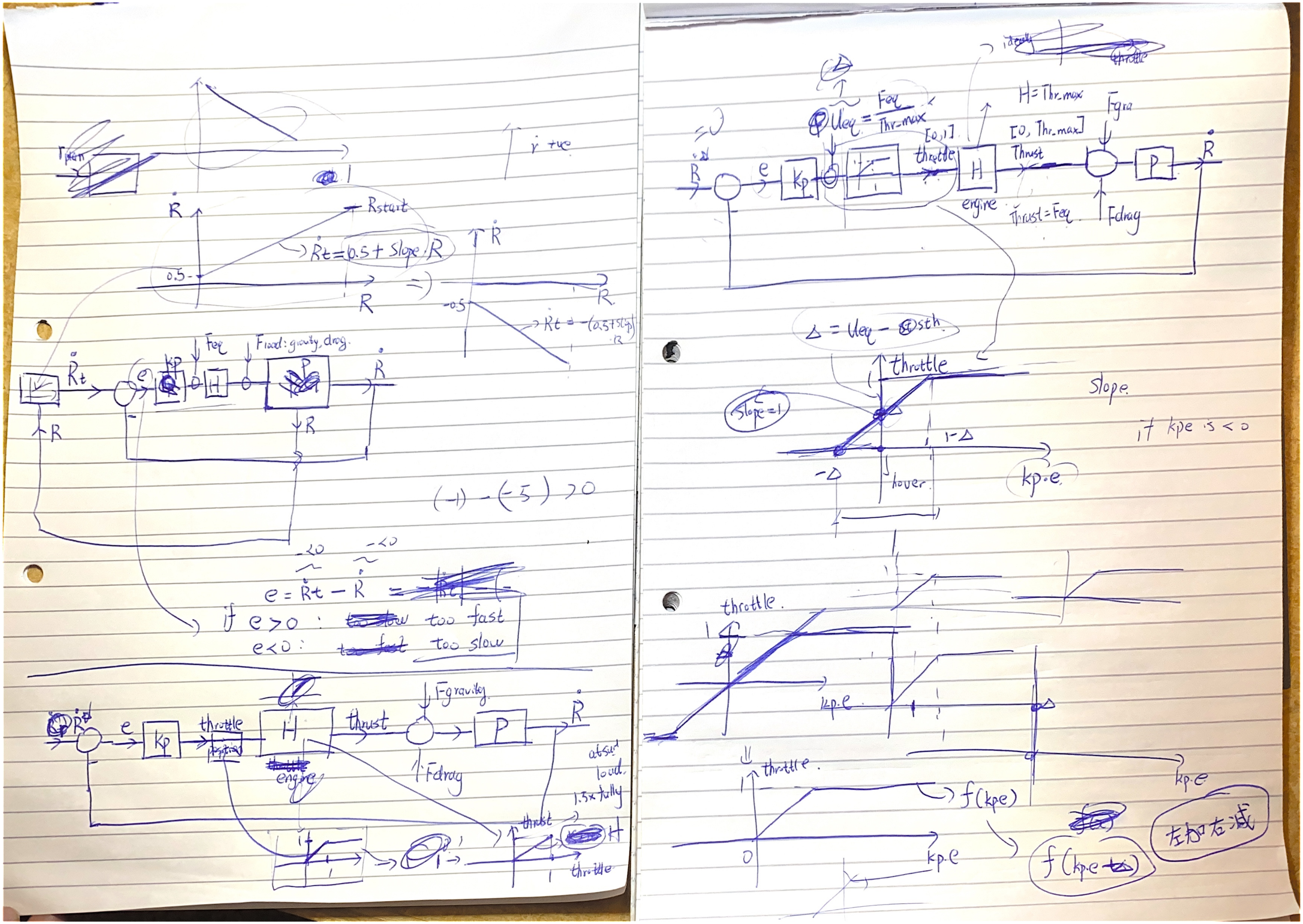
Too high: saturated. can't deliver enough thrust.
Too low: too much fuel burning out.

8 Extension assignments

Here are some suggestions for optional extensions, in approximately increasing order of difficulty. Allow an hour or two for the early assignments. The later assignments are more open-ended, require significant background reading and could easily take weeks if not months!

1. Set up Scenario 6 to model an areostationary orbit.
2. The simple autopilot developed earlier doesn't use the parachute and hence won't arrest the ground speed sufficiently to land safely after orbital re-entry. Modify the autopilot so that it works in all scenarios, after manual firing of the engine to initiate the re-entry sequence.
3. Attitude control: introduce a new key to rotate the craft in the plane of the orbit.
4. Extend the autopilot to cover orbital re-entry as well as the final descent stage.
5. Use any-angle attitude control to inject the craft into orbit (start in Scenario 1, 3 or 5, and give yourself an infinite fuel supply). Extend the autopilot to cover orbital injection.

Interpretation of the autopoilt in the handout



2P6: lag is approximated by a first order lag.
Delay in laplace transform is exp(-tau*s).
stability via Nyquist diagram.

6. The engine lag and delay (see Appendix B) are currently zero. Make them nonzero and try to land the craft, first manually and then with the autopilot.
7. The simulator's graphics take some account of the planet's rotation, but the mechanics currently ignore this. Correct this deficiency.
8. Extend the simulator to model steady atmospheric wind flow (i) without and (ii) with additional random gusts.
9. How might you tune your autopilot to achieve (i) minimal fuel usage, (ii) minimal descent time, (iii) minimal peak acceleration? **3F2. minimal energy.**
10. Improve the graphics, which are currently written in legacy, fixed-function-pipeline OpenGL. You might wish to rewrite the graphics using modern, shader-based OpenGL or Vulkan. Alternatively, you could try a higher-level game engine like Unity or Unreal.
11. The planet's surface is currently assumed to be perfectly spherical. Generate some realistic surface terrain and then adapt the simulator to account for the terrain in (i) the graphical rendering functions and (ii) the mechanics functions. For safe landing, you will need to equip your lander with a laser altimeter.

Engineering Tripos Part IIB
Module 4F3 — An Optimisation Based Approach to Control
Part 1: Optimal Control

F. Forni (f.forni@eng.cam.ac.uk)

Based on previous courses by E.N. Hartley (2014), K. Glover (2013) and G. Vinnicombe (2004)

Lent Term 2022

Contents

Solving convex optimisation problems

- No analytical solution (but global minimum)
- Reliable and efficient algorithms
- Computation time (roughly) proportional to $\max\{n^3, n^2 m, F\}$ where F is the cost of evaluating f_i's and their first and second derivatives

Using convex optimisation

research area. maybe transform coord sys then problem becomes conv

- Often difficult to recognise
- Many tricks for transforming problems into convex form
- Surprisingly many problems can be solved via convex optimisation

2.7 Quadratic programming (QP)

A special case of a convex optimisation problem is the so-called quadratic programme (QP): (assume $P = P^T \geq 0$)

$$\begin{array}{ll} \text{minimise} & (1/2)x^T P x + q^T x + r \\ \text{subject to} & Gx \leq h \\ & Ax = b \end{array}$$

- Efficient methods and solvers exist
- S. Boyd and Li. Vandenberghe, *Convex Optimization*, Cambridge University Press, 2004 `https://stanford.edu/~boyd/cvxbook/`
- CVX: Matlab Software for Disciplined Convex Programming: `http://cvxr.com/cvx/`
- Python-embedded modeling language for convex optimization problems: `https://www.cvxpy.org`
- OSQP

3 Introduction to Optimization in Control

3.1 Optimization in control

System $x_{k+1} = f(x_k, u_k)$, $y = h(x_k, u_k)$ (input u, output y, state x).

Drive the system optimally

- Find the optimal sequence u_k^* which minimizes... (energy, time, ...)
- Find the optimal trajectory x_k^*... (shortest path, ...)

Optimal feedback control

- Find state-feedback controllers $u = \Sigma(x)$ (complete information) or output-feedback controllers $u = \Sigma(y)$ (incomplete information) which guarantee optimal closed loop performance/robustness (LQR, $\mathcal{H}_2$, $\mathcal{H}_\infty$).

Optimization assisted control

- Predictive control (handling constraints, flexible objectives, ...)
- Controllers that learn / adapt (policy iteration, deep neural nets, ...)
- Computer-assisted control design

3.2 Example: Goddard's rocket problem

How to send a rocket as high up in the air as possible?

Model

$$M\ddot{h} = -Mg - D(\dot{h}, h) + u \qquad \dot{M} = ku$$

$\dot{m} < 0$
$u > 0$
$k < 0$

rocket + fuel mass M, height h, gravity g, air resistance D, and $k > 0$.

Optimal problem

Find $0 \leq u(t) \leq u_{\max}$ which at time T maximizes $h(T)$.

Solution

∵ fuel is heavy, get rid of fuel as early as possible

- $D = 0$: $u(t) = u_{\max}$, burn fuel fast, because it costs to lift it.
- $D \neq 0$: hard - it can be optimal to move at low speed when air resistance is high.

3.3 Example: minimum connection time

How to minimize connection time between stations?

Model $M\ddot{x} = u$

train + people mass M, position x, acceleration u.
force.

Optimal problem

Find control input $-1 \le u(t) \le 1$ which guarantees $x(0) = A$, $\dot{x}(0) = 0$, $x(T) = B$, $\dot{x}(T) = 0$ in minimum time T.

Solution

Maximal acceleration/deceleration (bang bang control)

$$u(t) = 1 \text{ for } 0 \le t < t^*_{A,B} \qquad u(t) = -1 \text{ for } t^*(A,B) \le t \le T_{A,B}.$$

3.4 Example: shortest path

How to steer optimally a mobile robot?

Model

$$\dot{x} = v\cos\theta \qquad \dot{y} = v\sin\theta \qquad \dot{\theta} = \omega$$

where (x, y) is the robot position and θ its orientation.

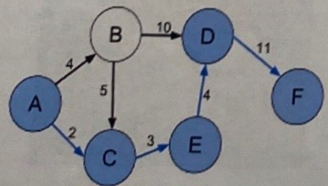
Optimal problem

Find the optimal controls $0 \le v(t) \le v_{\max}$ and $-\omega_{\max} \le \omega(t) \le \omega_{\max}$ that minimize the path length

$$\min \int_0^T \sqrt{\dot{x}^2 + \dot{y}^2}\,dt = \min \int_0^T |v|\,dt$$

to reach a goal position $x(T) = x^*$

3.5 Example: shortest path on graphs



Find the path between two nodes which minimizes the sum of the weights along the path.

Model

$$x_{k+1} = u_k \qquad u_k \in \text{neighborhoods}(x_k)$$

where x_k is the current node and x_{k+1} is the next node (assigned by the control input u_k). The transition from x_k to $x_{k+1} = u_k$ costs $w_{x_k,u_k} \ge 0$.

Optimal problem

Find $u_0^*, \ldots, u_{T-1}$ (and trajectory $x_0^* = A, x_1^*, \ldots, x_{T-1}, x_T = F$) which minimizes

$$\sum_{k=0}^{T-1} w_{x_k,u_k} \rightarrow 2021\ 4M12\ Q1$$

3.6 Example: optimal feedback controller

Find feedback controllers which optimizes a given index.

Model

$$x_{k+1} = Ax_k + Bu_k$$

where x_k is the current state and x_{k+1} is the next state, u_k is the control input, and A and B are matrices of suitable dimension.

Optimal problem

it is There is an dist. that state has high cost so will avoid that state

Find the state-feedback controller $u = Kx$ which minimizes

$$\sum_{k=0}^{T-1} \left[\underbrace{x_k^\top Q x_k}_{\text{distance from } x_{ref} = 0} + \underbrace{u_k^\top R u_k}_{\text{control effort}} \right] + \underbrace{x_T^\top S x_T}_{\text{final distance from } x_{ref} = 0}$$

where $Q \ge 0$, $R \ge 0$, and $S \ge 0$ (finite-horizon linear quadratic regulator).

e.g for rocket R is large
Q is small

Show (using the standard formula for the total derivative of a function of several variables) that

$$\frac{dV}{dt} = \frac{\partial V}{\partial x}\frac{dx}{dt} + \frac{\partial V}{\partial t}$$

Hence deduce that

$$\lim_{h \to 0} \frac{1}{h}(V(x(t+h), t+h) - V(x(t), t)) = \frac{\partial V}{\partial x}\frac{dx}{dt} + \frac{\partial V}{\partial t}$$

(b) Let $V = x^T X x$, where $X^T = X$. Show (by expanding) that

$$\frac{\partial V}{\partial x} = 2x^T X$$ → check cribs ?

(c) Consider the finite horizon, continuous time LQR problem (as defined in the lecture notes) with the associated HJB equation

$$-\frac{\partial V}{\partial t} = \min_u \left\{ x^T Q x + u^T R u + \frac{\partial V}{\partial x}\underbrace{(Ax + Bu)}_{\frac{\partial x}{\partial t}} \right\}$$

$$V(x, T) = x^T X_T x$$

where $R > 0$ and assume $X_T = X_T^T$.

Show that, at $t = T$, the right hand side of the first expression above can be evaluated to give

$$-\left.\frac{\partial V}{\partial t}\right|_{t=T} = x^T(Q + X_T A + A^T X_T - X_T B R^{-1} B^T X_T)x$$

(Hint: use the result of (b), and note that $x^T X(Ax + Bu)$ is a scalar).

Hence, deduce that the solution to the HJB equation is of the form

$$V = x^T X(t) x$$

where $X(t)$ solves an appropriate Riccati differential equation.

3. Consider the problem of bringing a (unit) mass to rest on a given point in minimum time, using a force of bounded magnitude (also assumed to be equal to 1). This can be written as an optimal control problem, with state equations

$$\dot{x}_1 = x_2$$
$$\dot{x}_2 = u$$

with $|u(t)| \le 1$ for all t, $x(T) = 0$ and T is to be minimised.

Ask why stage cost = 0

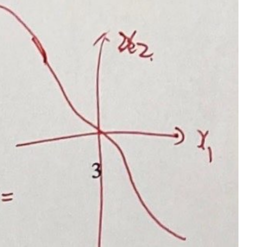
Let $V(x)$ be the minimum time to reach the origin from a state x. Consider the transition from x to $x + \delta x$ in time δt and taking limits show that V should satisfy the H-J-B equation:

$$\min_{u:|u|\le 1}\left(\frac{\partial V}{\partial x_1}x_2 + \frac{\partial V}{\partial x_2}u + 1\right) = 0.$$

(c) but this complicate the sol

Cab't $t^2 c(t)$ can be

stage cost for min tim $= \int_0^T 1\,dt$ ∴ stage cost = 1



if let > ⇒ =

Show that the optimal input u is only ever equal to ± 1, and find an expression for the optimal u in terms of V.

Show that a solution to the HJB equation is,

$$V(x) = \begin{cases} x_2 + \sqrt{2x_2^2 + 4x_1} & \text{for } 4x_1 > -2x_2|x_2| \\ -x_2 + \sqrt{2x_2^2 - 4x_1} & \text{for } 4x_1 < -2x_2|x_2| \end{cases}$$

and sketch the resulting state-plane trajectories.

4. (a) Consider the following finite horizon LQR problem:

$$\text{minimize} \int_0^T (\alpha^2 x(t)^2 + u(t)^2)\,dt + x(T)^2$$

for the single input, single state system,

$$\dot{x} = x + u, \quad x(0) \neq 0$$

Show that the value function for this problem is given by

$$V(x, t) = x^2 X(t) \text{ where } X(t) = 1 - \sqrt{1+\alpha^2}\tanh\left(\sqrt{1+\alpha^2}(t - T)\right).$$

What are the optimal cost and the optimal control?

If $V = (x(t), t) = 5x + 3t$, $\frac{\partial V}{\partial t} = 3$, $\frac{\partial V}{\partial x} = 5$

(b) Consider now the infinite horizon problem:

$$\text{minimize} \int_0^\infty (\alpha^2 x(t)^2 + u(t)^2)\,dt.$$

Take the value function

$$V(t) = Xx(t)^2$$ ⇒ ~~Ask: why no x(t)? when take~~ $\dot{V}$ → Ask PB of HD, if $V = x^2 X(t)$, $\frac{\partial V}{\partial t} = x^2 \dot{X}(t)$ → But why here $\dot{V} = 2x\dot{x}X$ why assume X is indep of t

and show that

$$\dot{V}(t) + \alpha^2 x(t)^2 + u(t)^2 = (u + Xx)^2 + (2X + \alpha^2 - X^2)x^2.$$

Deduce that the optimal control is $u = -Xx$ where X is the stabilizing solution to $2X + \alpha^2 - X^2 = 0$. What is the optimal cost?

(c) Find $\lim_{t\to-\infty} X(t)$, (where $X(t)$ is from your answer to part (a)) and verify that this corresponds to the stabilizing solution to the algebraic Riccati equation that you found in part (b).

5. Take two matrices X and Y such that XY is square. Show that $\mathrm{trace}(XY) = \mathrm{trace}(YX)$. Hence prove that

$$\mathrm{trace}(CPC^*) = \mathrm{trace}(B^* Q B)$$

Ask in exam to prove this kind of matrix eq

Minimum energy transfer

3F2

Theorem 1.1 *The input,* $\underline{u}(t) = \underline{u}_o(t) = B^T e^{A^T(t_1-t)} W_c(t_1)^{-1}\underline{x}_1$*, takes the state from* $\underline{x}(0) = \underline{0}$ *to* $\underline{x}(t_1) = \underline{x}_1$ *with minimum energy.*

Proof:

Let $\underline{u}(t) = \underline{u}_o(t) + \underline{u}_1(t)$ then $\underline{x}(t_1) = \underline{x}_1 + \int_0^{t_1} e^{A(t_1-t)} B\underline{u}_1(t)\, dt$ and hence $\int_0^{t_1} e^{A(t_1-t)} B\underline{u}_1(t)\, dt = \underline{0}$

Energy in $\underline{u}(t)$ for $0 < t < t_1$ is defined as:

$$\begin{aligned}\int_0^{t_1} \|\underline{u}(t)\|^2 dt &= \int_0^{t_1} \underline{u}(t)^T \underline{u}(t)\, dt \\ &= \int_0^{t_1} \left(\underline{u}_o(t)^T \underline{u}_o(t) + \underline{u}_o(t)^T \underline{u}_1(t) + \underline{u}_1(t)^T \underline{u}_o(t) + \underline{u}_1(t)^T \underline{u}_1(t)\right)\, dt\end{aligned}$$

Now

$$\int_0^{t_1} \underline{u}_o(t)^T \underline{u}_1(t)\, dt = \int_0^{t_1} \underline{x}_1^T W_c(t_1)^{-1} e^{A(t_1-t)} B\underline{u}_1(t)\, dt = 0 = \int_0^{t_1} \underline{u}_1(t)^T \underline{u}_o(t)\, dt$$

Hence

$$\int_0^{t_1} \underline{u}(t)^T \underline{u}(t)\, dt = \underline{x}_1^T W_c(t_1)^{-1} \underline{x}_1 + \int_0^{t_1} \underline{u}_1(t)^T \underline{u}_1(t) dt$$

Since both terms are ≥ 0 the minimum energy is achieved when $\underline{u}_1(t) = \underline{0}$ and hence $\underline{u}(t) = \underline{u}_o(t)$ when

$$\min \int_0^{t_1} \underline{u}(t)^T \underline{u}(t)\, dt = \underline{x}_1^T W_c(t_1)^{-1} \underline{x}_1.$$

A A brief introduction to C++

Euler simulation of a mass on a spring

```
#include <iostream>
#include <fstream>
#include <vector>

using namespace std;

int main() {

  // declare variables
  double m, k, x, v, t_max, dt, t, a;
  vector<double> t_list, x_list, v_list;

  // mass, spring constant, initial position and velocity
  m = 1;
  k = 1;
  x = 0;
  v = 1;

  // simulation time and timestep
  t_max = 100;
  dt = 0.1;

  // Euler integration
  for (t = 0; t <= t_max; t = t + dt) {

    // append current state to trajectories
    t_list.push_back(t);
    x_list.push_back(x);
    v_list.push_back(v);

    // calculate new position and velocity
    a = -k * x / m;
    x = x + dt * v;
    v = v + dt * a;

  }

  // Write the trajectories to file
  ofstream fout;
  fout.open("trajectories.txt");
  if (fout) { // file opened successfully
    for (int i = 0; i < t_list.size(); i = i + 1) {
      fout << t_list[i] << ’ ’ << x_list[i] << ’ ’ << v_list[i] << endl;
    }
  } else { // file did not open successfully
    cout << "Could not open trajectory file for writing" << endl;
  }
}
```

Readers of this document are most likely coming to C++ for the first time, having programmed a little in Python. For this reason, it is instructive to look at the program `spring.cpp` above, which is the C++ equivalent of `spring.py` for simulating the numerical dynamics of a mass on a spring using the Euler method. In so doing, we will identify many similarities between Python and C++, but also some important differences. Both programs are included in a zip file on the Moodle course page.

Lines 1–3

`#include` is used in much the same way as `import` in Python. So these lines indicate that we are going to be using some facilities from C++'s *Standard Library*. Specifically, we need `iostream` for input and output to the user's console, `fstream` for input and output to file, and `vector` for storing and manipulating lists of items.

Line 5

This line is a little controversial. Without it, any terms that we use from the Standard Library would need to be prefixed by `std::`, to make it completely unambiguous what we are referring to. For example, we would need to write `std::vector` and `std::cout` instead of just `vector` and `cout`. Line 5 indicates that we are, somewhat lazily, assuming the standard namespace. It saves quite a bit of typing, but does raise the possibility of ambiguity if, for example, we include some other library that uses `vector` to refer to something else. But any sensible library should avoid duplicating Standard Library names, so this shorthand is reasonably safe, albeit not particularly good practice.

Line 7

Here is a significant difference from Python. Line 7 indicates that the main program starts here: it finishes with the closing curly bracket at line 48. In Python, you will be familiar with the `def` command for defining functions. In C++, the main program itself is a special function, called `main`, which is called when the program is run. Line 7 is the function header, equivalent to Python's `def`, except in this instance we are defining the start of the main program, not a sub-function of the main program.

Line 9

This is a comment. In C++, comments are started either by `//` and terminated by the end of the line, or started by `/*` and terminated by `*/`. The latter form is useful when writing longer comments, or when you want to "comment out" a block of code.

Lines 10–11

These lines illustrate another important difference between Python and C++. In Python, you can introduce new variables as and when you need them. For example, the Python statement `dt = 0.1` creates a new variable called `dt` and immediately assigns the real number `0.1` to it. In C++, we need to explicitly declare every variable, *and the type of data we intend to store in it*, before we use the variable. Hence, in line 10, we declare several variables of type `double`, since we will be storing double-precision real numbers in them. Double precision is a good idea if you are concerned about the accuracy of your computations: compared with single-precision `float` variables, they offer more decimal places at the expense of more memory consumption. In addition to `double` and `float`, other basic C++ data types are `int` for integers, `bool` for Boolean (true or false) values and `char` for characters. In line 11, we declare three variables which are vectors (i.e. lists) of double-precision real numbers. Both of these lines ends in a semicolon, which C++ uses to terminate statements. In contrast, Python statements are terminated by the end of the line. This means that in C++ you can have more than one statement on each line if you like.

Lines 13–21

These lines are used to initialise the variables. They are very much like the Python equivalents, except for the semicolons at the end of each statement.

Line 24

This is the start of the main loop for the Euler integration. C++'s `for` syntax is different to Python's. After the `for` statement, there are three semicolon-separated clauses inside the parentheses. The first is the initialization condition: in this case, we want to initialise the loop by setting time `t` to zero. The second is the continuation condition: we want to continue looping while time `t` is less than or equal to `t_max`. And the third is the increment statement, which is executed at the end of each loop cycle: in this case, we want to increment `t` by `dt`. The main body of the loop (i.e. the set of statements we wish to execute each time round the loop) is enclosed in curly brackets. This is different to Python, where the body of the loop is indented. In C++, whitespace has no semantic significance, unlike in Python, where it does. It is nevertheless good practice to indent C++ code sensibly, so as to make it more legible.

Lines 26–29

These lines append the current values of time `t`, position `x` and velocity `v` to the ends of the trajectory lists `t_list`, `x_list` and `v_list`. `push_back` is used with `vector` types to append items to the end of the list.

Lines 31–34

These lines calculate the acceleration and perform the Euler update step. Apart from the terminating semicolons, they are identical to the corresponding lines of Python code.

Line 36

This curly bracket terminates the body of the `for` loop which started on line 24.

Line 38

At this point, the C++ program performs different tasks to the Python script. Whereas the latter plotted the trajectories on graphs, the former writes the trajectory data to a file. This is because there is no friendly C++ equivalent of `matplotlib` for producing graphs. So instead, we write the data to a file, which we can then read in to a Python program and plot using `matplotlib`.

Line 39

The file is referred to using the variable `fout` which is of type `ofstream` (output file stream). Variables do not all need to be declared at the top of the program: you can declare them lower down the program, as and when they are needed. However, you must declare variables before you use them, so `fout` can only be used after line 39.

Lines 40–41, 45–47

These lines attempt to create a writable file called `trajectories.txt` in the current folder. There is always a chance that the file creation fails, for example if you do not have permission to write to the folder. That is why we include line 41, to check that all is well. If not, line 46 is executed. This prints an error message on the screen. `cout` refers to standard console output, `<<` indicates that we are sending what follows to `cout`, and `endl` means "end line" (i.e. move the cursor down to the next line). Note the syntax of the C++ `if`...`else` command, with the curly brackets to delineate the conditional code blocks.

Line 42

This is a `for` loop that runs through all the elements in the trajectory lists. Note the use of `t.size()` to determine how long the lists are: we could just as well have used `x.size()` or `v.size()`, since all three lists are the same length. Note how the counting variable `i` is declared at the same time as it is initialized, in the `for` loop initialization clause. A variable declared like this can only be used within the `for` loop.

Line 43

This line writes the i'th element of each trajectory to the output file, with space characters (`' '`) separating the three numbers. Note the similarity between console output using `cout` and file output using `fout`. The only difference is that, whereas `cout` always refers to the standard console output, we need to declare and define what `fout` refers to in lines 39–40.

Compiled vs. interpreted languages

Perhaps the most significant difference between Python and C++ is the way in which the code is actually run on the computer. The processor inside the computer is designed to fetch, decode and execute a set of instructions written in a very restricted low-level language called *machine code*. Each processor family has its own machine code so, for example, an Intel processor understands different machine code to an ARM processor. At some stage, then, the high-level Python and C++ code needs to be translated into processor-specific machine code before it can be run.

Python is an example of an *interpreted* language, where the code is (more or less) translated on-the-fly as the program is run. In contrast, C++ is an example of a *compiled* language, where the code must first be translated, in a separate process called *compilation*, to produce an *executable* (machine code) file which is then run. So, before you can run the `spring.cpp` C++ program, you need to install a suitable compiler on your PC and compile the C++ source code. Compilation instructions for Windows (Visual C++), Linux and Mac OS X can be found in the "Getting started" section of the Moodle course page.

Interpreted and compiled languages each have their advantages and disadvantages. Broadly speaking, interpreted languages are easier to learn and use, easier to deploy on different computers, and allow complex tasks to be performed in relatively few steps. They are excellent for rapid prototyping and general purpose computing. Where they are somewhat limited is in their ability to interact, at a low level, with the computer hardware. This is why operating systems and device drivers (software which allows computers to interface with peripheral hardware) are typically written in compiled languages, usually some variant of C/C++. Most significantly, the process of translating interpreted languages on-the-fly takes time, slowing down the execution of the program. Compiled languages are therefore preferred for performance-critical applications, especially when running parallel threads on multicore processors, or when interactive graphics is involved.

To appreciate these issues, compile and execute the `spring.cpp` program on your computer. Note that two distinct processes are involved here: first you must compile the code, then run (execute) it. The "Getting started" instructions on Moodle describe the basics of how to do this under Windows, Linux and Mac OS X, but you should go further and discover some of the intricacies of your compiler. In particular, explore the options your compiler offers for *optimization*. Typically, your compiler will offer a standard, quick compilation process, but also different levels of optimized compilation, which take longer but produce machine code that runs faster.

As an example, here are some timings for `spring.cpp` and `spring.py` running on a desktop PC with an Intel Core i7-2600 processor. The operating system was Linux and the C++ compiler was g++. For a fair comparison, the graph-plotting part of `spring.py`, and the file-writing part of `spring.cpp`, were commented out, so that each program was just performing the Euler iterations. `dt` was decreased from 0.1 to 0.00001, so that the execution took sufficiently long to be timed.

code	`spring.py`	`spring.cpp` standard compilation	`spring.cpp` optimized compilation
time (s)	7.13	0.52	0.17

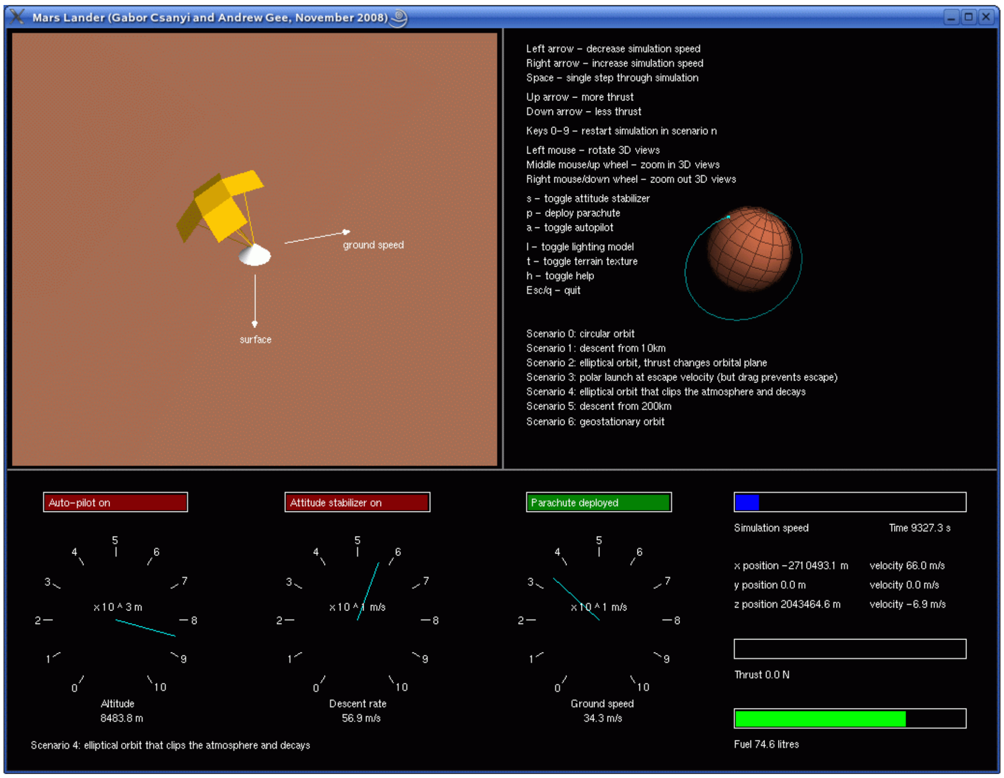
Repeat this experiment yourself, on your own PC, to familiarise yourself with the different ways of running interpreted and compiled software, and the different performances you can expect. You might find that the Python code is even slower if running on a remote notebook server via a web browser, rather than locally on your own PC.

Making progress with C++

Between this appendix and Appendix B, we provide you with enough information to complete Assignments 3–5. To become a proficient C++ programmer, though, requires much more than a couple of days' effort. Although there is a C++ laboratory in Part IB to look forward to, mastering a programming language requires the sort of continuous practice that cannot be timetabled in a laboratory rota.

So, now that you have a compiler installed on your PC, we encourage you to explore the language some more in your own time. There are many excellent online resources, including structured tutorials: why not find one that you like the look of and give it a go?

B The lander program



Code structure

The lander program is supplied in three files:

lander.cpp contains skeleton numerical dynamics functions, which you need to complete

lander_graphics.cpp contains all the other functions, mostly concerned with the graphical visualization

lander.h contains class and structure definitions, global variable declarations, constants and function prototypes[1]

Compilation instructions for Windows (Visual C++), Linux and Mac OS X can be found in the "Getting started" section of the Moodle course page. When you have successfully compiled the program, run it and press 'h' to see the various keyboard and mouse operations for manual control of the lander. You should see something like the screenshot above.

[1] In C++, just as it is necessary to declare all variables before they are used, so is it necessary to declare all functions before they are used: hence the function prototypes in `lander.h`.

The vector3d class

Mechanical simulation requires the manipulation of three-dimensional vectors. To make this easier for you, a `vector3d` class is provided. Unless you are an experienced programmer, you will not have come across C++ classes, so don't worry about how they are defined for now (though have a look in `lander.h` if interested), just try to get a feel for how to use the `vector3d` class. For example

```
vector3d a, b, c;
```

declares three objects, `a`, `b` and `c`, that are three-dimensional vectors. You can access the data elements using the 'dot' notation, for example

```
double x, y, z;
x = a.x; // sets x to the x-component of a
y = a.y; // sets y to the y-component of a
z = a.z; // sets z to the z-component of a
```

The `vector3d` class also provides the necessary functions and operators to manipulate the data elements. Here are some examples:

```
bool t;
c = a + b; // vector addition
c = a - b; // vector subtraction
c = a * x; // vector multiplied by scalar
c = a / x; // vector divided by scalar
t = (a == b); // vector comparison: t set true if a and b are identical
t = (a != b); // vector comparison: t set true if a and b are different
a += b; // a incremented by b
a -= b; // a decremented by b
a *= x; // a scaled by x
a /= x; // a scaled by 1/x
a = b ^ c; // vector product
x = a * b; // scalar product
x = a.abs(); // x set to the magnitude of a
x = a.abs2(); // x set to the squared magnitude of a
b = a.norm(); // normalization: b set to the unit vector parallel to a
```

You should use the `vector3d` class in your numerical simulation for the lander's various state variables (position, velocity, acceleration). This will make your code far more elegant, compact and legible than if you were to use separate `double` variables for every element of every vector.

Global variables

The `lander.h` file includes declarations of a large number of **global variables**. These are variables that are accessible to the main program and every function, without the need to pass them to and fro as parameters.

Here is a list of the global variables that you will probably need to access (and in some cases update) in the functions you write:

```
enum parachute_status_t { NOT_DEPLOYED = 0, DEPLOYED = 1, LOST = 2 };
double fuel; // proportion of fuel remaining (in the range 0.0 to 1.0)
double simulation_time; // elapsed time since start of simulation (s)
double delta_t; // simulation time step (s)
double throttle; // range 0.0 (no thrust) to 1.0 (maximum thrust)
vector3d position; // lander's position (m)
vector3d velocity; // lander's velocity (m/s)
vector3d orientation; // lander's orientation (degrees)
bool autopilot_enabled; // whether the autopilot is enabled
```

```
bool stabilized_attitude; // whether the attitude stabilizer is enabled
int stabilized_attitude_angle; // angle of lander's nose to vertical (degrees)
parachute_status_t parachute_status; // the state of the parachute
```

The C++ `enum` statement associates legible strings with sequences of integers. This allows us to write clauses like

```
if (parachute_status == DEPLOYED)
```

which is much nicer than

```
if (parachute_status == 1)
```

Note that the lander's state vectors are defined in a Cartesian coordinate system whose origin is at the planetary centre. This makes calculating the gravitational force easy: it is in the direction `-position.norm()`.

C++ functions

Although you will not need to create any functions to complete the core assignments, you will need to edit the contents of a few, so it is useful to review here briefly the structure of C++ functions. We shall look at three examples from `lander_graphics.cpp`.

Example function 1

```
vector3d thrust_wrt_world(void)
```

This is the header for the function which calculates the current engine thrust in the world coordinate system (i.e. the coordinate system whose origin is at the planetary centre). It takes no parameters, hence the `void` inside the parentheses. Unlike Python, in C++ we must state explicitly what type of data the function returns, in this case a `vector3d`. The body of the function follows the header and is delineated by curly brackets. Like Python, the function body will typically make use of local variables. Also like Python, the function body ends with a `return` statement. A typical call of this function might look like

```
thr = thrust_wrt_world();
```

where the variable `thr` must have been declared previously with type `vector3d`.

Example function 2

```
void attitude_stabilization(void)
```

This function does not return a value, and is therefore defined with type `void`. Its purpose is to stablize the attitude of the lander so that its base points towards the planetary centre. A typical call would look like

```
attitude_stabilization();
```

Example function 3

```
double atmospheric_density(vector3d pos)
```

This function returns a value of type `double` and also takes one parameter of type `vector3d`. Its purpose is to calculate and return the atmospheric density at a particular position. A typical call would look like

```
d = atmospheric_density(position);
```

where the variable `position` must have been declared and initialized previously with type `vector3d`, and the variable `d` must have been declared previously with type `double`.

Functions you need to edit

For most of this exercise, you need to edit only the three functions in `lander.cpp`.

numerical_dynamics

This is the key function that updates the state variables `position`, `velocity` and `orientation`, using a numerical integrator of your choice. You will first need to calculate the instantaneous forces on the lander (gravity, thrust, drag). We provide some functions to help you with this:

vector3d thrust_wrt_world (void) returns the current thrust (N) in the world coordinate system (i.e. the coordinate system whose origin is at the planetary centre)

void attitude_stabilization (void) automatically adjusts the global variable `orientation` to keep the lander's thruster pointing towards the planetary centre

double atmospheric_density (vector3d pos) returns the density of the atmosphere (kg/m^3) at position `pos`

The visualization routines in `lander_graphics.cpp` use your updates of `position` and `orientation` to display the lander at each time step. `velocity`, however, is ignored: the dials showing descent rate and ground speed are adjusted using the current and previous values of `position`. However, you might need to update `velocity` as part of your numerical integration algorithm.

initialize_simulation

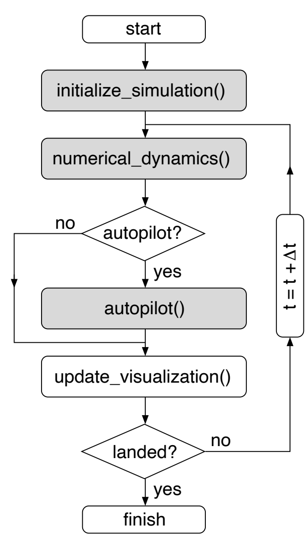
This function sets up the initial conditions of the simulation. The program allows for ten different scenarios, each with different initial conditions. The scenarios are activated by the user pressing a key in the range 0 to 9 when running the simulation. Scenarios 0 to 5 are already defined, but you might want to set up Scenarios 6 to 9 as you see fit. For each additional scenario, you need to provide a short description near the top of the function (for the help screen) and the various initial conditions lower down in the appropriate `case` statement. Use the provided scenarios as templates, and note how to initialize `vector3d` objects, for example:

```
orientation = vector3d(0.0, 0.0, 90.0);
```

autopilot

This function replaces the manual lander control. It should be called from `numerical_dynamics` (though only if the autopilot is enabled) and needs to set the variables `throttle`, `parachute_status` and `stabilized_attitude`.

The flowchart shows how these three functions (the grey boxes) fit in with other elements of the simulation. In particular, note that the main loop over time is handled elsewhere. All you need to do in `numerical_dynamics` is update the lander's state for a single time step.



Constants

A number of useful constants are defined in `lander.h`. You might want to tweak one or two of them for testing and debugging purposes. Here are some of the more important constants:

```
#define MARS_RADIUS 3386000.0 // (m)
#define LANDER_SIZE 1.0 // (m)
#define DRAG_COEF_LANDER 1.0
#define DRAG_COEF_CHUTE 2.0
#define UNLOADED_LANDER_MASS 100.0 // (kg)
#define FUEL_CAPACITY 100.0 // (l)
#define FUEL_DENSITY 1.0 // (kg/l)
#define MARS_MASS 6.42E23 // (kg)
#define GRAVITY 6.673E-11 // (m^3/kg/s^2)
#define FUEL_RATE_AT_MAX_THRUST 0.5 // (l/s)
#define MAX_THRUST 1.5 times lander's fully fuelled surface weight // (N)
#define MAX_PARACHUTE_SPEED 500.0 // (m/s)
#define MAX_PARACHUTE_DRAG 20000.0 // (N)
#define ENGINE_DELAY 0.0 // (s)
#define ENGINE_LAG 0.0 // (s)
#define MARS_DAY 88642.65 // (s)
```

You will need `MARS_RADIUS` to calculate the lander's altitude: this is especially useful for the autopilot and the initial conditions. `LANDER_SIZE` is the radius of the lander's circular base, while each of the five parachute segments is a square with side length `2.0*LANDER_SIZE`: so this is a key constant in calculating the drag on the lander. The drag coefficients of the lander and the parachute are given by `DRAG_COEF_LANDER` and `DRAG_COEF_CHUTE` respectively. The lander's mass is a function of the quantities `UNLOADED_LANDER_MASS`, `fuel`, `FUEL_CAPACITY` and `FUEL_DENSITY`: you will also need the constants `MARS_MASS` and `GRAVITY` to calculate the gravitational force. `FUEL_RATE_AT_MAX_THRUST` is a number you might want to tweak: for example, setting it to zero gives you effectively infinite fuel, making landing and autopilot prototyping far easier! Knowledge of `MAX_THRUST` should help you tune the offset Δ in the autopilot. You will need to refer to `MAX_PARACHUTE_SPEED` and `MAX_PARACHUTE_DRAG` when writing sophisticated autopilots that use the parachute. If you open the parachute while travelling above `MAX_PARACHUTE_SPEED`, the temperature of the gas in the shock layer will vaporize the fabric. Likewise, the fabric and tethers will fail if the total mechanical load on the parachute exceeds `MAX_PARACHUTE_DRAG`. Alternatively, you could use the provided function `bool safe_to_deploy_parachute(void)` that does the calculations for you. Throttle changes are delayed by `ENGINE_DELAY` before reaching the engine. Thereafter, the engine's response might be sluggish: `ENGINE_LAG` is the **time constant**, the time for the engine to reach 63% of the requested thrust. Leave these two constants set to zero at first, though for more challenging manual and autopilot landing, try setting them to realistic, nonzero values. Finally, `MARS_DAY` is a constant you will need when calculating the radius and velocity of the areostationary orbit.

Static variables

One final C++ concept you might find useful is the **static variable**. This is best explained by way of an example. Suppose you declare a local variable in `numerical_dynamics` as follows:

```
static vector3d previous_position;
```

Compared with a normal local variable, the difference is that `previous_position` is *not* destroyed when the function returns. The next time `numerical_dynamics` is called, the contents of the variable `previous_position` are preserved intact from the previous call. So you can use static variables to remember data from one call to the next. You will need to do something like this if implementing a higher order integrator that requires knowledge of not just the current position, but also the previous one. You could alternatively use a global variable, but the static option is nicer if the data can be localized to a particular function.