The Extended Mars Lander
Exercise 1: Euler's equations and quaternions

A.M.P. Tombs
arthur.tombs@cantab.net

September 18, 2013

1 Introduction

This exercise is an extension to the Mars Lander exercise presented in Part IA of the Engineering Tripos, in which students implemented numerical simulation of the landing of a spacecraft (termed 'the lander') on Mars. The exercise covered numerical simulation of three-dimensional positional dynamics in both Octave/Matlab and C++, using a Verlet integration scheme. The program was subsequently analysed as part of the linear control theory course in Part IB, in order to design an autopilot to make the lander hover. Knowledge of the original exercise is assumed and highly recommended, but will not be essential for the completion of this document. This document aims to lead students through extending the Mars Lander simulation to cover rotational rigid-body dynamics, which were previously not included for reasons of simplicity. The content of this document is summarized here:

- A recap of the content from 3C5: Euler's equations of rigid-body motion, and Euler angles.
- An explanation of gimbal lock.
- The use of quaternions as an alternative to Euler angles.
- The Velocity-Verlet integrator.
- Tasks for the student to complete.

2 Euler's equations

Euler's equations are a set of linear differential equations that govern the dynamical behaviour of a rigid body in three dimensions. The equations are most commonly expressed in their *body-fixed* form, meaning that the angular-velocity components ω_i and torques Q_i are specified with the principal axes of the body as a reference frame. The values A, B and C are the moments of inertia of the body about each of the principal axes, and are constants in this reference frame.

$$\begin{aligned} A\dot{\omega}_1 - (B-C)\omega_2\omega_3 &= Q_1 \\ B\dot{\omega}_2 - (C-A)\omega_3\omega_1 &= Q_2 \\ C\dot{\omega}_3 - (A-B)\omega_1\omega_2 &= Q_3 \end{aligned} \tag{1}$$

For our purposes, these may be written more concisely as a single matrix equation.

$$\dot{\underline{\omega}} = I^{-1}\left((I\underline{\omega} \times \underline{\omega}) + \underline{Q}\right) \tag{2}$$

In this form, the matrix I is composed of the principal moments of inertia of the body.

$$I = \begin{bmatrix} A & 0 & 0 \\ 0 & B & 0 \\ 0 & 0 & C \end{bmatrix}$$

You may want to expand Equation (2) to show that it is equivalent to the longer form above.

3 Euler angles

To specify the *pose* of a body in 3D, a transformation must be defined that maps between *world coordinates* and *local coordinates*. This transformation comprises two parts: a *translation* (position), and a *rotation* (orientation), both relative to the initial reference frame. While position can be simply represented as a 3D vector (as seen in the Part I exercise), there are several ways to define a rotation. While a 3×3 rotation matrix is the most direct representation, it has the disadvantage of containing redundant information; the nine matrix elements represent just three degrees of freedom between them. To reduce the amount of computation, and accumulation of precision errors, it is more usual to select a reduced set of parameters for numerical simulation.

The *Euler angle* approach is to specify three angles (usually given symbols ϕ, θ and ψ) which are used to rotate around three separate axes in turn. The order of rotations, and the choice of which axes to use, varies depending on the context; a commonly used set of rotation axes is the ZXZ configuration, particularly for scenarios that have a clearly defined ‘spin’ axis e.g. gyroscopes.

$$\begin{aligned}
R &= R_z(\psi)R_x(\theta)R_z(\phi) \\
&= \begin{bmatrix} \cos\psi & \sin\psi & 0 \\ -\sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & \sin\theta \\ 0 & -\sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} \cos\phi & \sin\phi & 0 \\ -\sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{bmatrix} \\
&= \begin{bmatrix} \cos\psi\cos\phi - \sin\psi\cos\theta\sin\phi & \cos\psi\sin\phi + \sin\psi\cos\theta\cos\phi & \sin\psi\sin\theta \\ -\sin\psi\cos\phi - \cos\psi\cos\theta\sin\phi & -\sin\psi\sin\phi + \cos\psi\cos\theta\cos\phi & \cos\psi\sin\theta \\ \sin\theta\sin\phi & -\sin\theta\cos\phi & \cos\theta \end{bmatrix}
\end{aligned} \tag{3}$$

The convention adopted here is that the combined matrix R rotates vectors from world to local coordinates i.e.

$$\underline{x}_{\text{local}} = R\underline{x}_{\text{world}}$$

As with all orthonormal matrices $R^{-1} = R^T$, which would describe the reverse mapping – you will find that some other sources define rotation matrices as the transpose of these.

3.1 Angular velocity

Although Euler's equations for angular velocity are independent of any orientation, the presence of an angular velocity vector implies that the orientation of the body must be changing. To complete the set of equations needed to describe rigid-body dynamics, an expression is required that links angular velocity to the Euler angles used to represent orientation. The following equation can be written down by considering incremental changes in each of the three rotation axes.

$$\begin{aligned}
\underline{\omega} &= \begin{bmatrix} 0 \\ 0 \\ \dot\psi \end{bmatrix} + R_\psi \begin{bmatrix} \dot\theta \\ 0 \\ 0 \end{bmatrix} + R_\psi R_\theta \begin{bmatrix} 0 \\ 0 \\ \dot\phi \end{bmatrix} \\
&= \begin{bmatrix} \sin\psi\sin\theta & \cos\psi & 0 \\ \cos\psi\sin\theta & -\sin\psi & 0 \\ \cos\theta & 0 & 1 \end{bmatrix} \begin{bmatrix} \dot\phi \\ \dot\theta \\ \dot\psi \end{bmatrix} \\
&= J^{-1}\underline{\dot{E}}
\end{aligned} \tag{4}$$

This matrix can be inverted to give the rate of change of each Euler angle.

$$\begin{aligned}
\begin{bmatrix} \dot\phi \\ \dot\theta \\ \dot\psi \end{bmatrix} &= \begin{bmatrix} \dfrac{\sin\psi}{\sin\theta} & \dfrac{\cos\psi}{\sin\theta} & 0 \\ \cos\psi & -\sin\psi & 0 \\ \dfrac{-\cos\theta\sin\psi}{\sin\theta} & \dfrac{-\cos\theta\cos\psi}{\sin\theta} & 1 \end{bmatrix} \underline{\omega} \\
&= J\underline{\omega}
\end{aligned} \tag{5}$$

For calculating second derivatives (as is required for the Verlet integrator), this expression can be differentiated with respect to time by using the product and chain rules of differentiation.

$$\underline{\ddot{E}} = \dot{J}\underline{\omega} + J\underline{\dot\omega} \tag{6}$$

$$\dot{J} = \begin{bmatrix} \dfrac{-\sin\psi}{\sin\theta\tan\theta}\dot\theta + \dfrac{\cos\psi}{\sin\theta}\dot\psi & \dfrac{-\cos\psi}{\sin\theta\tan\theta}\dot\theta + \dfrac{-\sin\psi}{\sin\theta}\dot\psi & 0 \\ -\dot\psi\sin\psi & -\dot\psi\cos\psi & 0 \\ \dfrac{\sin\psi}{\sin^2\theta}\dot\theta + \dfrac{-\cos\theta\cos\psi}{\sin\theta}\dot\psi & \dfrac{\cos\psi}{\sin^2\theta}\dot\theta + \dfrac{\cos\theta\sin\psi}{\sin\theta}\dot\psi & 0 \end{bmatrix} \tag{7}$$

3.2 Gimbal lock

It is apparent from Equation (5) that when $\sin\theta = 0$, evaluating J will involve dividing by zero – this is known as a singularity. In practice, the form of this expression causes the calculation to become poorly conditioned for all small values of $\sin\theta$, a phenomenon known as *close encounter*.

It should be noted that in this configuration a change in either ϕ or ψ will affect the rotation matrix R in the same manner. This is equivalent to observing that the constructed matrix J^{-1} is rank deficient, and so there is no unique solution to the conversion from angular velocity to Euler angles.

While it is possible to reduce the effects of gimbal lock for any given scenario (e.g. by choosing a different set of rotation axes), all approaches that use Euler angles will contain these ‘forbidden’ states.

4 Quaternions

In order to avoid the problems of gimbal lock, it is common to use *quaternions* instead of Euler angles for specifying orientation. A quaternion is a collection of four numbers (also known as *Euler parameters*), which can be written either as a four-dimensional vector, or as a scalar value paired with a three-dimensional vector.

$$\mathbf{q} \equiv \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} \equiv \begin{bmatrix} s \\ \underline{v} \end{bmatrix} \tag{8}$$

A rotation in 3D space can be described by a single quaternion, which makes them ideal for keeping track of orientation. A general axis-angle representation of rotation – by an angle θ about an axis $\underline{\hat{a}}$ (where $\underline{\hat{a}}$ is a unit vector) – can be converted simply into a quaternion:

$$\mathbf{q}_r = \begin{bmatrix} \cos\frac{\theta}{2} \\ \sin\frac{\theta}{2} \times \underline{\hat{a}} \end{bmatrix} \tag{9}$$

A *unit* quaternion is one which represents a pure rotation. This is when it has a unit l_2 norm (i.e. the sum of the squared components is equal to 1). You may want to verify, using standard trigonometric identities, that Equation (9) will always produce a unit quaternion.

4.1 Operations

Quaternions can be added and subtracted in the same manner as vectors (i.e. component-wise). Additionally, a quaternion multiplication operation is defined.

$$\mathbf{q}_1 \times \mathbf{q}_2 \equiv \begin{bmatrix} s_1 \\ \underline{v}_1 \end{bmatrix} \times \begin{bmatrix} s_2 \\ \underline{v}_2 \end{bmatrix} \equiv \begin{bmatrix} s_1 s_2 - (\underline{v}_1 \cdot \underline{v}_2) \\ s_1\underline{v}_2 + s_2\underline{v}_1 + \underline{v}_1 \times \underline{v}_2 \end{bmatrix} \tag{10}$$

Note that, due to the cross-product term ($\underline{v}_1 \times \underline{v}_2$), this operation is not commutative (i.e. the order matters, as with matrix algebra).

Due to the relationship between quaternions and complex numbers, the conjugate of a quaternion is also defined:

$$\mathbf{q}^* \equiv \begin{bmatrix} s \\ -\underline{v} \end{bmatrix} \tag{11}$$

To transform a vector by a quaternion, a process known as conjugation (not to be confused with the conjugate) is used. The vector to be transformed is treated as a quaternion with $s = 0$, and the transformation is then performed with quaternion multiplications.

$$\begin{bmatrix} 0 \\ \underline{x}' \end{bmatrix} = \mathbf{q}^* \times \begin{bmatrix} 0 \\ \underline{x} \end{bmatrix} \times \mathbf{q} \tag{12}$$

A 3×3 rotation matrix (using the same rotation convention as in Equation (3)) can be constructed by transforming each of the basic vectors $\underline{i}$, $\underline{j}$ and $\underline{k}$ with Equation (12).

$$R = \begin{bmatrix} s^2 + q_1^2 - q_2^2 - q_3^2 & 2(q_1q_2 + sq_3) & 2(q_1q_3 - sq_2) \\ 2(q_1q_2 - sq_3) & s^2 - q_1^2 + q_2^2 - q_3^2 & 2(sq_1 + q_2q_3) \\ 2(sq_2 + q_1q_3) & 2(q_2q_3 - sq_1) & s^2 - q_1^2 - q_2^2 + q_3^2 \end{bmatrix} \tag{13}$$

4.2 Angular velocity

As before, an expression is needed to convert between angular velocity and the rate of change of orientation. To do this, it is helpful to consider the meaning of the angular velocity vector: its direction describes an axis about which the body is rotating, and its magnitude represents the instantaneous rotation rate. The rate of change of the orientation quaternion is found by multiplying $\mathbf{q}$ by the angular velocity (written as a quaternion):

$$\dot{\mathbf{q}} = \frac{1}{2}\left(\mathbf{q} \times \begin{bmatrix} 0 \\ \underline{\omega} \end{bmatrix}\right) = \frac{1}{2}\begin{bmatrix} -\underline{v} \cdot \underline{\omega} \\ s\underline{\omega} + \underline{v} \times \underline{\omega} \end{bmatrix} \tag{14}$$

A similar expression for angular acceleration can be found by differentiation.

$$\ddot{\mathbf{q}} = \frac{1}{2}\left(\mathbf{q} \times \begin{bmatrix} 0 \\ \dot{\underline{\omega}} \end{bmatrix}\right) - \frac{1}{4}\|\underline{\omega}\|^2\mathbf{q} \tag{15}$$

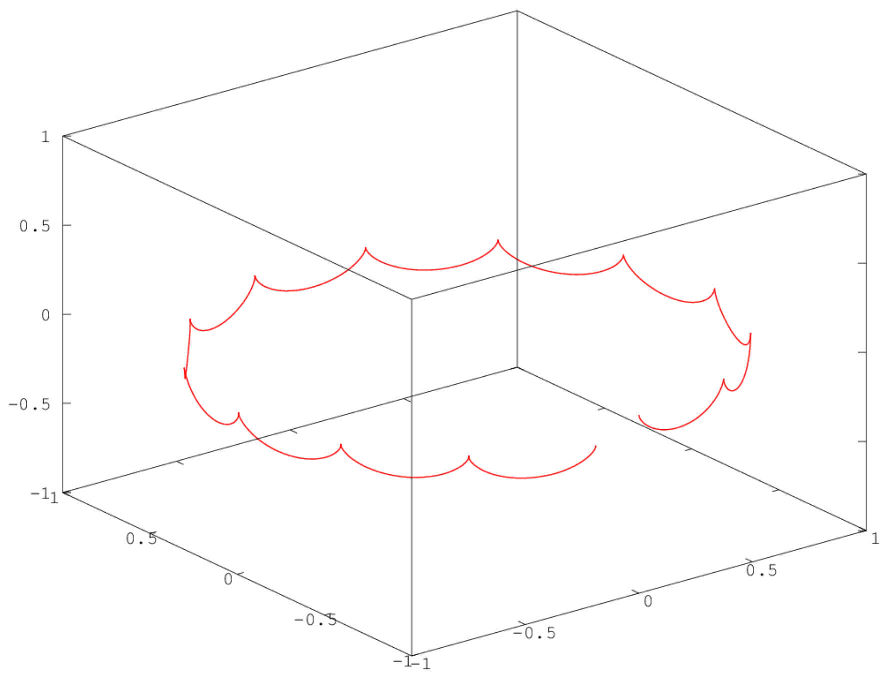

Figure 1: The plot that should be produced by the supplied program, showing the motion of the z-axis of a gimballed rotor undergoing precession and nutation.

5 Velocity-Verlet integration

The software sources given in the following tasks use the *Velocity-Verlet* scheme for numerical integration. This scheme has several benefits over traditional Verlet (as was used in the original Mars Lander exercise), e.g. there being no special case needed for the first time step. Each time-step evaluation (calculating $\underline{r}$ and $\underline{v} \equiv \dot{\underline{r}}$) can be summarized with the following four steps:

1. $\underline{v}(t + \frac{\Delta t}{2}) = \underline{v}(t) + \frac{1}{2}\dot{\underline{v}}(t)\Delta t$
2. $\underline{r}(t + \Delta t) = \underline{r}(t) + \underline{v}(t + \frac{\Delta t}{2})\Delta t$
3. Estimate $\dot{\underline{v}}(t + \Delta t)$ using $\underline{r}(t + \Delta t)$
4. $\underline{v}(t + \Delta t) = \underline{v}(t + \frac{\Delta t}{2}) + \frac{1}{2}\dot{\underline{v}}(t + \Delta t)\Delta t$

For a more detailed explanation of Verlet-like methods, there is a good page available on Wikipedia.

6 Tasks

1. You are provided with an implementation of the Euler angle method from section 3 (a file named `Exercise_01_euler.m`). Firstly, verify that you can run this program in Octave/Matlab without modification. It should produce a graph similar to Figure 1.

   The code is set up to simulate a spinning rotor held within a freely rotating gimbal mount, with a weight applied to one end (see Figure 2). The plot shows a track of where the z axis of the rotor is pointing over time, relative to the mount point at the origin. Observe the *cuspidal motion* of the rotor consistent with releasing it from rest, as may be familiar from the 3C5 laboratory experiment.

   The code is written such that the accelerations are calculated in a separate function from the main integrator (in `calculate_accelerations.m`). For this simple system there is very little code in this function, but in principle it could include much more complicated dynamics (as we shall see). Make sure you understand the two lines that calculate the torque and angular acceleration terms in this case.

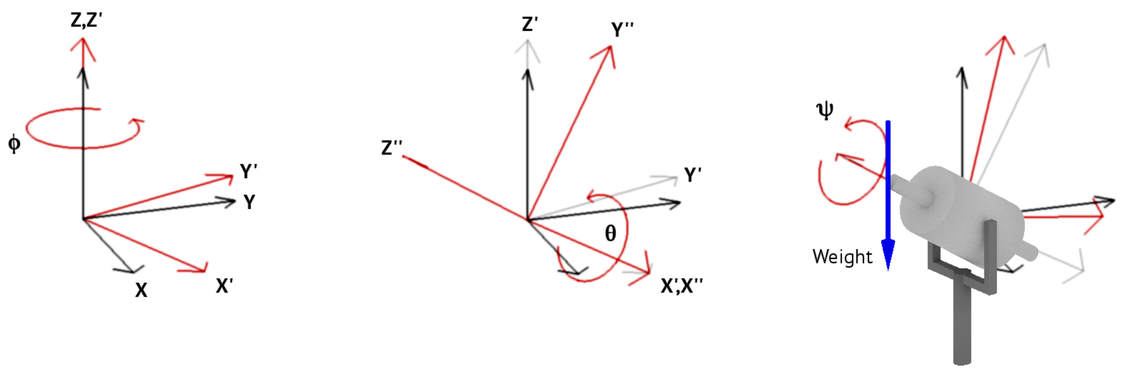

Figure 2: A battery powered rotor held in a gimbal mount, similar to the 3C5 laboratory experiment. The black axes show the global coordinate system, and the red axes show the principal axes of the body.

2. The initial conditions for this simulation are such that the rotor remains approximately horizontal, with the Euler angles far from any singularity points.

   To observe a trivial gimbal lock scenario, set the initial orientation to `[0;0;0]` and the angular velocity to `[0;1;0]`. The rotor is now acting as a pendulum (as there is no spin), released from the upright position with an initial angular velocity in θ. What motion would you expect from this setup? What actually happens?

   The program protects against gimbal lock by generating a warning when $\sin\theta$ is equal to zero, but this does not completely avoid the problems of close-encounter. Try some more dynamic conditions that come close to the singularity point, and observe that the simulation does not always behave as expected. e.g. `E = [0;pi/2;0];` and `omega = [0;1;0];`

3. Your task is to write a similar simulation that uses quaternions in place of Euler angles. To do this, you are provided with the main framework of the program as `Exercise_01_skeleton.m`, also included at the end of this document.

   - As a first step, replace the initial vector `E` with an equivalent quaternion (it is suggested that you name it `q` to save confusion). You will need to use Equation (9) for this – consider what rotation axis will match with the second Euler angle rotation.

     Note that the rest of the supplied code assumes the quaternion to be a 4×1 vector.
   - Next, complete the portion of the code that updates the orientation at each time step. You will need to refer to Equation (14) for calculating the rate of change of the quaternion. After updating the quaternion, add a new line that normalizes it back to unit length – this procedure will reduce error accumulated over time, thus improving the simulation accuracy.

     A function `quat_mult` has been provided for calculating quaternion multiplications.

4. Test your new quaternion implementation with the same test cases as before. You should observe that there is no gimbal lock phenomenon in any configuration.

The following tasks relate to the Lander C++ code (provided from the Camtools site in a zip archive). At this point you should make sure that you have the necessary libraries and programs to compile and run the Lander (details are on the Camtools site). You will only need to edit the file `lander.cpp` for this exercise.

The Lander program is provided with a working integrator (Velocity-Verlet, as seen above) that simulates the lander motion as in the Part I exercise. Run the program to remind yourself how it works, and observe that the orientation of the lander remains fixed in all scenarios (unless the attitude stabilizer is engaged). Your task is now to extend the code to also include the rotational dynamics discussed in this document.

1. Begin with function `calculate_accelerations` in `lander.cpp`. This function has exactly the same purpose as the equivalent Matlab function: it takes the current system state (position, velocity etc.), and calculates accelerations (linear and rotational) based on the system dynamics.

   Add to this function the calculation of `omega_dot`, using the same equation as before. Assume for now that the `torque` vector is already calculated.

NB: The Lander C++ code provides the operator ^ for vector cross-products.
You will need to write matrix multiplications involving I in terms of the three diagonal components A, B and C e.g. `vector3d(A*omega.x, B*omega.y, C*omega.z)`.

Uncomment the two lines in `numerical_dynamics` that update the orientation and angular velocity variables. Now compile and run the program, and view scenario 8 – this sets the lander a large distance from the planet, so it is essentially spinning in free space. You should observe the lander spinning anti-clockwise at a constant rate – anything else indicates a mistake somewhere in your code.

2. In this simulated system, the torque acting on the lander is created by the aerodynamics and the thrusters (think why gravitational attraction does not generate a torque?). The aerodynamical component of `torque` is already calculated by the code, but you will need to add on a component from the thrusters (currently, the function `torque_from_thrusters()` returns a null vector).

   The thruster parameters are stored in three arrays, named `thruster_direction`, `thruster_position` and `throttle`. Write a `for` loop over the `N_THRUSTERS`, that sums the torque contributed by each of them (all vectors are expressed in the body-fixed coordinate system, so you need not deal with rotation matrices). The formula for calculating torque is identical to the one used in the Octave/Matlab example.

   Note that the force acting on the lander is in the opposite direction to the thruster's direction, according to Newton's third law:
$$\underline{F}_i = -\hat{\underline{d}}_i \times \text{throttle} \times \text{MAX\_THRUST}$$

   Again run scenario 8, and note that there is just one thruster on the lander. Pause the simulation and try adding a little thrust (a throttle of 0.2 is sufficient). Advancing the simulation slowly, you should observe that the lander begins to spin away from the direction of thrust, as you would expect.

3. Try running scenario 6, which demonstrates the lander falling in an uncontrolled manner. Using the provided throttle control (which affects all thrusters at once) is it possible to stabilize the fall and land safely? What would be needed to stabilize the tumble?

Extension exercises for you to think about (optional):

- When the lander is spinning very fast (e.g. after using all fuel up in scenario 8), how does the visualization of the lander change between time steps, when stepping through the simulation? How does this relate to the concept of *aliasing* in signal processing?
- What effect does fast spin have on the accuracy and stability of the integrator? Can the value of `delta_t` be changed to improve this in all cases?
- When in circular orbit (scenario number 0), the climb rate of the lander varies as a small-amplitude sinusoid (use the graphing view to observe this). What it the root cause of this? How does it compare with scenario 2?
- Implement a higher-order integrator (e.g. Fourth-order Runge-Kutta) and compare relative performance and accuracy.
- How do the principal moments of inertia of the lander affect the stability of different types of motion? What about the stability of the simulation?

```
% Define some global variables to share with functions
global I Ii F r;

% Define moments of inertia and compose inertia matrix I
A = 2;
B = 2;
C = 4;

% Compute the inertia matrix and its inverse
I  = diag([A,B,C]);
Ii = diag(1 ./ [A,B,C]);

% Define initial conditions for the simulation
% !!! Students, begin here !!!
q     = [     ];     % Orientation - 4x1 quaternion
omega = [0;0;4];     % Body-fixed angular-velocity (rad/s)

% Define simulation parameters
dt = 0.01;  % Time per iteration
N  = 1000;  % Number of iterations

% Constant force acting on the body
F = [0;0;-10];  % Force acting, in global coordinates
r = [0;0;1];    % Point of action, in body coordinates

% Track z-axis over time
track = zeros(N,3);

% Construct rotation matrix R from orientation
R = quat_to_matrix(q);

% Loop over all time steps
for n = 1:N

  % Store z-axis at this time step
  track(n,:) = R(3,:);

  % Calculate angular accelerations
  omega_dot = calculate_accelerations(R, omega);

  % Calculate angular velocity at half timestep
  omega_half = omega + omega_dot * (dt * 0.5);

  % Calculate new orientation estimate
  % !!! Students, you will need to complete this !!!
  q_dot =  ;
  q     = q + q_dot * dt;

  % Construct rotation matrix R from orientation
  R = quat_to_matrix(q);

  % Calculate new angular accelerations
  omega_dot = calculate_accelerations(R, omega + omega_dot * dt);

  % Calculate updated angular velocity estimate
  omega = omega_half + omega_dot * (dt * 0.5);

end

% 3D plot of z-axis over time
figure(1);
clf;
plot3(track(1:2:n,1), track(1:2:n,2), track(1:2:n,3));
axis([-1 1 -1 1 -1 1]);
```

Exercise_01_skeleton.m