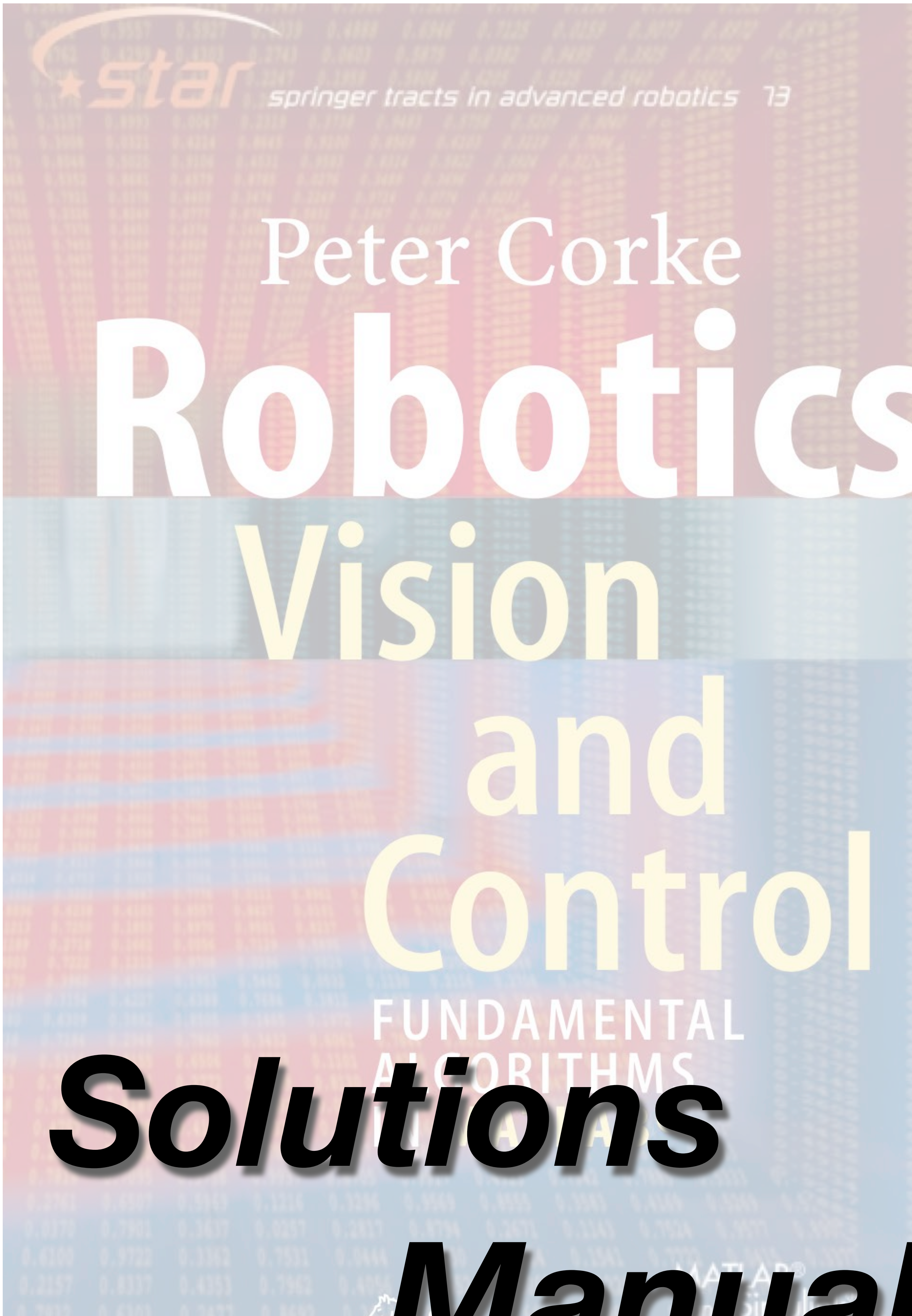


star springer tracts in advanced robotics 73

Peter Corke

Robotics Vision and Control

FUNDAMENTAL ALGORITHMS IN MATLAB

Solutions Manual

Contents

Introduction

This manual contains the solutions for end-of-chapter exercises posed in the book "Robotics, Vision & Control". The exercises are essentially of three different types:

1. Experimenting with Toolbox functions, varying the parameters of examples presented in the book. This *hands on* activity is useful in increasing understanding of an algorithm as well as understanding the limits to its performance.
2. Writing new code to extend the functionality of the Toolboxes.
3. Derivation of underlying principles.
4. Other

The style is somewhat like the book, showing the MATLAB commands and results (both numeric and graphical) as well as some discussion. In general there are multiple ways to solve most problems but in most cases only one way is presented — this is the way that I'd solve the problem but that doesn't mean it's the best way.

Part I

Foundations

Chapter 2

Representing position and orientation

Q1 We can easily evaluate this numerically, firstly for the case where all angles are positive

```
>> R = rpy2r(0.1, 0.2, 0.3);
>> tr2rpy(R)
ans =
    0.1000    0.2000    0.3000
```

and we see that the angles, after conversion to a rotation matrix and back, are all positive. For the cases where one of the angles is negative

```
>> R = rpy2r(-0.1, 0.2, 0.3);
>> tr2rpy(R)
ans =
   -0.1000    0.2000    0.3000

>> R = rpy2r(0.1, -0.2, 0.3);
>> tr2rpy(R)
ans =
    0.1000   -0.2000    0.3000

>> R = rpy2r(0.1, 0.2, -0.3);
>> tr2rpy(R)
ans =
    0.1000    0.2000   -0.3000
```

we can see that sign is preserved after tansformation to an orthonormal rotation matrix and back again. This is not the case for Euler angles.

Try experimenting with angles whose magnitude is $> \pi/2$ and $> pi$.

Q2 The function `trplot` has many options. First we will create a transform to display

```
>> T = transl(1, 2, 3) * rpy2tr(0.1, 0.2, 0.3);
```

and then plot it

```
>> trplot(T)
```

We can make the axes a different color, for example red

```
>> trplot(T, 'color', 'r')
```

or label the frame with a name, for example $\{A\}$

```
>> trplot(T, 'frame', 'A')
```

in which case the frame name is displayed near the origin and the axes are labelled X_A, Y_A and Z_A.

By default the axes are indicated by unit-length line segments. The option `'arrow'` uses the third party function `arrow3` (which is provided with RTB) to draw 3D arrows with conical heads.

```
>> trplot(T, 'arrow', 'axis', [0 5 0 5 0 5])
```

In this case we need to provide the `'axis'` option to define the plot volume: x, y and z all span the range [0,5] (see `help axis` for details). The conical arrow heads are automatically chosen in proportion to the size of the axes with respect to the overall plot, and the value can be adjusted by providing the `'width'` option.

You can also show the frame in 3D by

```
>> trplot(T, '3d')
```

but you need red/cyan anaglyph glasses (the cheap 3D glasses with coloured lenses) to view the figure and get the 3D effect.

Q3 To show translation along the x-, y- or z-axes we can use

```
>> tranimate( transl(1, 0, 0) )
>> tranimate( transl(0, 1, 0) )
>> tranimate( transl(0, 0, 1) )
```

To show rotation about the x-, y- or z-axes we can use

```
>> tranimate( trotx(pi) )
>> tranimate( troty(pi) )
>> tranimate( trotz(pi) )
```

which shows the coordinate frame rotating one half revolution. Note that providing an argument of 2π does not leads to rotation of one full revolution, in fact it gives zero rotation since the initial and final rotation matrix will be equal.

Q4a A cube has 8 vertices and 12 edges. We will define a cube centred at the origin with a side length of two (it makes life a bit easier, all the coordinates are ± 1). The vertices are listed in clockwise order looking down (in the negative z-axis direction), first the top plane then the bottom plane, with one row per vertex

```
V = [
 1  1  1
 1 -1  1
-1 -1  1
-1  1  1
 1  1 -1
 1 -1 -1
-1 -1 -1
-1  1 -1]';
```

which we then transpose to obtain one vertex per column which is the Toolbox convention for a set of points.

The simplest way to draw the cube is one edge at a time, and we define a simple helper function

```
function drawline(V, v1, v2)
   V1 = V(:,v1); V2 = V(:,v2);
   plot3( [V1(1) V2(1)], [V1(2) V2(2)], [V1(3) V2(3)]);
 end
```

where V is the matrix of points, and v1 and v2 are the indices of the vertices (range 1 to 8) between which lines are drawn. Then to draw the cube we can write

```
    % edges of top face
    drawline(V, 1, 2);
    drawline(V, 2, 3);
    drawline(V, 3, 4);
    drawline(V, 4, 1);

    % edges of bottom face
    drawline(V, 5, 6);
    drawline(V, 6, 7);
    drawline(V, 7, 8);
    drawline(V, 8, 5);

    % vertical edges (between top and bottom faces)
    drawline(V, 1, 5);
    drawline(V, 2, 6);
    drawline(V, 3, 7);
    drawline(V, 4, 8);
```

which has a large number of function calls. This code is in ch2_4a.m.

A more concise alternative uses the MATLAB plot3 function which is typically called as plot3(x, y, z) where each of x, y and z are vectors. The i'th point has coordinates x(i), y(i) and z(i) and plot3 draws a line from the first point, the second point to the third point and son on. A little known trick is to insert NaN values into the vectors which breaks the drawing process, like lifting the pen. We define a column vector of NaN values

```
    sep = [NaN; NaN; NaN];
```

and then build up a list of points (each column represents a point) that define line segments

```
    L = [ V(:,1:4) V(:,1) sep V(:,5:8) V(:,5) sep V(:,1) V(:,5) ...
            sep V(:,2) V(:,6) sep V(:,3) V(:,7) sep V(:,4) V(:,8) ];
```

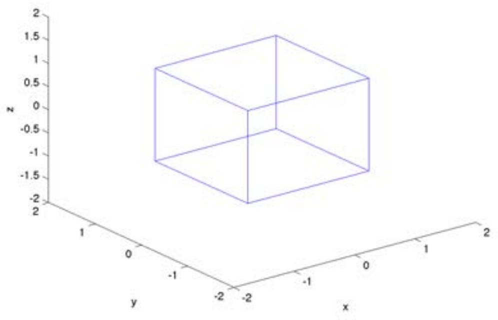


Figure 2.1: Q4a. Wireframe display of cube.

which in this case has a value (first 11 columns only) of

```
>> L =
  Columns 1 through 12
     1     1    -1    -1     1   NaN     1     1    -1    -1     1   NaN
     1    -1    -1     1     1   NaN     1    -1    -1     1     1   NaN
     1     1     1     1     1   NaN    -1    -1    -1    -1    -1   NaN

  Columns 13 through 23
     1     1   NaN     1     1   NaN    -1    -1   NaN    -1    -1
     1     1   NaN    -1    -1   NaN    -1    -1   NaN     1     1
     1    -1   NaN     1    -1   NaN     1    -1   NaN     1    -1
```

and we can see the columns of `NaN` values that break the line segments. The first row are the x-coordinates, the second row the y-coordinates and the third row the z-coordinates (this is the convention used by the Toolboxes). To plot the line is

```
      plot3(L(1,:)', L(2,:)', L(3,:)');
```

and the result is the same as the version above. This code is in `ch2_4a_2.m`.

Q4b The advantage of keeping the points in the Toolbox convention, one point per column, is that we can easily apply a homogeneous transformation to all the points

```
LT = homtrans(T, L);
```

resulting in another set of points `LT` which we then plot

```
plot3(LT(1,:)', LT(2,:)', LT(3,:)');
```

The complete function is given in `ch2_4b`.

Q4c Now that we display the cube with an arbitrary pose it is easy to animate it for rotation about the x-axis

```
for theta=[0:.1:2*pi]
  ch2_4b( trotx(theta) );
  pause(0.1)
end
```

MATLAB does not draw graphics right away, it saves the graphics commands away until either a `pause` or `drawnow` command is encountered.

Q4d It is a trivial extension to animate the cube for rotation about multiple axes at once

```
for t=[0:0.1:20]
  ch2_4b( trotx(t*1.5)*troty(t*1.2 )*trotx(t) );
  pause(0.1)
end
```

where the rate of rotation about each axis is different.

There are more elegant ways of implementing this. We note that at every animation step we clear the figure, redraw the axes, labels etc. and then redraw all the line segments. Instead of erasing all the lines and redrawing we can change the line by creating a dummy line (just a point)

```
h = plot3(0, 0, 0);
```

and saving a handle to it which is stored in `h`. Then at a subsequent time step instead of clearing and redrawing we just change the data associated with the line

```
set(h, 'Xdata', LT(1,:), 'Ydata', LT(2,:), 'Zdata', LT(3,:));
```

An even more elegant approach is to use a somewhat recent addition to MATLAB

```
    h = hgtransform;
    plot3(L(1,:)', L(2,:)', L(3,:)', 'Parent', h);
```

where we create an `hgtransform` object which we set as the parent of the line segments which define the cube in its unrotated configuration. The example code `ch2_4d` implements this

```
h = ch2_4d();
```

and returns the handle to the `hgtransform` object. Now to rotate all the line segments we simply set the 4×4 homogeneous transformation matrix within that handle

```
set(h, 'Matrix', trotx(0.3) )
```

and the cube is almost instantly redrawn in the new orientation.

Q5 If $\boldsymbol{T}$ is a homogeneous transformation show that $\boldsymbol{T}\boldsymbol{T}^{-1} = \boldsymbol{I}$. From page 38 of RVC we have

$$\boldsymbol{T} = \begin{pmatrix} \boldsymbol{R} & \boldsymbol{t} \\ \boldsymbol{0}_{1\times 3} & 1 \end{pmatrix}$$

and

$$\boldsymbol{T}^{-1} = \begin{pmatrix} \boldsymbol{R}^T & -\boldsymbol{R}^T\boldsymbol{t} \\ \boldsymbol{0}_{1\times 3} & 1 \end{pmatrix}$$

Expanding the left hand side of our problem, $\boldsymbol{T}\boldsymbol{T}^{-1}$, we write

$$\begin{aligned}
&\begin{pmatrix} \boldsymbol{R} & \boldsymbol{t} \\ \boldsymbol{0}_{1\times3} & 1 \end{pmatrix} \begin{pmatrix} \boldsymbol{R}^T & -\boldsymbol{R}^T\boldsymbol{t} \\ \boldsymbol{0}_{1\times3} & 1 \end{pmatrix} \\
&= \begin{pmatrix} \boldsymbol{R}\boldsymbol{R}^T + \boldsymbol{t}\boldsymbol{0}_{1\times3} & -\boldsymbol{R}\boldsymbol{R}^T\boldsymbol{t} + \boldsymbol{t} \\ \boldsymbol{0}_{1\times3}\boldsymbol{R}^T + \boldsymbol{0}_{1\times3} & -\boldsymbol{0}_{1\times3}\boldsymbol{R}^T\boldsymbol{t} + 1 \end{pmatrix}
\end{aligned}$$

and eliminating terms multiplied by zero we obtain

$$= \begin{pmatrix} \boldsymbol{R}\boldsymbol{R}^T & -\boldsymbol{R}\boldsymbol{R}^T\boldsymbol{t} + \boldsymbol{t} \\ \boldsymbol{0}_{1\times3} & 1 \end{pmatrix}$$

Since $\boldsymbol{R}$ is orthogonal we know (from page 27 of RVC) that $\boldsymbol{R}^T = \boldsymbol{R}^{-1}$

$$= \begin{pmatrix} \boldsymbol{R}\boldsymbol{R}^{-1} & -\boldsymbol{R}\boldsymbol{R}^{-1}\boldsymbol{t} + \boldsymbol{t} \\ \boldsymbol{0}_{1\times3} & 1 \end{pmatrix}$$

and applying the identity $\boldsymbol{R}\boldsymbol{R}^{-1} = \boldsymbol{I}_{3\times3}$ we write

$$\begin{aligned}
&= \begin{pmatrix} \boldsymbol{I}_{3\times3} & -\boldsymbol{t} + \boldsymbol{t} \\ \boldsymbol{0}_{1\times3} & 1 \end{pmatrix} \\
&= \begin{pmatrix} \boldsymbol{I}_{3\times3} & 0 \\ \boldsymbol{0}_{1\times3} & 1 \end{pmatrix} \\
&= \boldsymbol{I}_{4\times4}
\end{aligned}$$

Q7 The short answer is *most probably* in the Musee de l'Homme, Paris. Rene Descartes died in Stockholm on 2 February 1650 and as a Roman Catholic in a Protestant nation, he was buried in a corner of a graveyard reserved for aliens at the Adolf Fredriks kyrka in Stockholm. In 1666 in accordance with the wishes of his friends his remains were returned to France. Various parts of his skeleton were pilfered along the way, he was reburied twice in Paris but it is highly likely his remains were lost during the revolution.

More details (probably too many details) can be found at `http://www.petercorke.com/descartes.pdf`.

Q8 We will create a class `QuatVec` to represent an $SE(3)$ transformation in terms of a quaternion (for rotation) and a vector (for translation). We start by defining the class

```
classdef QuatVec
```

and we define two properties

```
  properties
    q        % quaternion part
    t        % vector part
  end
```

that represent rotation (as a Quaternion) and translation (as a vector). and then start the methods part of the class

```
methods
```

The first method we will create is the constructor which creates an instance of a new `QuatVec` class. We will consider several calling cases: for no arguments, a single argument which is a homogeneous transform, and two arguments which are a `Quaternion` and a 3-vector.

```
function qv = QuatVec(a1, a2)
    if nargin == 0
        % QuatVec()
        qv.t = zeros(3,1);
        qv.q = Quaternion();
    elseif nargin == 1
        % QuatVec(T)
        qv.t = transl(a1);
        qv.q = Quaternion(a1);
    elseif nargin == 2
        % QuatVec(q, t)
        qv.q = a1;
        qv.t = a2;
    end
end
```

For the homogeneous transform case we set the vector and quaternion parts of the representation from the translational and rotational elements of the homogeneous transform. Note the `Quaternion` constructor accepts a homogeneous transform argument but considers only the rotation matrix component and returns the equivalent quaternion. Other cases can be easily created and the `Quaternion` constructor is a good example of how to do this. Note also that this example is missing logic to check that the passed arguments are of the correct type.

Multiplication of two `QuatVec` objects is implemented by the overloaded multiplication operator as per the definition on page 37 of RVC

```
function qv = mtimes(qv1, qv2)
    % qv1 * qv2
    t = qv1.t + qv1.q * qv2.t;
    q = qv1.q * qv2.q;
    qv = QuatVec(q, t);
end
```

Inversion of a `QuatVec` object is implemented by the overloaded `inv` function as per the definition on page 37 of RVC

```
function qi = inv(qv)
    % inv(qv)
    t = -(qv.q.inv() * qv.t);
    q = qv.q.inv();
    qi = QuatVec(q, t);
end
```

Finally we define a method to convert a `QuatVec` object to a homogeneous transform

```
function TT = T(qv)
    R = qv.q.R;
    t = qv.t;
    TT = rt2tr(R,t);
end
```

For completeness we will add two methods to display the value of a `QuatVec` object

```
    function s = char(qv)
        s = sprintf( '[%f, %f, %f], %s', qv.t, qv.q.char() );
    end

    function display(q)
       disp( qv.char() );
    end
```

The first method generates a string representation of the `QuatVec` object and the second is invoked whenever the value is to be displayed. Note that the Toolbox `display` functions have a bit of extra logic to handle the compact and long display formats.

Finally we need to close the method block and our class definition

```
    end  % methods
  end % classdef
```

To test our implementation

```
  >> T = transl(1,2,3) * troty(0.3) * trotz(0.2)
  >> qv = QuatVec(T)
  1.000000, 2.000000, 3.000000], 0.98383 < 0.014919, 0.14869, 0.098712 >
  >> qv*qv
  [2.443262, 4.158802, 5.693802], 0.93585 < 0.029355, 0.29257, 0.19423 >
  >> qi=qv.inv()
  [-0.464744, -1.946469, -3.161530], 0.98383 < -0.014919, -0.14869, -0.098712 >
  >> qi*qv
  [0.000000, 0.000000, 0.000000], 1 < 0, 0, 0 >
```

which shows composition, inversion and composition with the inverse which results in a null transformation. The code is provided as `QuatVec.m`.

Chapter 3

Time and motion

Q1 The simple case with no output arguments

```
>> tpoly(0, 1, 50)
```

plots the trajectory and shows nice smooth motion from 0 to 1 with initial and final velocity of zero. However a high initial velocity (fourth argument)

```
>> tpoly(0, 1, 50, 1, 0)
```

leads to significant overshoot. Time is required to decelerate during which the trajectory moves well past the destination and has to "back track".

Conversely a large final velocity

```
>> tpoly(0, 1, 50, 0, 1)
```

requires the trajectory to move initially in the negative direction so that it has sufficient distance over which to accelerate to the desired final velocity.

Q2 The simple case

```
>> lspb(0,1,50)
```

plots the trajectory and shows nice smooth motion from 0 to 1 with initial and final velocity of zero, see Figure 3.1(a). We see that the constant velocity is approximately 0.03, and that the acceleration, coast and deceleration phases of motion are of about equal length — the three motion phases are shown by different colored circles: red, green and blue for acceleration, coast and deceleration respectively.

The velocity of the coast phase can be specified by providing a fourth argument. By trial and error the lowest coast velocity that can be achieved, without the function reporting an error, is

```
>> lspb(0,1,50, 0.021)
```

in which case the motion is almost completely at constant velocity, see Figure 3.1(b), with very short acceleration and deceleration phases.

Again by trial and error, the highest coast velocity that can be achieved is

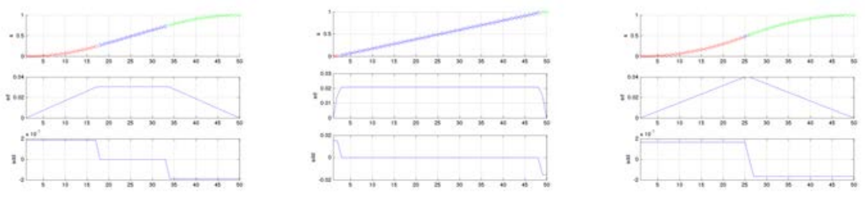

(a) (b) (c)

Figure 3.1: `lspb` trajectories from Q2.

```
>> lspb(0,1,50, 0.04)
```

in which case the coast time is very short, see Figure 3.1(c), and with the motion dominated by acceleration and deceleration time.

Q3 Consider the LSPB trajectory first. We use trial and error where we hold the coast velocity constant and adjust the third argument, the number of time steps, until the function can compute a valid trajectory. The result

```
>> lspb(0,1, 41, 0.025)
```

takes 41 time steps to complete.

For the polynomial trajectory we need to test the maximum velocity, which is one of the output variables, as we adjust the number of time steps. We find the quickest trajectory is

```
[s,sd,sdd]=tpoly(0,1,76); max(sd)
ans =
    0.0250
```

The polynomial is therefore

```
>> 76/41
ans =
    1.8537
```

times slower than the LSPB trajectory — the reason being that LSPB spends more time at a higher speed.

Q4a First we will create two orientations represented by orthonormal rotation matrices

```
>> R1 = eye(3,3);
>> R2 = rotx(2.2)*rotz(1.3)*roty(1.4);
```

Firstly, a quaternion interpolation is achieved by converting the rotation matrices to quaternions

```
>> q1 = Quaternion(R1);
>> q2 = Quaternion(R2);
```

and then interpolating the quaternions using an LSPB speed profile which results in a vector of `Quaternion` objects

```
>> s = lspb(0, 1, 50);
>> q = q1.interp(q2, s);
```

and displaying the sequence of rotations graphically

```
>> tranimate(q)
```

For Euler angles we first convert the rotations to Euler angle values

```
>> e1 = tr2eul(R1);
>> e2 = tr2eul(R2);
```

then interpolate these values using an LSPB speed profile

```
>> eul = mtraj(@lspb, e1, e2, 50);
```

and then convert back to a sequence of rotation matrices (represented by a matrix with the three dimensions where the last index represents the position along the sequence)

```
>> R = eul2r(eul);
>> tranimate(R)
```

For roll-pitch-yaw angles we first convert the rotations to RPY angle values

```
>> rpy1 = tr2rpy(R1);
>> rpy2 = tr2rpy(R2);
```

then interpolate these values using an LSPB speed profile

```
>> rpy = mtraj(@lspb, rpy1, rpy2, 50);
```

and then convert back to rotation matrices

```
>> R = rpy2r(rpy);
>> tranimate(R)
```

Q4b As discussed on page 32 of RVC there are singularities for ZYZ-Euler angles when $\theta = k\pi, k \in \mathbb{Z}$ and for roll-pitch-yaw angles when pitch $\theta = \pm(2k+1)\pi/2$. Illustrating this for the RPY angle case

```
>> R2 = rpy2r(1, pi/2, 1.5);
>> rpy2 = tr2rpy(R2);
>> rpy_4 = mtraj(@lspb, rpy1, rpy2, 50);
>> R = rpy2r(rpy_4);
>> tranimate(R)
```

and we observe that the motion is smooth.

Q5 We now consider a case where the path passes through the singularity, we engineer the start and end roll-pitch-yaw angles so that θ passes through the singularity at $\theta = \pi/2$.

```
>> rpy_5 = mtraj(@lspb, [0 pi/4 0], [1 3*pi/4 1.5], 51);
```

now we can convert these to rotation matrices

```
>> R= rpy2tr(rpy_5);
```

and if we animate this

```
>> tranimate(R)
```

and we observe that the motion is smooth. However if we convert the rotations back to roll-pitch-yaw angles

```
>> rpy_5b = tr2rpy(R);
>> plot(rpy_5b)
```

we will notice that the resulting roll-pitch-yaw angle trajectory is very different and has a discontinuity around time step 26.

Q6 We will use the trajectory variables `q`, `rpy` and `eul` from **Q3**. We will plot the locus followed by the x-axis of the rotated coordinate frame, which is the first column of each rotation matrix in the sequence, as it moves over the surface of the unit sphere.

Using the RPY data from the previous solution

```
>> x = R(:,1,:);
>> about x
x [double] : 3x1x51 (1224 bytes)
```

which we see has a middle dimension of 1 (which MATLAB calls a singleton dimension). We actually want a 3×51 matrix so we need to *squeeze out* the singleton

```
>> x = squeeze(x);
>> about x
x [double] : 3x51 (1224 bytes)
```

Next we generate a nice looking sphere

```
>> plot_sphere([], 1, colorname('skyblue'), 'alpha', 0.8, 'mesh', 'k');
```

which is a sky blue color (note we use the MVTB function `colorname` to get the RGB values of this color), it slightly translucent, and the mesh lines are visible (they act like lines of lattitude and longitude on a globe).

Now we overlay the path taken by the x-axis of the coordinate frame

```
>> hold on
>> plot2(x', 'w')
```

The function `plot2` is a Toolbox helper function that plots a line comprising points given by rows of the argument, in this case a 3D line since the argument has the columns.

Now for the Euler angle case we use the same logic

```
>> R = eul2r(eul);
>> x = R(:,1,:); x = squeeze(x);
>> plot2(x', 'w')
```

and finally for the quaternion case

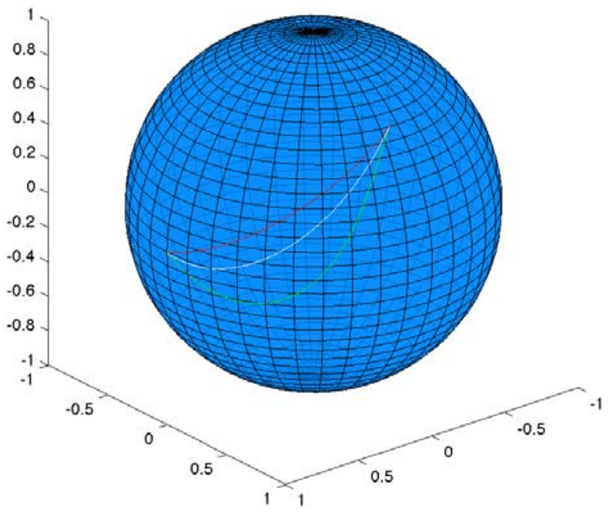


Figure 3.2: Angle interpolation trajectories from Q2. RPY (white), Euler angle (green) and quaternion (red).

```
>> R = q.R;
>> x = R(:,1,:); x = squeeze(x);
>> plot2(x', 'r')
```

The result is shown in Figure 3.2 and we can see that each approach to angle interpolation has given a different trajectory. The quaternion interpolation is the most direct and follows a great circle on the sphere, the most efficient way to get from A to B on a sphere. The difference in paths depends strongly on the start and end points. For small changes in attitude the results will be very similar but for large changes the results can be quite different.

Q7a The example from page 47 of RVC is

```
>> via = [ 4,1; 4,4; 5,2; 2,5 ];
>> q = mstraj(via, [2,1], [], [4,1], 0.05, 0);
```

which has a zero acceleration time. In order to achieve a non-zero initial or final velocity we must set a finite acceleration time so that a blending polynomial will be used to match the boundary conditions. Since there are two axes of motion in this example both initial and final velocity are 2-vectors, so to set a non-zero initial velocity we use

```
>> mstraj(via, [2,1], [], [4,1], 0.05, 1, [1 1], [0 0]);
```

and we observe a small initial wiggle in the motion as it smoothly moves from the given initial velocity to the velocity required to hit the first via point. Similarly a non-zero final velocity

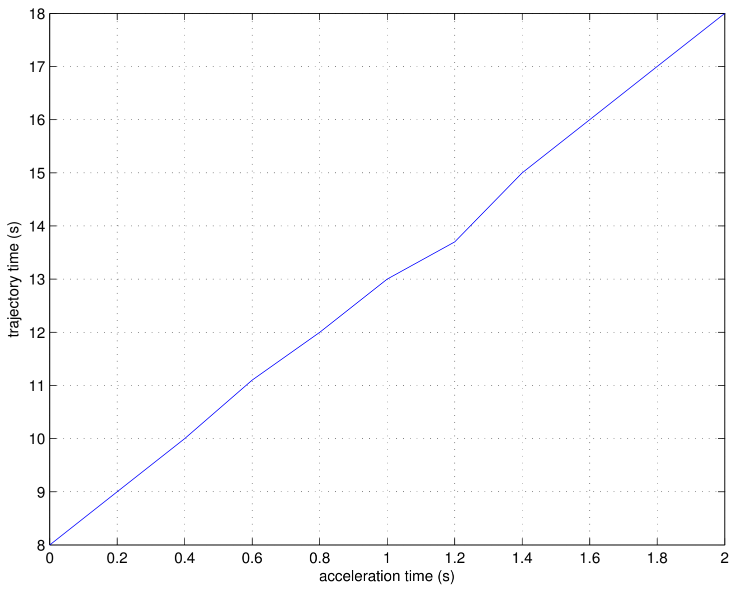


Figure 3.3: Trajectory time as a function of acceleration time.

```
>> mstraj(via, [2,1], [], [4,1], 0.05, 1, [0 0], [1 1]);
```

and we observe a small final wiggle in the motion.

Q7b We will investigate the effect as acceleration time varies from 0 to 2 seconds in steps of 0.2 seconds

```
>> t_acc = [0:0.2:2];
>> t_total = [];
>> for ta = t_acc
>>   traj = mstraj(via, [2,1], [], [4,1], 0.05, ta);
>>   t_total = [t_total numrows(traj)*0.05];
>> end
```

Note the for loop which exploits a useful MATLAB feature, the right-hand side is a matrix and the left-hand side is assigned to consecutive columns. Note also that the time step 0.05 is used to scale the trajectory length from time steps to seconds. The result is a vector `t_total` of the total trajectory time which we plot

```
>> plot(t_acc, t_total)
```

and the result is shown in Fig 3.3. This indicates a linear relationship with a minimum motion time of 8 seconds and a slope of nearly 5. Every second of acceleration time increases total motion time by 5 seconds. This is because there are 5 points in the path, the initial coordinate, and the four via points (the last being the destination).

Q9a We create the angular velocity signal as suggested

```
>> t = [0:0.01:10]';
>> w = [0.1*sin(t) 0.2*sin(0.6*t) 0.3*sin(0.4*t)];
```

which simulates what we would measure from a tri-axial gyroscope mounted on the robot. We start by assuming that the initial orientation of the robot is null

```
>> R = eye(3,3);
```

and we create a variable to hold the sequence of estimated attitudes

```
>> R_hist = [];
```

and also compute the sample interval

```
>> dt = t(2) - t(1);
```

Now for every time step we compute (3.13), normalise the estimated rotation to ensure that $\boldsymbol{R} \in SE(3)$ and add it to the sequence

```
>> for i=1:numrows(w)
>>  R = dt * skew(w(i,:)) *R + R;
>>  R = trnorm(R);
>>  R_hist = cat(3, R_hist, R);
>>  end
```

and finally we plot the estimate roll-pitch-yaw angles for the robot

```
>> plot( tr2rpy(R_hist) )
```

Q9b Using quaternions the implementation would look like

```
>> q = Quaternion();
>> q_hist = [];
>> dt = t(2) - t(1);
>> for i=1:numrows(w)
>>   q = dt * q.dot(w(i,:)) + q;
>>   q = q.unit();
>>   q_hist = cat(3, q_hist, q);
>> end
>> plot( tr2rpy(q_hist.R) )
```

Q9c We can add Gaussian noise to the original angular velocity measurements

```
>> w0 = w;
>> w = w0 + randn(size(w0)) * 0.001;
```

and in this case the noise is zero mean and has a standard deviation of 0.001. We will first save the result of the noise free case

```
>> q_hist0 = q_hist;
```

and then repeat the quaternion based integration from above

```
>> q = Quaternion();
>> q_hist = [];
>> dt = t(2) - t(1);
>> for i=1:numrows(w)
>>  q = dt * q.dot(w(i,:)) + q;
>>  q = q.unit();
>>  q_hist = cat(3, q_hist, q);
>>  end
>> plot( tr2rpy(q_hist.R) )
```

and comparing with the original solution

```
>> hold on
>> plot( tr2rpy(q_hist0.R), '--' )
```

we see no perceptible difference.

Q9d For the case where the standard deviation is 0.01 there is also no perceptible difference, but there is for the case where the standard deviation is 0.1. Explore the noise level at which the difference becomes significant, which might involve coming up with a workable definition of *significant*.

Part II

Mobile Robots

Chapter 4

Mobile robot vehicles

Q1a From page 68 of RVC we have

$$R = \frac{L}{\tan\gamma}$$

where R is the turn radius, L the wheel base and γ the steered wheel angle. The radii of the curves followed by the two back wheels are

$$R_{1L} = R - W/2$$
$$R_{1R} = R + W/2$$

The problem is actually quite complex but to a first approximation we will compute, as per Fig 4.2, R_2 for the inner and outer wheels

$$R_{2L} = \frac{R_{1L}}{\tan\gamma}$$
$$R_{2R} = \frac{R_{1R}}{\tan\gamma}$$

```
>> R = [10 50 100];
>> L = 2; W = 1.5;
>> R1L = R-W/2; R2L = R+W/2;
>> R2L = ( atan(L ./ R1L) - atan(L ./ R1R) ) *180/pi
ans =
    1.6613    0.0687    0.0172
```

in units of degrees. For a tight turn we see that the front wheels are 1.6 deg off parallel. Note that we've made use of MATLAB code vectorization here to compute the result in one hit for various values of `R`. Since `R` is a vector and `L` is a scalar we must use the element-by-element division operator `./`. In fact the two front wheels will not exactly lie on the radial line through ICR and this properly needs to be taken into consideration (exercise for the reader).

Q1b We will consider the back wheels and assume the wheel radius is $R_w = 0.3$. The left- and right-hand wheels have respective path radii of

$$R_{1L} = R - W/2$$
$$R_{1R} = R + W/2$$

Vehicle speed V is measured at the origin of the frame $\{V\}$ located between the back wheels. We can apply equation (4.1) to both back wheels

$$\dot{\theta} = \frac{V}{R} = \frac{V_{1L}}{R_{1L}} = \frac{V_{1R}}{R_{1R}}$$

and the wheel velocity is related to wheel angular velocity by $V_i = \omega_i R_{w_i}$ so we can write

$$\frac{V}{R} = \frac{\omega_{1L} R_w}{R_{1L}} = \frac{\omega_{1R} R_w}{R_{1R}}$$

In MATLAB this is simply

```
>> R = [10 50 100];
>> L = 2; W = 1.5; Rw = 0.3;
>> V = 80 * (1000/3600)   % m/s
V =
   22.2222
>> R1L = R-W/2; R1R = R+W/2;
>> omega1L = V ./ R .* R1L/Rw
omega1L =
   68.5185   72.9630   73.5185

>> omega1R = V ./ R .* R1R/Rw
omega1R =
   79.6296   75.1852   74.6296

>> omega1R./omega1L
ans =
    1.1622    1.0305    1.0151
```

So for the tight turn (10 m) radius the outside wheel is rotating 16% faster than the inside wheel. That's why a car's motor is not connected directly to the wheels but rather via a differential gearbox which allows the wheels to rotate at different speeds.

Q2 The rotation rates of the back wheels are each a function of the turn rate $\dot{\theta}$

$$\dot{\theta}_L = \frac{\omega_{1L} R_w}{R_{1L}}$$
$$\dot{\theta}_R = \frac{\omega_{1R} R_w}{R_{1R}}$$

so a reasonable strategy would be to take the mean of the turn rate estimate derived from each wheel.

Q4 We first set the initial pose of the vehicle (x, y, θ)

```
>> x0 = [8 5 pi/2];
```

and the final position (x, y)

```
>> xg = [5 5];
```

We then get Simulink to run the simulation by

```
>> r = sim('sl_drivepoint');
```

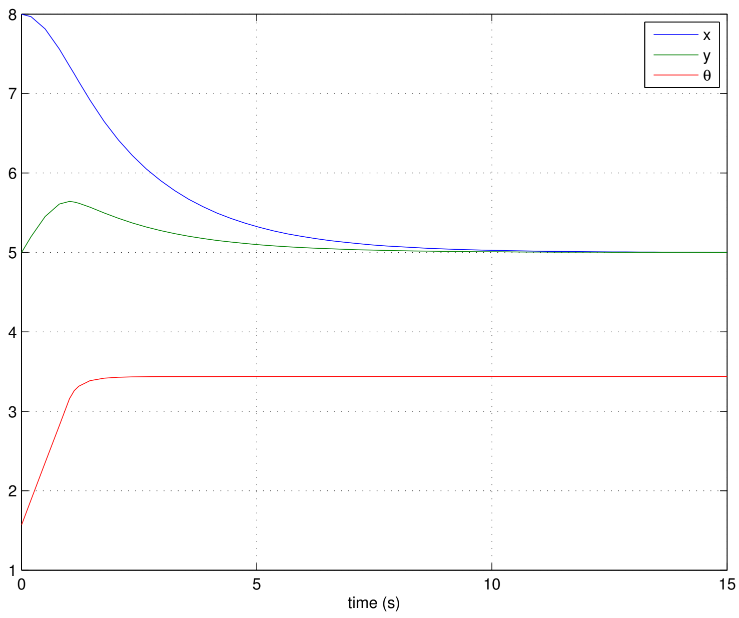


Figure 4.1: Simulation results from `sl_drivepoint` (Q4).

and the results are all stored with the variable `r` which is an object of type `Simulink.SimulationOutput`. If we display it it shows the simulation output records that it contains

```
>> r
Simulink.SimulationOutput:
    tout: [56x1 double]
    yout: [56x3 double]
```

in this case a vector of times (`tout`) and a vector of simulation outputs (`yout`) where each row corresponds to the time at the corresponding row of `tout`. Each row has three values which are the values at the output connectors 1 (a 2-vector containing x and y) and 2 (scalar θ) shown in Figure 4.6 of RVC. We extract these signals from the `Simulink.SimulationOutput` object

```
>> t = r.find('tout');
>> q = r.find('yout');
```

and then plot the result

```
>> plot(t, q)
```

which is shown in Figure 4.1. We see that the heading angle θ converges very quickly and the position more slowly. Reducing the heading gain K_h would reduce the rate at which θ converges. Conversely increasing the other Gain term will increase the rate at which the position converges.

A Toolbox convenience function `plot` can perform this more concisely

```
>> mplot(r, 'label', 'x', 'y', '\theta')
```

Q5a We load the model

```
>> sl_pursuit
```

and start the simulation from the Simulink Simulation menu which gives the result shown in Figure 4.11 of RVC. After an initial catchup period the vehicle follows the pursuit target with zero error. Reducing the integral gain to zero results in a steady state velocity error — the vehicle maintains a constant distance offset and travels at slightly lower velocity than the pursuit point — the vehicle is moving on a smaller radius circle.

Reducing the speed of the pursuit point around the circle highlights some interesting aspects. Firstly you need to increase the simulation period from the Simulink `Simulation/Configuration Parameters` menu, to say 200 seconds in order to see the motion over multiple revolutions. Now the following error is less — it is proportional to the velocity of the goal point.

Reinstating the integral gain now results in severe oscillation, an interaction between the integral action and system non linearities. For a non-linear system the appropriate gain settings will depend on the operating regime. Note that the Bicycle model can be configured with strong saturation non-linearities. The maximum velocity, maximum acceleration and steering angles can all be set to finite values, see the block parameters for details and also "look under the mask" to see how the model works.

Q5b The steady state velocity for the motion is $r\omega$ and $\omega = 2\pi f$ and f is the rotational frequency is revolutions per second, in this case $f = 1$. So a constant of 2π could be added to the output of the error block. Without integral action good tracking performance is observed.

Q5c To create a slalom course in the x-axis direction we could set the x-coordinate to be a linear ramp obtained from a Simulink Sources/Ramp block. The y-coordinate would be a triangular wave but there is no block that generates this directly. The closest is the Sources/RepeatingSequence block where we can set the Time Values to `[0 1 2 3 4]` and the Output Values to `[0 1 0 -1 0]`.

Q6a We can set a different initial configuration

```
>> xg = [5 5 pi/2]
>> x0 = [7 7 0]
```

and start the simulation

```
>> r = sim('sl_drivepose');
```

and we see the vehicle moves as expected.

Q6b For parallel parking we could consider the goal pose as $(5,5,0)$ and the initial pose is displaced laterally with respect to the final heading direction, perhaps $(0,9,0)$. The robot is able to park by following an s-shaped path, it just drives straight in — something we generally cannot do in real life. As the initial lateral displacement is reduced the vehicle motion is constrained by its steering angle limits and turning radius — for an initial pose of $(0,7,0)$ it is unable to converge.

Normally when we parallel park we move forwards and then backwards. This controller is only capable of moving in one direction which is determined at the outset. A better approach is path planner as discussed in Section 5.2.5 of RVC.

Q7 The GUI for MATLAB is somewhat limited in primitives. You could use a pair of sliders, one for velocity and one for steering angle. The Simulink library “Simulink 3D animation utilities” includes a joystick input block with outputs that reflect the various joystick axes as well as buttons and switches. It supports a range of standard USB output gaming joysticks and human interface devices.

Q8a We load the model

```
>> sl_quadcopter
```

To set the damping gains to zero change the block D_pr to zero. We see that the vehicle oscillates in roll and pitch before becoming completely unstable.

Q8b If we set the feedforward term in block w0 to zero we observe the quadrotor doesn’t leave the ground. The vehicle motion data is saved in the workspace variable `result` which has columns: $t, x, y, z, \theta_y, \theta_p, \theta_r, \cdots$ so to plot altitude is

```
>> plot(result(:,1), result(:,4))
```

and verifies that it does not really lift off. Remember that we are using a coordinate convention with the z-axis downward so when he vehicle is flying it will have a negative z-value.

If we increase the loop gain `P_z` to eight times its original value, to -320, then the vehicle lifts off and achieves an altitude of about -1.4 m but this is substantially less than the desired value of -4 m.

If we add a Continuous/PID controller block from the Simulink menu, and set it up for PID Controller in Ideal Form, then gains of $P = -40$, $I = 1$ and $D = 1$ provide acceptable height control.

Q8c The thrust of the i’th rotor is given by (4.3) as

$$T_i = b\omega_i^2$$

and for four rotors all operating at a nominal angular velocity ω_0 the thrust would be

$$T = 4b\omega_0^2$$

In vertical equilibrium $T = mg$ so we can write

$$mg = 4b\omega_0^2$$

and solve for ω_0.

Q8d The orientation of the quadrotor, with respect to the world frame, is given by

$$\boldsymbol{R} = \boldsymbol{R}_z(\theta_y)\boldsymbol{R}_y(\theta_p)\boldsymbol{R}_x(\theta_r)$$

and we can expand this algebraically. The third column is the orientation of the vehicle’s z-axis in the world frame which is parallel to the thrust vector. The z-component of this vector

$$\cos\theta_p \cos\theta_r$$

is the component of the thrust acting in the world frame vertical direction, that is, the fraction that contributes to lift rather than lateral acceleration. To compensate for the loss of thrust we should increase the overall thrust by the inverse of this amount

$$T' = \frac{T}{\cos\theta_r \cos\theta_p}$$

Chapter 5

Navigation

Q1a The initial position of the vehicle is set by parameters of the Bicycle block. Change them from `[5, 5, 0]` to `[50, 50, -pi/3]` and see the different path followed. Now reduce the Gain block from 5 to 4 and we see that the vehicle turns much more slowly. Increase it to 6 and the vehicle turns much more quickly.

To display the sensor field, like Fig 5.3, we first compute its value at regular points in the robot's workspace. Rather than use nested `for` loops we will instead we will demonstrate some of the vectorization features of MATLAB. We first create the coordinate matrices `meshgrid`

```
>> [X,Y] = meshgrid(1:100, 1:100);
```

where `X` is a 100×100 matrix and element $X(i, j) = j$. Similarly `X` is a 100×100 matrix and element $Y(i, j) = i$. To compute the function

$$\frac{200}{(x-x_c)^2+(y-y_c)^2+200}$$

we can use vectorization to compute the function at all points by writing it as

```
function sensor = sensorfield(x, y)
    xc = 60; yc = 90;
    sensor = 200./((x-xc).^2 + (y-yc).^2 + 200);
```

Note the use of the `./` and `.^` operators and understand why they are there. The function, written in this form, can accept either scalar or matrix arguments.

Now we can evaluate the sensor field at all points in the workspace

```
>> field = sensorfield(X, Y);
```

and display it by

```
>> image(field, 'CDataMapping', 'scaled')
>> set(gca, 'Ydir', 'normal')
>> colorbar
```

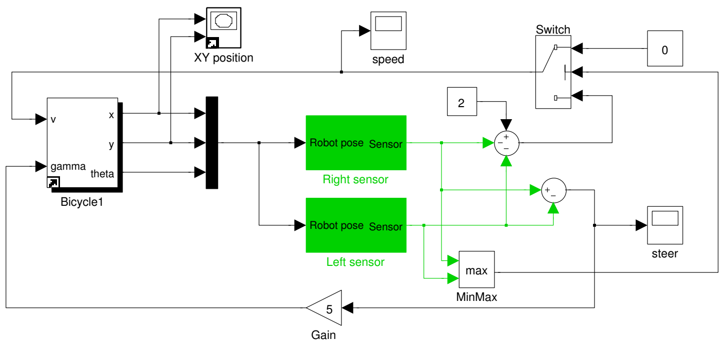


Figure 5.1: Braitenberg model with stopping logic (Q1d).

The `set` function is to ensure that the y-coordinate increases upward, for an image the convention is that the y-coordinate increases downward. Using the companion Machine Vision Toolbox for MATLAB this can be expressed more concisely as

```
>> idisp(field, 'ynormal', 'colormap', 'jet', 'bar')
```

The simulation places the results in the workspace, with time in the vector `tout` and the vehicle pose in the matrix `xout` with one row per time step and each row is x, y, θ. We can overlay this on the sensor field by

```
>> hold on
>> plot(xout(:,1), xout(:,2), 'g')
```

Q1b Simply change the gain to -5 and the vehicle now runs away from the light. Try starting the vehicle pointing toward the light, set the initial state to `[50, 50, pi/3]` and it quickly turns and moves away.

Q1c The sensor models can be seen by right-clicking and choosing Look under mask. We append the time to the vector passing between the two Fcn blocks, using a Clock and Mux block. We modify the second Fcn block to pass time into the function. We modify the `sensorfield` function

```
function sensor = sensorfield(x, y, t)
    vx = 0.1; vy = 0.1;                % speed of target
    xc = 60 + t*vx; yc = 90 - t*vy;
    sensor = 200./((x-xc).^2 + (y-yc).^2 + 200);
```

to accept the time argument and compute `xc` and `yc` as functions of time. The modified model is provided as `sl_braitenberg2`. Settting Gain to 5 and the initial state to `[5, 5, 0]` we see that the vehicle moves toward the goal and then turn sharply in order to follow it along its linear path.

Q1d The model `sl_braitenberg3`, shown in Figure 5.1, includes the stopping logic. We take the maximum of the two sensor outputs and use that to control a switch which selects the vehicle velocity from the sensors when the maxima is high, and to zero when the maxima is small. The threshold on the maximum value is a parameter of the Switch block.

Q1e To create a sensor function with two peaks requires only a simple modification of the existing `sensorfield` function

```
function sensor = sensorfield4(x, y)

    % peak 1
    xc = 30; yc = 90;
    sensor1 = 200./((x-xc).^2 + (y-yc).^2 + 200);

    % peak 2
    xc = 70; yc = 90;
    sensor2 = 200./((x-xc).^2 + (y-yc).^2 + 200);

    sensor = sensor1 + sensor2;
```

creates another peak and we add the two contributions together. The peaks are centred at $(30,90)$ and $(70,90)$ respectively.

Repeat the code from Q1a to show the two peaks

```
>> [X,Y] = meshgrid(1:100, 1:100);
>> field = sensorfield4(X, Y);
>> image(field, 'CDataMapping', 'scaled')
>> set(gca, 'Ydir', 'normal')
>> colorbar
```

With an initial state half way between the two peaks `[50, 90, 0]` the vehicle moves to the peak $(70,90)$ it is pointing toward.

If we initialize the vehicle so its heading is orthogonal to the line between the peaks, say `[50, 90, -pi/2]` then it moves downward in a straight line and never turns toward either peak. Experiment with adding some random noise to the steering signal, using a Random Number and Sum block, and see if the vehicle can be knocked into the attractive region of one or other peak.

Q2a The `makemap` function can be invoked easily from the command line

```
>> map = makemap(100)
```

to create a map in a 100×100 occupancy grid. Use interactive commands as described by

```
>> help makemap
```

to add different primitives. Note that the keyboard characters to invoke different primitives must be typed with the mouse over the figure not the command window.

The starting point for the simulation is set by the argument to the `path` method of the `Bug2` object

```
>> bug.path( [x, y] )
```

The bug automaton will always find a path if one exists. Create a map with a prison, four rectangular blocks that form walls with no gaps. Place the robot goal outside and the robot inside, or vice versa, and run the simulation. After a complete circuit of the prison cell, or the prison, the simulation will stop and report that the robot is trapped.

Q2b There are lots of web resources for creating mazes, and even some MATLAB functions on MATLAB Central. In this case we created a maze using the web site

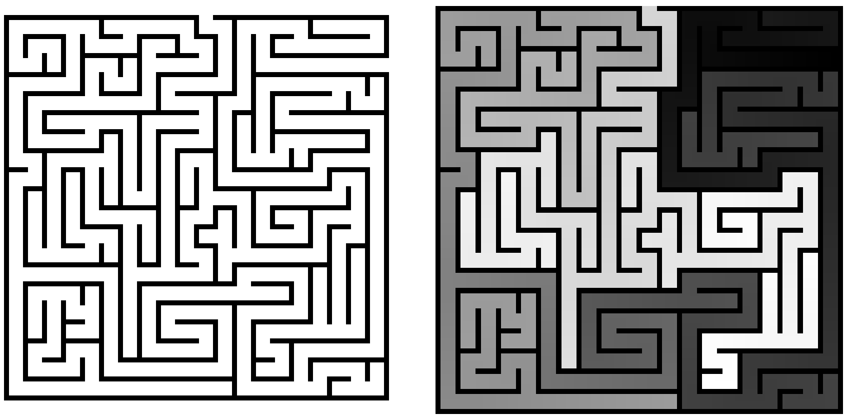

(a) (b)

Figure 5.2: Maze and its distance transform (Q2b).

`http://hereandabove.com/maze` and saved the resulting maze as an image `maze.png` which has pixel values either 0 (free) or 1 (obstacle).

We loaded the maze image and converted it to a double precision matrix with values of 0 or 1.

```
>> z=iread('maze.gif');
>> map=idouble(z>0);
```

The maze has free space around it and the robot will sneak around the edge rather than go through the maze, so we need to trim this off

```
>> map=map(7:end-6,7:end-6);
```

We display the map, see Figure 5.2(a), and pick the coordinates of the start (top left) and goal (bottom right)

```
>> idisp(map, 'normal')
```

Then create the robot and set it's goal

```
>> bug=Bug2(map);
>> bug.goal = [162,23];
```

Then simulate the motion

```
>> bug.path( [2,86] );
```

This takes a long time to run and traverses every wall in the maze. Can you do it more quickly and with a shorter path?

One option is to compute the distance transform of the map.

```
>> d = distancexform(map);
>> idisp(d)
```

and this is shown in Figure 5.2(b). Start at the goal, the darkest pixels and work back through increasing brightness toward the start. Note the very bright region in the centre left of the maze which are the points furthest from the goal, but in a deadend region that is not on the path to between start and goal.

Q2c Details for alternative bug algorithms such as bug1 and tangent are given in Lumelsky and Stepanov (1986).

Q3 The Earth is approximately a sphere with a radius of 6400 km so its surface area is

$$4\pi R^2$$

which gives

```
>> 4*pi*6400e3^2
ans =
   5.1472e+14
```

in units of square metres. If each square metre is represented by a single-bit (0=free, 1=obstacle) then this requires

```
>> ans/8
ans =
   6.4340e+13
```

bytes, or

```
>> ans/(1024)^4
ans =
   58.5167
```

which is 58 Tbyte, which is a lot more than available on a commodity computer (in 2012).

According to WikiPedia the land area is 148,940,000 km^2 so repeating the above calculation we get

```
>> land_area = 148940000*1e6
land_area =
   1.4894e+14
```

in units of square meters, and the number of bytes required is

```
>> land_area/8
ans =
   1.8618e+13
>> ans/(1024)^4
ans =
   16.9325
```

or 17 Tbyte, still quite a lot.

The inverse calcuation is that we start with 1 Gbyte of memory (a very modest amount of storage) or

```
>> (1024)^3*8
ans =
   8.5899e+09
```

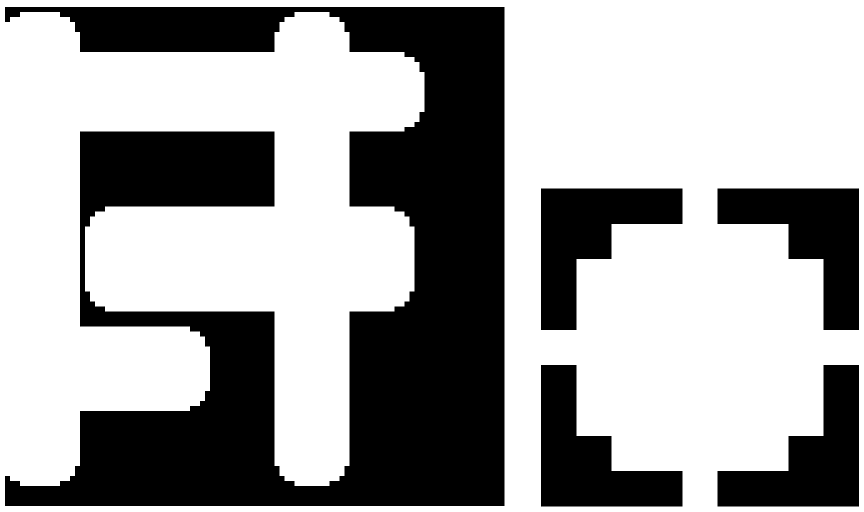


Figure 5.3: Map after dilation, and the dilation kernel (Q4).

bits. The area per bit is

```
>> land_area/ans
ans =
   1.7339e+04
```

square metres, which is a patch

```
>> sqrt(ans)
ans =
  131.6772
```

metres on a side. This would be too coarse to represent useful things like roads which can be only 5 m across.

Discuss some ways to solve this problem. Some ideas include compression of the bit map, or only mapping the country or city of interest.

Q4 Consider that the map is in the variable `map`, then the dilated obstacle is

```
>> map=idilate(map, kcircle(4));
```

which is shown in Figure 5.3(a). The function `kcircle` creates a kernel, a matrix in this case 9×9, that contains a centred circular disc of ones surround by zeros, shown in Figure 5.3(b). Such a kernel will add 4 pixels to the edge of all obstacles. Note that these functions require the Machine Vision Toolbox for MATLAB to be installed.

Q5 The D* example on page 95 of RVC starts by loading a map and the setting the start and goal positions

```
>> load map1
>> goal = [50; 30];
>> start = [20; 10];
```

We then create a D* planner

```
>> ds = Dstar(map);
>> ds.plan(goal);
```

The example on page 96 shows how to increase the cost of a region that the vehicle moves through.

```
>> for y=78:85
>>    for x=12:45
>>        ds.modify_cost([x,y], 2);
>>    end
>> end
```

If we create a region of very low cost cells (zero, as easy as it gets)

```
>> for y=85:95
>>   for x=20:50
>>     ds.modify_cost([x,y], 0);
>>   end
>> end
```

and then replan

```
>> ds.plan()
```

and then compute the path

```
>> ds.path(start)
```

we see that the robot's path has been *pulled* into the zero cost region that it did not previously enter, and the robot travels as far as it usefully can in that region.

Q6a The following script, provided as `prmlength.m`, runs the PRM planner 100 times and records the path length for each iteration.

```
prm = PRM(map);

clear plength;
for i=1:100
    prm.plan();

    try
        % try to find a path
        p = prm.path(start, goal);
        % record the length
        plength(i) = numrows(p);

    catch
        % failed to find a path
        plength(i) = NaN;
    end
end
```

The `path` method can fail if there is not a continuous roadmap between the start and the goal, in which case an exception is thrown. We catch this and indicate a failure by recording the path length for this iteration as `NaN`.

The number of failures is

```
>> length(find(isnan(plength)))
ans =
    38
```

that is nearly 40% of the time there was no path from start to goal. Remember that due to the probabilistic nature of this planner this statistic will vary from run to run.

The maximum path length is

```
>> max(plength)
ans =
   314
```

the minimum is

```
>> min(plength)
ans =
   254
```

and the mean is

```
>> mean (plength( ~isnan(plength) ))
ans =
  281.0645
```

Note that the `min` and `max` functions exclude `NaN` values but `mean` does not — in any element is `NaN` the mean will be `NaN`. Therefore we need to remove the `NaN` values from the length vector prior to invoking `mean`.

Q6b Changing the distance threshold to 15 (the default is 30) in the script

```
prm = PRM(map, 'distthresh', 15);
```

results in an increased number of failures

```
>> length(find(isnan(plength)))
ans =
    98
```

Changing it to 50 greatly increases the computation time

```
>> length(find(isnan(plength)))
ans =
    15
```

but the number of failures is reduced. This is an important tradeoff in considering how to set the values of the parameters for this planner.

Q6c Consider the vehicle as a circle approximated by a polygon, and defined with respect to its own origin, which represents all possible orientations of the vehicle. For an arbitrary pose we can transform the polygon points using its `transform` method. Consider the world as a vector of `Polygon` objects. In the planning stage, for every random point chosen, we need to determine if the path is collision free. There are two subtests:

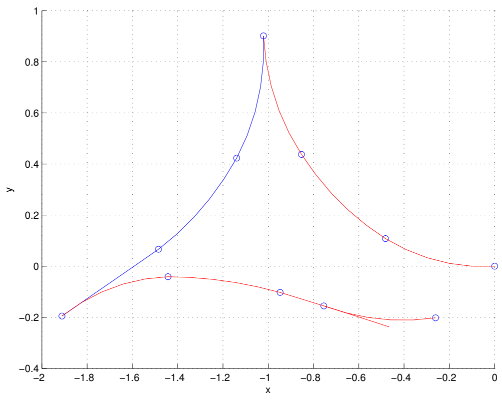


Figure 5.4: RRT for three-point turn, forward motion is shown in blue, backward motion in red. (Q7).

1. Determine if a path from the centre of the vehicle to the point is feasible, if not another point must be selected. The `intersect_line` method can test the intersection of a line segment against a list of obstacle polygons.
2. Test whether the vehicle can fit at every point along the line. Transform the vehicle polygon using it's `transform` method and then test for intersection against a list of obstacle polygons using the `intersection` method.

Q7a We will define the initial pose as $(0,0,0)$ and the final pose as $(0,0,\pi)$, that is the position is unchanged but the heading direction is opposite

```
>> x0 = [0 0 0];
>> xg = [0 0 pi];
```

We now create a vehicle kinematic model

```
>> veh = Vehicle([], 'stlim', 1.2);
```

and then compute the navigation tree. By default the RRT chooses 500 points which is too few to provide a good match to the final pose. Increasing the number of points to 1000 gets closer

```
>> rrt = RRT([], veh, 'goal', xg, 'range', 5, 'npoints', 1000)
rrt =
RRT navigation class:
  occupancy grid: 0x0
  goal: (50,30)
  region: X -5.000000 : 5.000000; Y -5.000000 : 5.000000
  path time: 0.500000
  graph size: 1000 nodes
  3 dimensions
  0 vertices
  0 edges
  0 components
Vehicle object
  L=1, maxspeed=5, alphalim=1.2, T=0.100000, nhist=0
  x=0, y=0, theta=0
```

```
>> rrt.plan
graph create done
```

Now we can plan a path

```
>> p=rrt.path(x0, xg);
```

and the resulting path, see Figure 5.4, shows a classical 3 point turn.

Q7d The steering angle is chosen in the private method `bestpath` at around line 400. Change the instance of `rrt.rand` to `rrt.randn`.

Chapter 6

Localization

Q1 The Longitude Prize was £20,000 and first offered in 1714. According to `http://www.measuringworth.com` the CPI growth index is the most appropriate in this case and gives a value of £2,250,000 or over USD 3M (in 2012) and is of the same order of magnitude as today's X Prize.

Q2 We can modify the existing class `RandomPath`

```
% private method, invoked from demand() to compute a new waypoint
function setgoal(driver)
    % choose random radius

    r = driver.randstream.rand(1,1) * driver.radius;
    th = driver.randstream.rand(1,1) * 2 * pi;

    driver.goal = r*[cos(th) sin(th)] + driver.centre;
end
```

and substitute this into a copy of RandomPath that we name CircularRegionPath.

Now let's test it by creating a vehicle

```
>> veh = Vehicle()
>> driver = CircularRegionPath([2,2], 5)
driver =
CircularRegionPath driver object
  current goal=?
  centre (2.0, 2.0), radius 5.0
>> veh.run()
```

Q3 See **Q2** for Chapter 4. **Q4a** We can look at the final covariance of the filter over the simulation time by

```
>> ekf.plot_P()
```

We see that the final covariance is almost invariant to the value of $\boldsymbol{P}_0$ chosen. If we make the $\hat{\boldsymbol{V}} > \boldsymbol{V}$ by

```
>> ekf = EKF(veh, 10*V, P0);
```

then the final covariance will be greatly increased. The filter believes the odometry is much less reliable than it really is, and thus the uncertainty of the estimated state will be much higher. Conversely if $\hat{\boldsymbol{V}} < \boldsymbol{V}$ by

```
>> ekf = EKF(veh, 0.1*V, P0);
```

then the final covariance will be greatly reduced. The filter believes the odometry is much more reliable than it really is, and thus the uncertainty of the estimated state will be much lower — in fact the filter is over confident of its estimates which is potentially dangerous.

Q4b Similar to Fig 6.4, but compare actual and estimated heading

```
>> est = ekf.plot_xy
>> true = veh.plot_xy
>> plot([angdiff(true(:,3)) angdiff(set
(:,3))])
```

Q4c The covariance associated with position and heading varies along the path. This can be seen clearly by

```
>> ekf.plot_error()
```

which shows the estimation error (actual from the vehicle simulation minus the EKF estimate) overlaid on the $\pm 3\sigma$ uncertainty bounds. For this question we could consider the final covariance which will be consistent if the vehicle always follows the same path, which the `RandomPath` driver object does (it resets its random number generator on every run). The covariance of the vehicle states can be found by

```
 >> Pv = diag( ekf.P_est )'
Pv =
    0.0050    0.0072    0.0002
```

Now we can create a table of final covariance for the cases where we vary the covariance of the sensor observations $\boldsymbol{W}$ by a command like

```
>> sensor.W = diag( [0.2 1*pi/180].^2);
```

and then rerunning the simulation.

Q5a We first need to consider what we mean by performance. The final covariance is a measure of the error in the vehicle's location and typically becomes close to zero quite quickly. A more meaningful measure is the distance between the estimated and true vehicle position

```
>> est = ekf.plot_xy();
>> true = veh.plot_xy();
>> dxy = true(:,1:2) - est(:,1:2);
>> dth = angdiff(true(:,3), est(:,3));
>> mplot([], [dxy dth], 'label', 'dx', 'dy', 'd\theta');
```

and for default setting a plot like that shown in Figure 6.1 is produced. Performance could be taken as the worst case given by the maximum of the absolute value or some kind of average as given by the root mean square

```
>> rms([dxy dth])
ans =
    0.0892    0.1170    0.0166
```

which is a compact representation. Remember that these values will depend on the map you use and the random path followed by the vehicle. The EKF class resets the random number generator in the RandomPath object for each simulation run in order to create consistent vehicle paths.

We can now make a table with filter performance described by three columns, RMS error in vehicle pose (x, y and θ), and we are now in a position to see the effect of parameter changes.

Sensor covariance can be changed by, for example, making the sensor more noisy than the filter estimate of the noise

```
>> sensor = RangeBearingSensor(veh, map, W*2);
>> ekf = EKF(veh, V, P0, sensor, W, map);
```

Alternatively we could make the filter's estimate of the covariance greater than the sensor's noise covariance

```
>> sensor = RangeBearingSensor(veh, map, W*2);
>> ekf = EKF(veh, V, P0, sensor, W, map);
```

We can change the sensor's sample rate by for example

```
>> sensor = RangeBearingSensor(veh, map, W, 'skip', 5);
```

so that it returns a range and bearing measurement every fifth time its method is called by the EKF.

We can also adjust the effective range of the sensor, for example

```
>> sensor = RangeBearingSensor(veh, map, W, 'range', 4);
```

limits the observation to landmarks that are within 4 m of the sensor. Alternatively we can limit returns to landmarks that are within ± 90 deg of the orientation of the sensor by

```
>> sensor = RangeBearingSensor(veh, map, W, 'angle', [-1 1]*pi/4);
```

If the parameters of the sensor are changed as above this creates a new instance of the sensor object, and a new instance of the filter needs to be created

```
>> ekf = EKF(veh, V, P0, sensor, W, map);
```

that is connected to the new sensor object.

Alternatively the properties of the sensor object can be adjusted without needing to create a new EKF filter object, for example

```
>> sensor.interval = 5;
>> sensor.r_range = [0 4];  % [minrange maxrange]
>> sensor.th_range = [-1 1]*pi/4; % [minrange maxrange]
```

Q5b $\boldsymbol{W}$ is the covariance of the sensor itself and is the 2×2 matrix value passed to the RangeBearingSensor constructor. $\hat{\boldsymbol{W}}$ is the estimate of the covariance of the sensor and is the 2×2 matrix value passed as the fifth argument to the EKF constructor. For the case where we make $\hat{\boldsymbol{W}} > \boldsymbol{W}$

```
>> sensor = RangeBearingSensor(veh, map, W*10);
```

the filter becomes over confident and if we look at the estimation error

```
>> ekf.plot_error()
```

we see that the error lies well outside the estimated confidence bounds.

Conversely if $\hat{\boldsymbol{W}} < \boldsymbol{W}$

```
>> sensor = RangeBearingSensor(veh, map, W*0.10);
```

the filter becomes more conservative and the errors lie more often within the confidence bounds.

Q5c A bearing-only sensor returns the direction of a landmark relative to the vehicle, but not its distance. We could create such a sensor model by copying and then modifying the existing RangeBearingSensor object. We need to change the name of the class and the constructor function and then modify the methods , , and .

Now the performance of this sensor can be compared to that of the range and bearing sensor.

Q5d The RangeBearingSensor allows for simulation of temporary sensor failure

```
>> sensor = RangeBearingSensor(veh, map, W, 'fail', [100 150]);
```

between time steps 100 and 150. To simulate more complex failures requires modifying the source code of the function reading() in RangeBearingSensor.m.

To simulate random sensor failure we could add

```
          if s.randstream.rand < 0.1
                    return
          end
```

before the line

```
              % simulated failure
```

With a probability of 10% it will fail to return a valid observation so the EKF will perform only the prediction step.

Just before the end of the function we could simulate more drastic failures such as

```
          if s.randstream.rand < 0.05
                    z = zeros(size(z));
          end
```

which returns a reading of zero with a probability of 5%. Alternatively we could simulate random wrong landmark identity by

```
        if s.randstream.rand < 0.05
                   jf = s.selectFeature();
        end
```

which, with a probability of 5%, returns a random feature id.

Q5e The relevant code is inside the EKF class and the method, just below the comment "process observations". The identity of the landmark is taken as the value reported by the sensor's method but now we wish to use the range and bearing data to determine which landmark we are observing. Since this is a SLAM problem we cannot assume that the location of the landmarks is known in advance. Instead we use the estimated vehicle pose and the sensor reading to estimate the location of the landmark and compare that to the estimated landmark positions in the state vector, taking into account the uncertainty in both robot pose and the landmark position. The essential task is to make a decision about whether the observed feature has been seen before or is a new feature. Techniques that are used include: nearest neighbour, individual compatibility, joint compatibility, multi-hypothesis data association and many variants and combinations thereof. A good summary is given in the thesis by Cooper at http://dspace.mit.edu/handle/1721.1/32438.

Q5f Fig 6.7, but compare actual and estimated heading

```
>> est = ekf.plot_xy
>> true = veh.plot_xy
>> plot([angdiff(true(:,3)) angdiff(set
(:,3))])
```

Q5g See **Q4c**.

Q6a Once again we need to first define our measure of performance. For map building it should relate to the error in estimating landmark locations. The Cartesian error between true and estimated location is given by

```
>> dmap = sqrt(colnorm( map.map - ekf.plot_map() ));
```

and we can look at some simple statistics such as

```
>> max(dmap)
ans =
    0.5803

>> mean(dmap)
ans =
    0.4197
```

We can now make a table with filter performance described by the mean landmark position error and we are now in a position to see the effect of parameter changes.

Sensor covariance can be changed by, for example, making the sensor more noisy than the filter estimate of the noise

```
>> sensor = RangeBearingSensor(veh, map, W*2);
>> ekf = EKF(veh, [], [], sensor, W, []);
```

Alternatively we could make the filter's estimate of the covariance greater than the sensor's noise covariance

```
>> sensor = RangeBearingSensor(veh, map, W*2);
>> ekf = EKF(veh, [], [], sensor, W, []);
```

We can change the sensor's sample rate by for example

```
>> sensor = RangeBearingSensor(veh, map, W, 'skip', 5);
```

so that it returns a range and bearing measurement every fifth time its method is called by the EKF.

We can also adjust the effective range of the sensor, for example

```
>> sensor = RangeBearingSensor(veh, map, W, 'range', 4);
```

limits the observation to landmarks that are within 4 m of the sensor. Alternatively we can limit returns to landmarks that are within ± 90 deg of the orientation of the sensor by

```
>> sensor = RangeBearingSensor(veh, map, W, 'angle', [-1 1]*pi/4);
```

We can also simulate a temporary failure of the sensor

```
>> sensor = RangeBearingSensor(veh, map, W, 'fail', [100 150]);
```

between time steps 100 and 150.

If the parameters of the sensor are changed as above this creates a new instance of the `sensor` object, and a new instance of the filter needs to be created

```
>> ekf = EKF(veh, [], [], sensor, W, []);
```

that is connected to the new `sensor` object.

Alternatively the properties of the sensor object can be adjusted without needing to create a new EKF filter object, for example

```
>> sensor.interval = 5;
>> sensor.r_range = [0 4];   % [minrange maxrange]
>> sensor.th_range = [-1 1]*pi/4; % [minrange maxrange]
```

Q6b See the approach from **Q5c**.

Q6c

Q7a For a SLAM system performance has two aspects: estimation of vehicle pose and estimation of the map landmark positions. These were discussed individually in the solutions for **Q5a** and **Q6a**. We can now make a table with filter performance described by four columns: RMS error in vehicle pose (x, y and θ) and mean landmark position error. Now we are in a position to see the effect of parameter changes.

Sensor covariance can be changed by, for example, making the sensor more noisy than the filter estimate of the noise

```
>> sensor = RangeBearingSensor(veh, map, W*2);
>> ekf = EKF(veh, V, P0, sensor, W, []);
```

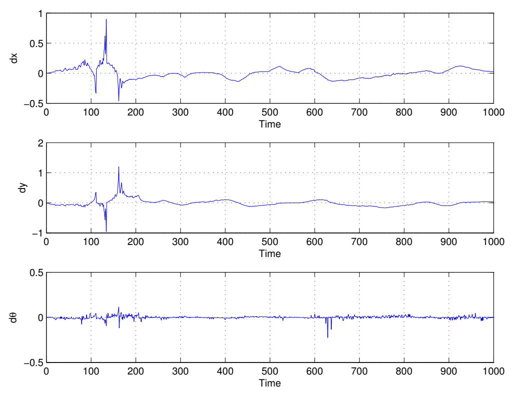


Figure 6.1: Vehicle pose estimation error. (Q7).

Alternatively we could make the filter's estimate of the covariance greater than the sensor's noise covariance

```
>> sensor = RangeBearingSensor(veh, map, W*2);
>> ekf = EKF(veh, V, P0, sensor, W, []);
```

We can change the sensor's sample rate by for example

```
>> sensor = RangeBearingSensor(veh, map, W, 'skip', 5);
```

so that it returns a range and bearing measurement every fifth time its method is called by the EKF.

We can also adjust the effective range of the sensor, for example

```
>> sensor = RangeBearingSensor(veh, map, W, 'range', 4);
```

limits the observation to landmarks that are within 4 m of the sensor. Alternatively we can limit returns to landmarks that are within ±90 deg of the orientation of the sensor by

```
>> sensor = RangeBearingSensor(veh, map, W, 'angle', [-1 1]*pi/4);
```

We can also simulate a temporary failure of the sensor

```
>> sensor = RangeBearingSensor(veh, map, W, 'fail', [100 150]);
```

between time steps 100 and 150.

If the parameters of the sensor are changed as above this creates a new instance of the `sensor` object, and a new instance of the filter needs to be created

```
>> ekf = EKF(veh, V, P0, sensor, W, []);
```

that is connected to the new `sensor` object.

Alternatively the properties of the sensor object can be adjusted without needing to create a new EKF filter object, for example

```
>> sensor.interval = 5;
>> sensor.r_range = [0 4];  % [minrange maxrange]
>> sensor.th_range = [-1 1]*pi/4; % [minrange maxrange]
```

Q7b See the approach from **Q5c**.

Q7c See the approach from **Q5e**.

Q7d See the approach from **Q5f**.

Q7e The approach is similar to that for **Q5g**. Just as for plain localisation the covariance for the SLAM system varies along the path. This can be seen clearly by

```
>> ekf.plot_error()
```

which shows the estimation error (actual from the vehicle simulation minus the EKF estimate) overlaid on the $\pm 3\sigma$ uncertainty bounds. For this question we could consider the final covariance which will be consistent if the vehicle always follows the same path, which the RandomPath driver object does (it resets its random number generator on every run). The covariance matrix is large for the SLAM case, it includes the covariance of the features, but the covariance of the vehicle states can be found by

```
 >> Pv = diag( ekf.P_est(1:3,1:3) )'
Pv =
    0.0050    0.0072    0.0002
```

Now we can create a table of final covariance for the cases where we vary the covariance of the sensor observations $\boldsymbol{W}$ by a command like

```
>> sensor.W = diag( [0.2 1*pi/180].^2);
```

and then rerunning the simulation.

Q8a With the standard parameters it is very unlikely for the filter to not converge.

Q8b With standard settings as per RVC the filter converges in 10-20 cycles to a standard deviation of around 0.3.

If we run the filter with larger $\boldsymbol{Q}$ we see that the particle cloud is larger during the animation

```
>> pf=ParticleFilter(veh,sensor,Q*10,L, 1000);
>> pf.run(1000);
```

and the filter converges in around 10 cycles but the steady state standard deviation is larger, around 0.6. At each time step the filter explores a large range of possible poses.

If we increase $\boldsymbol{L}$ we are more tolerant of poses that poorly explain the sensory data. Convergence is slower, unto 30 cycles and the steady state standard deviation is larger, around 0.5. Reducing $\boldsymbol{L}$ has the opposite effect and you might observe multiple hypotheses for a while before one of them wins out.

To change w_0 we need to set the filter property after it has been constructed

```
>> pf=ParticleFilter(veh,sensor, Q, L, 1000);
>> pf.w0
ans =
    0.0500
```

and we can set it to a smaller value

```
>> pf.w0 = 1;
```

and then run the filter

```
>> pf.run(1000)
```

we see that convergence to a single hypothesis takes many cycles. The large offset w_0 means that the difference in weights between poses that explain sensory observations well or poorly is reduced.

Finally we can adjust the number of particles. A drastic reduction

```
>> pf=ParticleFilter(veh,sensor, Q, L, 100);
>> pf.run(1000)
```

results in a filter that has a poor chance of converging. The problem is that with 100 particles spread over the pose space it is unlikely that one will be close to the actual vehicle pose. Increasing L may improve things by spreading the likelihood function so that a particle close to the vehicle will have its weight increased. The execution time of the filter is proportional to the number of particles so there is some merit in trying to find the minimum number of particles that gives reliable estimation.

Of course the filter is stochastic and gives different results on every cycle so to be certain of the effect of parameters results should be averaged over several runs. The option `'no plot'` dispenses with the animation and makes the simulations much faster.

Q8c The initialisation of particles takes place in the `m`ethod with the line

```
pf.x = (2*pf.rand([pf.nparticles,3]) - 1) * diag([pf.dim, pf.dim, pi]);
```

The first part of the expression generates uniformly distributed random numbers over the interval $[-0.5, 0.5]$ which are then scaled to cover the x- and y-dimensions of the map and the full range of orientations.

If we knew the vehicle was within the box with $x \in [0,4]$ and $y \in [2,6]$ we could write instead

```
pf.x = [ 4*pf.rand([pf.nparticles,1]), ...
  4*pf.rand([pf.nparticles,1])+2, ...
  pi*(2*pf.rand([pf.nparticles,1])-1)];
```

and if, further we knew the orientation was 0.5 rad then we could write

```
pf.x = [ 4*pf.rand([pf.nparticles,1]), ...
  4*pf.rand([pf.nparticles,1])+2, ...
  0.5*ones(pf.nparticles,1)];
```

Q8d The likelihood function is currently hard coded into the `m`ethod

```
LL = -0.5*[invL(1,1); invL(2,2); 2*invL(1,2)];
e = [z_pred(:,1).^2 z_pred(:,2).^2 z_pred(:,1).*z_pred(:,2)]*LL;
pf.weight = exp(e) + pf.w0;
```

which is an example of vectored code and where `z_pred` is the error between the expected and actual sensor reading. We could write an inverse distance version as

```
e = colnorm( pf.L * z_pred')';
pf.weight = 1 ./ (e + pf.w0);
```

where we use w_0 to prevent an error when the error is zero.

Part III

Arm-type Robots

Chapter 7

Kinematics

Q1 We can create a model of the 6-axis Puma 560 robot

```
>> mdl_puma560
```

which is a workspace object called `p560`. We can then create a virtual "teach pendant" for this object by

```
>> p560.teach()
```

The teach window has one slider per joint, and moving the slide adjusts the angle of the corresponding robot joint and the motion of the robot can be seen in another figure.

Q2 With reference to Figure 7.3 we define the coordinates of the end-effector as (x_2, y_2) and the coordinates of the elbow joint as (x_1, y_1). We can then write

$$x_1 = a_1 \cos\theta_1 \tag{7.1}$$

$$y_1 = a_1 \sin\theta_1 \tag{7.2}$$

and

$$x_2 = x_1 + a_2 \cos(\theta_1 + \theta_2) \tag{7.3}$$

$$y_2 = y_1 + a_2 \sin(\theta_1 + \theta_2) \tag{7.4}$$

Q3 Consider the triangle comprising the base of the robot, the end-effector and the elbow joint, which is shown in Figure ??(a). Using the cosine law for this triangle we can write

$$(x_2^2 + y_2^2) = a_1^2 + a_2^2 - 2a_1a_2\cos\theta_2'$$

where θ_2' is the complementary angle at the elbow.

$$\cos\theta_2 = \frac{(x_2^2 + y_2^2) - a_1^2 - a_2^2}{2a_1a_2}$$

and we could solve for θ_2 using the arccos function. A better approach is to write

$$C_2 = \frac{(x_2^2 + y_2^2) - a_1^2 - a_2^2}{2a_1a_2}$$

and we can write the sine of the angle as

$$S_2 = \pm\sqrt{1 - C_2^2}$$

and then use the arctan function

$$\theta_2 = \tan^{-1} \frac{\pm\sqrt{1 - C_2^2}}{C_2}$$

This has two values which correspond to two geometric configurations, the elbow above or below the line joining the base to the end effector as shown in Figure ???(b)

Consider the triangle shown in Figure ??(c) where we can write an expression for the included angle θ as

$$\theta = \tan^{-1} \frac{a_2 \sin\theta_2}{a_1 + a_2 \cos\theta_2}$$

and we can write

$$\theta_1 + \theta = \tan^{-1} \frac{y_2}{x_2}$$

and then finally we obtain

$$\theta_1 \tan^{-1} \frac{y_2}{x_2} - \tan^{-1} \frac{a_2 \sin\theta_2}{a_1 + a_2 \cos\theta_2}$$

Q4 We will choose a singularity free pose with joint angles given by `qn`. The end-effector pose is

```
>> T = p560.fkine(qn);
```

The numerical solution is

```
>> p560.ikine(T)
ans =
   -0.0000   -0.8335    0.0940    0.0000   -0.8312   -0.0000
```

which if we plot it shows the robot in a right-handed elbow down configuration. Remember that with the numerical solution we cannot directly control the configuration except by appropriate choice of initial joint angles which are an optional second argument to `.` The analytic solution for this configuration is

```
>> p560.ikine6s(T, 'rdn')
ans =
   -0.0000   -0.8335    0.0940   -0.0000   -0.8312   -0.0000
```

which is the same to four significant figures.

To determine execution time we can write

```
>> tic; p560.ikine6s(T, 'rdn'); toc
Elapsed time is 0.002200 seconds.
>> tic; p560.ikine(T); toc
Elapsed time is 1.064866 seconds.
```

and see that the analytic solution is around 500 times faster. Using `tic` and `doc` is probably not very accurate to estimate the very short execution time of `.` Using the pseudo-inverse option to `r`esults in a significant speed up

```
>> tic; p560.ikine(T, 'pinv'); toc
Elapsed time is 0.033750 seconds.
```

Q5 We plot the initial pose

```
>> p560.plot(qn)
```

and we see that the robot is in an elbow up right-handed (righty) configuration. To obtain the elbow down configuration we first find the end-effector pose

```
>> T = p560.fkine(qn);
```

and then solve for the joint angles in the right-handed elbow down configuration

```
>> qd = p560.ikine6s(T, 'rd')
qd =
   -0.0000   -0.8335    0.0940    3.1416    0.8312   -3.1416
```

Now we can animate the motion of the robot as it changes configuration

```
>> qt = jtraj(qu, qd, 50);
>> p560.plot(qt)
```

and we see that the elbow rotates almost one revolution which is not physically achievable, and the end-effector ends up back at the initial pose.

Chapter 8

Velocity relationships

Q1 Mechanically it is not possible for three joint axes to be aligned. The axes of joints 4 and 6 are aligned if $q_5 = 0$. The arm has a horizontal offset at the "shoulder" which so these axes cannot align with the waist rotation axis of joint 1.

Q2 For the case of Euler angles we write

$$\boldsymbol{R} = \boldsymbol{R}_z(\phi)\boldsymbol{R}_y(\theta)\boldsymbol{R}_z(\psi)$$

which we can expand as

$$\boldsymbol{R} = \begin{pmatrix} c\phi c\psi c\theta - s\phi s\psi & -c\psi s\phi - c\phi c\theta s\psi & c\phi s\theta \\ c\phi s\psi + c\psi c\theta s\phi & c\phi c\psi - c\theta s\phi s\psi & s\phi s\theta \\ -c\psi s\theta & s\psi s\theta & c\theta \end{pmatrix}$$

$c\theta$ and $s\theta$ to mean $\cos\theta$ and $\sin\theta$ respectively. With some tedium we can write the derivative

$$\dot{\boldsymbol{R}} = \begin{pmatrix} -c\phi c\psi s\theta\dot{\theta} - (c\phi c\theta s\psi + c\psi s\phi)\dot{\psi} - (c\phi s\psi + c\psi c\theta s\phi)\dot{\phi} & c\phi s\psi s\theta\dot{\theta} + (s\phi s\psi - c\phi c\psi c\theta)\dot{\psi} + (c\theta s\phi s\psi - c\phi c\psi)\dot{\phi} \\ -c\psi s\phi s\theta\dot{\theta} + (c\phi c\psi - c\theta s\phi s\psi)\dot{\psi} + (c\phi c\psi c\theta - s\phi s\psi)\dot{\phi} & s\phi s\psi s\theta\dot{\theta} - (c\phi s\psi + c\psi c\theta s\phi)\dot{\psi} - (c\phi c\theta s\psi + c\psi s\phi\dot{\phi}) \\ -c\psi c\theta\dot{\theta} + s\psi s\theta\dot{\psi} & c\theta s\psi\dot{\theta} + c\psi s\theta\dot{\psi} \end{pmatrix}$$

Recalling Eq 3.4

$$\dot{\boldsymbol{R}} = [\omega]_\times \boldsymbol{R}$$

which we can arrange as

$$[\omega]_\times = \dot{\boldsymbol{R}}\boldsymbol{R}^T$$

That is the product of the two complicated matrices above is a relatively simple skew symmetric matrix

$$\begin{pmatrix} 0 & -c\theta\dot{\psi} - \dot{\phi} & c\phi\dot{\theta} + s\phi s\theta\dot{\psi} \\ c\theta\dot{\psi} + \dot{\phi} & 0 & s\phi\dot{\theta} - c\phi s\theta\dot{\psi} \\ -c\phi\dot{\theta} - s\phi s\theta\dot{\psi} & -s\phi\dot{\theta} + c\phi s\theta\dot{\psi} & 0 \end{pmatrix}$$

and by equating elements with Equation (3.5) we can write

$$\begin{pmatrix} \omega_x \\ \omega_y \\ \omega_z \end{pmatrix} = \begin{pmatrix} -s\phi\dot{\theta} + c\phi s\theta\dot{\psi} \\ c\phi\dot{\theta} + s\phi s\theta\dot{\psi} \\ c\theta\dot{\psi} + \dot{\phi} \end{pmatrix}$$

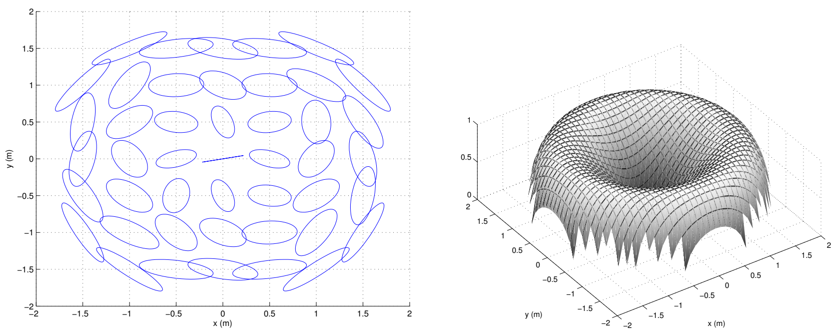


Figure 8.1: Q3a. (left) Manipulability ellipses, (right) manipulability scalar field.

which can be factored as

$$\omega = \begin{pmatrix} 0 & -s\phi & c\phi s\theta \\ 0 & c\phi & s\phi s\theta \\ 1 & 0 & c\theta \end{pmatrix} \begin{pmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{pmatrix}$$

and written concisely as

$$\omega = \boldsymbol{B}(\boldsymbol{\Gamma})\dot{\boldsymbol{\Gamma}}$$

This matrix $\boldsymbol{B}$ is itself a Jacobian that maps Euler angle rates to angular velocity.

We can also solve this using the MATLAB Symbolic Toolbox

```
>> syms t phi(t) theta(t) psi(t) real
>> R = rotz(phi) * roty(theta) * rotz(psi)
>> diff(R, t) * R.'
>> S = simple(ans)
```

Note that we set an assumption that all variables are real, and that `phi`, `theta` and `psi` are all functions of `t`. We also use the operator `.'` to indicate non-conjugate transpose. Performing the remaining steps in deriving the Jacobian, equating coefficients and factorizing, is somewhat cumbersome to achieve using the Toolbox.

Q3a We first load the two-link model

```
>> mdl_twolink
```

With the link lengths each being 1 m the work space is a circle centred at origin, of radius 2 m and lying in the xy-plane. We will create a grid of points at which to evaluate the manipulability

```
range = [-2:0.5:2];
[X,Y] = meshgrid( range, range);
```

and for each of these points we will determine the joint angles and then manipulability. The first code segment plots the manipulability ellipses

```
q0 = [0.1 0.2];
for i=1:numel(X)
  x = X(i); y = Y(i);
```

```
    if norm([x y]) < 2
      q = twolink.ikine( transl(x, y, 0), q0, [1 1 0 0 0 0], 'pinv');
      J = twolink.jacob0(q);
      J = J(1:2,:);
      plot_ellipse( 0.05*J*J', [x y]);
    end
  end
```

suitably scaled for visibility. Note that unreachable points, with a radius greater than 2 m, are skipped.

The second code segment uses a denser grid of points

```
  range = [-2:0.1:2];
  [X,Y] = meshgrid( range, range);
  M = zeros(size(X));

  q0 = [0.1 0.2];

  for i=1:numel(X)
    x = X(i); y = Y(i);
    if norm([x y]) < 2
      q = twolink.ikine( transl(x, y, 0), q0, [1 1 0 0 0 0], 'pinv');
      M(i) = twolink.maniplty(q, 'dof', [1 1 0 0 0 0]);
    else
      M(i) = NaN;
    end
  end
```

The array `M` contains the manipulability measures and the value is `NaN` if the point is not reachable by the robot. We can display manipulability as a scalar intensity field

```
  >> idisp(M, 'xydata', range, range, 'nogui')
  >> xlabel('x (m)'); ylabel('y (m)')
```

where pixel brightness is proportional to manipulability or as a 3-dimensional surface

```
  >> surf(X, Y, M)
  >> shading interp
```

Q3b At the configuration `qn` the manipulability is

```
  >> p560.maniplty(qn)
  ans =
      0.1112
```

If we perturb each joint angle by 0.1 rad and then by -0.1 rad we get manipulabilities

```
  >> p560.maniplty([bsxfun(@plus, qn', 0.1*eye(6,6)) bsxfun(@plus, qn', -0.1*eye(6,6))]')
  ans =
      0.1112
      0.1109
      0.1168
      0.1112
      0.1112
      0.1112
      0.1112
      0.1104
      0.1039
      0.1112
      0.1112
      0.1112
```

we see that the third one is higher which corresponds to an increase in the elbow angle.

Q3c Consider the path through a singularity described in Sec 7.4.3

```
>> T1 = transl(0.5, 0.3, 0.44) * troty(pi/2);
>> T2 = transl(0.5, -0.3, 0.44) * troty(pi/2);
>> Ts = ctraj(T1, T2, 50);
```

We initialise the graphics and draw an ellipsoid at a default location and obtain a handle for it

```
clf
h =  plot_ellipse(0.001*diag([1 1 1]), [0 0 0], 'FaceColor', 'none')
```

Then for every point along the path we update the animation of the robot and the animation of the ellipsoid

```
% For every point along the path
for i=1:size(Ts,3)
  T = Ts(:,:,i);
  q = p560.ikine6s(T);
  p560.plot(q);

  J = p560.jacob0(q);
  J =J(1:3,:);
  E = J*J';
  plot_ellipse(0.1*E, transl(T), 'alter', h, 'FaceColor', 'none')
  pause(0.1)
end
```

This example shows the translational velocity ellipsoid, alter line ?? to

```
  J =J(4:6,:);
```

to obtain the rotational velocity ellipsoid.

Q4a Edit the Simulink model `sl_rrmc` and adjust the constant in the block desired Cartesian velocity. For example the value `[0.05 0 0 0 0 0]` is slow motion in the world x-direction until the manipulator reaches a singularity at maximum extent. Choosing `[0 0 0 0.1 0 0]` rotates the end-effector coordinate frame around the world x-axis.

Q4b Run a wire from the output of the jacob0 block through an Interpreted MATLAB Function block which contains the MATLAB function `get` and then wire that to a To Workspace block with the save format set to Array and the variable name set to `jdet`. After running the simulation this workspace variable can be plotted

```
>> plot(jdet)
```

and shows the variation in the Jacobian determinant along the trajectory.

Q4c In the case of finite translational velocity the robot end-effector will move toward the workspace limit after which it can move no further.

Q4d We start the robot at pose

```
>> T1 = transl(0.5, 0.3, 0.44) * troty(pi/2);
```

which has joint angles

```
>> q = p560.ikine6s(T1)
q =
    3.4218    1.8818   -0.3310   -1.5015    0.2808   -1.6429
```

We modify the model and set the end-effector to move in the negative world y-direction by setting the Cartesian velocity to `[0 -0.1 0 0 0 0]`. We set the initial configuration to `q` by changing the initial condition of the Joint servo block to `q'`. Running the simulation we observe the end-effector rotation rotates by $\pi/2$ about its z-axis as we pass through the singularity. We can plot the joint angles

```
 >> mplot(out)
```

Q4e Replacing the `inv` function with `pinv` makes little difference in this case.

Q4f Replacing the `inv` function with `inv(u` 0.1*eye(6))+ to implement damped least squares again makes little difference in this case. The joint rates at the singularity are however lower.

Q4g Requires changing the argument of the inv block to `lscov(u, [0 -0.1 0 0 0 0]')`, and again makes no difference.

Q4h We load the Simulink model

```
>> sl_rrmc2
```

We need to change the name of the robot from `p560` to `p8` in the blocks fine and jacob0, change the function to `pinv` in the block inv and the initial conditions to `[0 0 qn]'` in the block joint servo. Plotting the joint angles

```
>> mplot([], q)
```

we see that most of the work in tracking the circle is done by joints 1, 2, 3 and 8. The other joints move from their initial angle to a constant value for the rest of the trajectory.

Q4i We modify the Simulink model of Fig 8.5 (RVC) to that shown in Fig 8.2. The wiring shown in green is the null-space motion controller. We compute the projection matrix $\boldsymbol{NN}^{+}$ and multiply it by the desired joint motion. The desired joint motion is for the Puma 560 joints (joints 3 to 6) to move towards zero which is achieved by multiplying the joint angles by $-[00111111]$. From the simulation results we see that Puma joints 1, 4 and 6 move to zero but the others maintain a finite value that is required to achieve the desired end-effector pose.

Q5a Challenging!

Q5b Consider the P8 robot at the pose

```
>> q = [0 0 qn]
and the Jacobian is
>> J = p8.jacob0(q);
>> about J
J [double] : 6x8 (384 bytes)
```

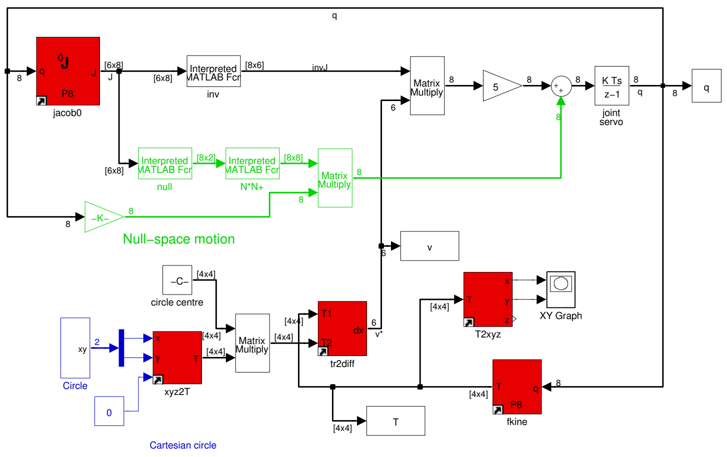


Figure 8.2: Q4i. Resolved rate motion control with null-space motion controller shown in green.

For a Jacobian each column is the end-effector spatial velocity for unit velocity of the corresponding joint. We wish to find the subset of six joints which can achieve any possible end-effector spatial velocity, while the remaining joints could be locked without limiting the ability to achieve arbitrary end-effector motion. In linear algebra terms we need to find the six columns that form a basis set that span $\mathbb{R}^6$. The motion due to the locked joints can be achieved by a linear combination of the selected joints, that is the columns of the locked joints can be expressed in terms of the six basis vectors.

We can determine the columns to use for our basis set by putting the Jacobian into reduced row echelon form

```
>> [r, jb] = rref(J);
>> jb
jb =
     1     2     3     4     5     6
```

which shows that joints 1–6 are required. Joints 7 and 8 could be effectively locked without limiting the achievable end-effector velocity.

Note that this analysis is kinematic only. It may be that the final Puma 560 joint, a high speed wrist joint, is capable of higher motion that the coordinated motion of several larger and slower joints closer to the robot's base.

Chapter 9

Dynamics

Q1 The code on p 194

```
[Q2,Q3] = meshgrid(-pi:0.1:pi, -pi:0.1:pi);
for i=1:numcols(Q2),
        for j=1:numcols(Q3);
                g = p560.gravload([0 Q2(i,j) Q3(i,j) 0 0 0]);
                g2(i,j) = g(2);
                g3(i,j) = g(3);
        end
end
surfl(Q2, Q3, g2); surfl(Q2, Q3, g3);
```

is available as the m-file `eg_grav`. To compute the figure we use

```
mdl_puma560
eg_grav
```

To draw the figures with axis labels we could use

```
surfl(Q2, Q3, g2);  xlabel('q_2'); ylabel('q_3'); zlabel('g_2');
surfl(Q2, Q3, g3); xlabel('q_2'); ylabel('q_3'); zlabel('g_3');
```

Adding a payload is quite straightforward.

```
p560.payload(2.5, [0, 0, 0.1]);
```

adds a 2.5 kg mass at the position $(0,0,0.1)$ m in the $\boldsymbol{T}_6$ coordinate frame.

Q2 The code on p 195

```
[Q2,Q3] = meshgrid(-pi:0.1:pi, -pi:0.1:pi);
for i=1:numcols(Q2),
    for j=1:numcols(Q3);
        M = p560.inertia([0 Q2(i,j) Q3(i,j) 0 0 0]);
        M11(i,j) = M(1,1);
        M12(i,j) = M(1,2);
    end
end
surfl(Q2, Q3, M11); surfl(Q2, Q3, M12);
```

is available as the m-file `eg_inertia`. To compute the figure we use

```
mdl_puma560
eg_inertia
```

To draw the figures with axis labels we could use

```
surfl(M11, Q3, g2);  xlabel('q_2'); ylabel('q_3'); zlabel('M_11');
surfl(M12, Q3, g3); xlabel('q_2'); ylabel('q_3'); zlabel('M_12');
```

Q3 Similar to the example above

```
mdl_puma560
q3 = linspace(-pi, pi, 50);
for i=1:numcols(q3);
      M22(i) = p560.inertia([0 0 q2(i) 0 0]);
end
```

and we plot M_{22}

```
plot(q2, M22);
xlabel('q_2'); ylabel('M_22');
```

We add a payload as before

```
p560.payload(2.5, [0, 0, 0.1]);
```

then repeat

```
mdl_puma560
q3 = linspace(-pi, pi, 50);
for i=1:numcols(q3);
      M22_p(i) = p560.inertia([0 0 q2(i) 0 0]);
end
```

and then overlay the plots

```
plot(q2, M22, M22_p);
xlabel('q_2'); ylabel('M_22');
legend('no payload', 'with payload')
```

Q4 Continuing on from Q3

```
>> max(M22)/min(M22)
??
```

For the case with a payload the inertia varies by

```
>> max(M22_p)/min(M22_p)
??
```

Q5 According to Spong et al. [1] the kinetic energy of the manipulator is given by the matrix quadratic expression

$$K = \frac{1}{2}\dot{q}^T \boldsymbol{M}(q)\dot{q}$$

Since kinetic energy must always be non-negative the matrix $\boldsymbol{M}$ must be positive definite and therefore symmetric. They also shown in equation 7.50 that $\boldsymbol{M}(\boldsymbol{q})$, which they call $\boldsymbol{D}(\boldsymbol{q})$ is the sum of two positive definite matrices.

Q6 We will work with the Puma 560 robot and assume that it is sitting on a plate that is sufficiently large to prevent it from toppling over. The plate has a flat base and moves over a frictionless table, for instance an air table, which opposes gravity load and any rotation about the x- and y-axes. The only degrees of freedom are position in the plane and rotation about the z-axis, that is, (x, y, θ_z) .

?????

Q7 See the solution to **Q8-3c** but the ellipse is instead given by

```
J = p560.jacob0(q);
M = p560.inertia(q);
Mx = (J * inv(M) * inv(M)' * J');
E = Mx(1:3,1:3);
```

Q8a We load the model

```
>> vloop_test
```

and view the mask parameter by right-clicking on the loop icon. First we will set the Signal Generator to output a square wave. If we reduce K_v, say to 0.1, we observe poor tracking behaviour, a longer rise time on the edges of the square wave and a significant steady state error, the actual velocity is lower than desired. If we increase the gain to 1.2 we observe considerable oscillation on the step edges, an overshoot of more than 100% and three cycles of oscillation. Although square wave response is a common criteria for control system performance in practice the demand to the velocity loop will be smooth. If we change the Signal Generator to output a sine wave we observe good tracking performance with this higher gain.

We now add a disturbance torque, in reality this could be caused by gravity acting on the links supported by the joint, by setting the constant tau_d to 10 and we observe that the joint velocity has a considerable offset from the desired. Increasing K_v can reduce the offset somewhat, but for values above 1.6 the system is unstable.

Instead we will return the velocity loop gain to a reasonable value, say $K_v = 1$ and increase the integral gain K_i instead. For a sine wave demand and for a value of $K_i = 5$ we notice that the offset is eliminated after one cycle of the sine wave. Using the Look under Mask option we can open the block vloop and see the integrator which is accumulating the velocity error and its output is tending to force that error toward zero. The output value of the integrator is available as an output of the vloop block and we can plot that by attaching a new Scope block. We see the integrator output asymptoting to the value that counters the disturbance force, that is, the integrator has *learnt* the disturbance. The gain K_i controls that rate at which the disturbance is learnt. If the disturbance is time varying, as it is for the case of gravity load in a robot whose configuration is changing, then this gain must be set high enough to track that change.

Q8b From RVC page 205 we can write the open loop dynamics as

$$\frac{\Omega(s)}{U(s)} = \frac{K_m K_a}{Js+B}$$

Using the Control Systems Toolbox we can write this as

```
>> Km = 0.228;
>> J = 372e-6;
>> B = 817e-6;
>> vloop = tf( Km, [J B]);
```

where `loop` is a `tf` class object that represents the continuous-time dynamics

```
>> vloop
vloop =

          0.228
  ---------------------
  0.000372 s + 0.000817

Continuous-time transfer function.
```

of the joint. The pole is given by

```
>> eig(vloop)
ans =
   -2.1962
```

We can plot the open-loop step response

```
>> step(vloop)
```

which has a first-order response dominated by inertia and friction. The root locus plot is

```
>> rlocus(vloop)
```

and we see the real pole at -2.2 rad/s. As loop gain increases the pole moves toward $-\infty$. There is no limit on the loop gain but from a stability point of view but increased gain will lead to larger demands on the motor that will ultimately lead to saturation. At that point the system becomes non-linear and our linear analysis no longer valid. The instability we observed above is related to numerical problems with the Simulink solver.

If we add a PI compensator

$$D(s) = 1 + \frac{K_i}{s} = \frac{s+K_i}{s}$$

```
>> d = tf( [1 Ki], [1 0]);
```

and the root locus of the system with compensator is

```
>> rlocus(d*vloop)
```

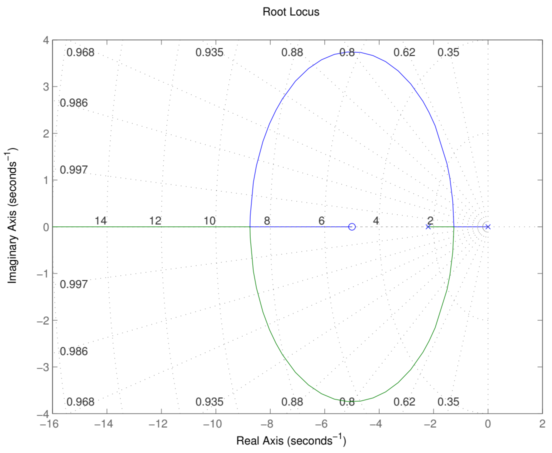


Figure 9.1: Q4a. Root locus of velocity loop with integral action.

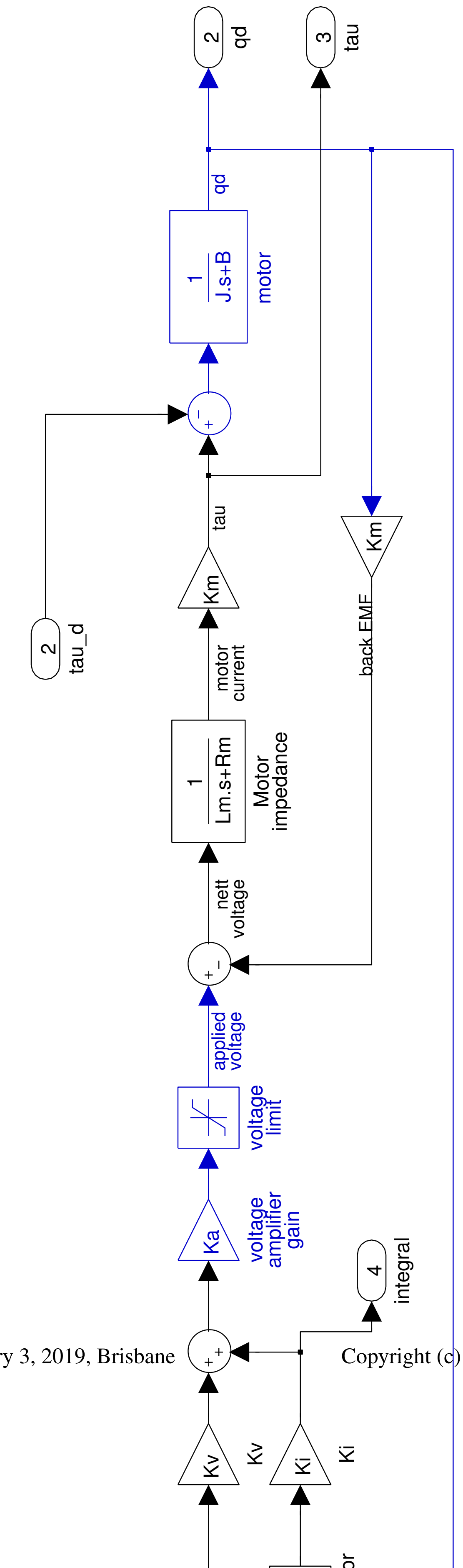
We see an additional pole at the origin due to the integral action and the poles branch away from the real-axis allowing us to choose a damping ratio less than one.

Q8c The motor model is shown in Figure 9.2 and we see that the motor current is now determined by the nett motor voltage, applied minus back EMF, divided by the motor impedance. The dynamics of this model are more complicated than for a current controlled motor since it has two poles: the mechanical pole as before at $s = -B/J$ and an electrical pole at $s = -L/R$. Typically the electrical pole is much faster than the mechanical pole. Back EMF behaves as an additional feedback loop and leads to a form of velocity control: for a given input voltage a current is forced through the motor causing it to accelerate and as its speed rises so to does the back EMF. The back EMF opposes the applied voltage and tends to reduce the motor current. The system reaches equilibrium when the nett motor voltage is sufficient to maintain a current that creates a torque that counters motor friction. We can also consider the back EMF effect as additional, electrical, damping.

Q8d If we use the the model `vloop_test` and set the demand to a square wave with amplitude 50 we can observe this effect. On the rising edges of the demand a large torque is required which exceeds the torque limit. The torque is therefore less than desired and the motor accelerates less quickly than desired and the resulting tracking error rapidly increases the integral value. Once the motor reaches the desired speed the demand due to velocity error falls to zero but the integrator has a large accumulated error which maintains a positive torque that pushes the motor past the demand and this leads to overshoot. The other problem is that the response to the step is low because when the demand changes sign the integrator has a considerable value of the opposite sign, subtracting from the demand computed due to velocity error. which leads we observe a very slow increase in motor velocity because the integrator has a large initial negative value which reduces the torque that is applied to the motor. The r which then drives the motor past the demand leading to overshoot.

Opening the inner vloop block we see the integrator block which we can open. If we check the Limit output box we can then set some limits for the integrator, say setting the upper and lower limits to +1 and -1 respectively.

Q8e


2
qd
3
tau
qd
1
J.s+B
motor
Km
tau
back EMF
Km
2
tau_d
motor current
1
Lm.s+Rm
Motor impedance
nett voltage
applied voltage
voltage limit
Ka
voltage amplifier gain
4
integral
Kv
Kv
Ki
Ki

Q9 We will do some manipulation of SI units and start with the motor torque constant units which are $\mathrm{N \cdot m \cdot A^{-1}}$. The Volt is a unit of electro-motive force but the word “force” is a misnomer, an unfortunate choice of words from nearly 200 years ago when these things were poorly understood.

A Volt is defined as the difference in electric potential across a wire when an electric current of one Ampere dissipates one Watt of power, that is $\mathrm{V = W/A}$. We can rearrange this as $\mathrm{A = W/V}$ and substitute into the above to give the torque constant units as $\mathrm{N \cdot m \cdot V/W}$. We know that a Watt is one Joule per second, $\mathrm{W = J \cdot s^{-1}}$ and that a Joule is the work done by a force of one Newton over a distance of one metre, $\mathrm{J = N \cdot m}$. Substituting these we can now write the motor torque constant units as $\mathrm{V \cdot s}$. Radians are treated as dimensionless so motor torque constant as $\mathrm{V \cdot s}$ is equivalent to back EMF constant $\mathrm{V \cdot s \cdot rad^{-1}}$.

Q10a We start by defining the robot model

```
>> mdl_puma560
```

and then loading the Simulink models

```
>> sl_fforward
>> sl_ctorque
```

Each of these models write variables to output ports and these are available within the Simulink.SimulationOutput object

```
>> r = sim('sl_fforward')
>> r
>> t = r.find('tout');
>> q = r.find('q');
```

The simulink models are also configured to write the simulation time and state directly to the workspace variables tout and yout directly.

Q10b

Part IV

Vision

Chapter 10

Light and Color

Q1 What is your blackbody emission spectrum. Spectrum in consideration from NUV(Near Ultra Violet) passing trough LWIR(Long Wave Infra Red)[from 8000nm to 15000nm] up to Thermal IR[from 15000nm to 100000nm]

```
clear all
lambda=[300:10:100000]*1e-9;

ZeroDegrees_Kelvin=273;%in Kelvin
BodyTemperature_Celsius=36.5;
MybodyTemperature=ZeroDegrees_Kelvin + BodyTemperature_Celsius;
MyBody = blackbody(lambda,MybodyTemperature);
plot(lambda*1e9,MyBody);
grid on
```

Q1b The peak emission frequency is

```
[PeakValue,PeakIndex]=max(MyBody);
PeakWavelength=(PeakIndex*10)+300;
PeakWavelength %in nanometers(nm)
PeakValue %in Watt/meters^3 (W/m^3)
%Answer:9370 nm.
```

This peak wavelength, nearly 10 μm is in the Long Wave Infra Red part of the spectrum.

Q1c A thermal infrared sensor which uses an array of micro bolometers rather than light sensitive silicon pixels.

Q2

```
h=6.626*1e-34;%Planck's constant in Joule per seconds (Js)
k=1.381*1e-23;%Boltzmann's constant in Joule divided Kelvin (J/K)
c=2.998*1e8;%speed of light in meters divided seconds (m/s)
lambda1=PeakWavelength*1e-9
P1=PeakValue

T1 = (h*c)/(k*lambda1*log(((2*h*(c^2))/(lambda1^5*P1)) + 1))

lambda2=(((PeakIndex+100)*10)+300)*1e-9
P2=MyBody(PeakIndex+100)
T2 = (h*c)/(k*lambda2*log(((2*h*(c^2))/(lambda2^5*P2)) + 1))
```

```
lambda3=(((PeakIndex-100)*10)+300)*1e-9
P3=MyBody(PeakIndex-100)
T3 = (h*c)/(k*lambda3*log(((2*h*(c^2))/(lambda3^5*P3)) + 1))

lambda4=(((PeakIndex+3000)*10)+300)*1e-9
P4=MyBody(PeakIndex+3000)
T4 = (h*c)/(k*lambda4*log(((2*h*(c^2))/(lambda4^5*P4)) + 1))

average=(T1+T2+T3+T4)/4

[rows,columns]=size(MyBody)

TotalEnergy=0;
for i=1:1:rows
    TotalEnergy=TotalEnergy+MyBody(i)*1e-8;
end

TotalEnergy
%Etotal=((2*(pi^5)*(k^4))/(15*(c^2)*(h^3)))*T^4 -->
%T^4=Etotal*((15*(c^2)*(h^3))/(2*(pi^5)*(k^4))) -->
%T=(Etotal*((15*(c^2)*(h^3))/(2*(pi^5)*(k^4))))^(1/4)
TBody=((TotalEnergy*(15*(c^2)*(h^3)))/(2*(pi^5)*(k^4)))^(0.25)


%Question number 3:
%Using the Stephan-Boltzman law compute the power emitted per square metre
%of the Sun's surface.Compute the total power output of the Sun.
TSun=6500%in Kelvin
lambdaSun=[10:10:6000]*1e-9;
Sun=blackbody(lambdaSun,TSun);
figure(2)
plot(lambdaSun*1e9,Sun)
grid on
ESuntotal=((2*(pi^5)*(k^4))/(15*(c^2)*(h^3)))*TSun^4%Stephan-Boltzman law
```

Q4

```
%Classical integration forward Euler
TotalEnergyVisibleSpectrum=0;
for i=40:1:70
    TotalEnergyVisibleSpectrum=TotalEnergyVisibleSpectrum+Sun(i)*1e-8;
end
TotalEnergyVisibleSpectrum
%Trapezoidal Integration
TotalEnergyVisibleSpectrum=0;
for i=40:1:69
    TotalEnergyVisibleSpectrum=TotalEnergyVisibleSpectrum+((Sun(i+1)+Sun(i))/2)*1e-8;
end
TotalEnergyVisibleSpectrum
%Answer:Using Euler forward method 1.2931*1e7 (Wm^2), using
%Trapezoidal integration 1.2543*1e7 (Wm^2)
TSunCheck=((TotalEnergyVisibleSpectrum*(15*(c^2)*(h^3)))/(2*(pi^5)*(k^4)))^(0.25)

%Question number 6:
%Given Typical illuminance as per page 229 determine the luminous intensity
%of the Sun.

%The illuminance is calculated as:E=I/d^2
%E=Illuminance of the point light source is the total luminous flux
%arriving(incident) on a surface that has a distance d from the light
%source, with the light source having a luminous intensity on its surface
%equal to I; I=Luminous intensity of the point light source;
```

```
%d=Distance between the light source and the surface receiveing the light.

AverageDistanceEarthSun=(147098074+152097701)/2;%First number perielium
%(minimum distance) in kms second number afelium(max distance) in kms
SunIlluminance=10000%in a sunny day measured as luxs
SunLuminousIntensity=(AverageDistanceEarthSun*1000)*SunIlluminance^2
```

Q7

```
lambda=[350:10:2200]*1e-9;%The data we have solar.dat starts from 350nm and
%and it ends about 2200nm
SunGround=loadspectrum(lambda,'solar.dat');
UVSunEmission=0.0;
VisibleSunEmission=0.0;
IRSunEmission=0.0;
ThermalIRSunEmission = 0.0;
TotalEmission=0;
[rows,columns]=size(lambda);
UVThreshold= 400*1e-9;
VisibleThreshold = 700*1e-9;
IRThreshold=10000*1e-9;
figure(3)
plot(lambda*1e9, SunGround);

for i=1:1:(columns-1)
    if (lambda(1,i)<=UVThreshold)
        UVSunEmission=UVSunEmission+((SunGround(i+1,1)+SunGround(i,1))/2)*1e-8;
    elseif ((lambda(1,i)>UVThreshold)&&(lambda(1,i)<=VisibleThreshold))
        VisibleSunEmission=VisibleSunEmission+((SunGround(i+1,1)+SunGround(i,1))/2)*1e-8;
    elseif ((lambda(1,i) > VisibleThreshold) && (lambda(1,i)<=IRThreshold))
        IRSunEmission=IRSunEmission+((SunGround(i+1,1)+SunGround(i,1))/2)*1e-8;
    else
        ThermalIRSunEmission=ThermalIRSunEmission+((SunGround(i+1,1)+SunGround(i,1))/2)*1e-8;
    end


end

TotalEmission=UVSunEmission+VisibleSunEmission+IRSunEmission+ThermalIRSunEmission
UVPercentage=(UVSunEmission/TotalEmission)*100
VisiblePercentage=(VisibleSunEmission/TotalEmission)*100
IRPercentage=(IRSunEmission/TotalEmission)*100
ThermalIRPercentage=(ThermalIRSunEmission/TotalEmission)*100
grid on
```

Q8

```
lambda=[350:10:10000]*1e-9;
TLamp=2600;
Lamp = blackbody(lambda,TLamp);
figure(4)
plot(lambda*1e9, Lamp);
[rows,columns]=size(lambda);
VisibleLampEmission=0;

for i=1:1:(columns-1)

    if ((lambda(i)>UVThreshold)&&(lambda(i)<=VisibleThreshold))
        VisibleLampEmission=VisibleLampEmission+((Lamp(i)+Lamp(i+1))/2)*1e-8;
    end
```

```
end
grid on

VisibleLampEmission
```

Q9 Plot and compare the human photopic and scotopic spectral response.

```
lambda=[300:10:800]*1e-9;
human=luminos(lambda);
figure(5)
plot(lambda*1e9, human);
grid on
%camera=ccdresponse(lambda);

flowers=iread('flowers4.png','double','gamma','sRGB');
yuv=colorspace('RGB->YUV',flowers);
hsv=colorspace('RGB->HSV',flowers);
idisp(yuv(:,:,3))

flowersmedia=(flowers(:,:,1)+flowers(:,:,2)+flowers(:,:,3))/3;
figure(6)
idisp(100*flowersmedia)
figure(7)
idisp(flowersmedia)
flowersmediarg=(flowers(:,:,1)+flowers(:,:,2))/2;
flowersmediabg=(flowers(:,:,2)+flowers(:,:,3))/2;
flowersmod=flowers;
v=abs(1-hsv(:,:,3));
flowersmod(:,:,1)=flowersmedia(:,:).*v(:,:);
flowersmod(:,:,2)=(hsv(:,:,1)/360).*v(:,:);
flowersmod(:,:,3)=(1-hsv(:,:,2)).*v(:,:);
figure(8)
idisp(flowersmod)
figure(9)
idisp(flowers)
figure(10)
idisp(abs(flowers-flowersmod))
```

Chapter 11

Image Formation

Q1 We create a central camera with default parameters by

```
>> cam = CentralCamera('default');
```

and a smallish cube that is 20 cm on a side

```
>> [X,Y,Z] = mkcube(0.2, [0 0 1], 'edge');
```

and situated 1 m in front of the camera, that is, in the camera's positive z-direction. The cube has a mesh representation which means we can visualize its edges rather than just its vertices.

To view the cube as seen by the camera

```
>> cam.mesh(X,Y,Z)
```

Now we can move the camera 20 cm to the right, in the camera's positive x-direction. An obvious approach is to generate another camera

```
>> cam2 = CentralCamera('default', 'pose', transl(0.2, 0, 0) );
```

where the centre has been moved in the x-direction, and viewing that

```
>> cam.mesh(X,Y,Z)
```

shows that the cube has moved to the left in the camera's field of view. Check that — as you move your head to the right, the world appears to move to the left.

Yet another way is to move our original camera

```
>> cam.T = transl(0.2, 0, 0);
```

and then visualize the cube

```
>> cam.mesh(X,Y,Z)
```

but this changes the camera's pose until we change it again. Let's put it back to the origin

```
>> cam.T = [];
```

and instead temporarily move the camera

```
>> cam.move( transl(0.2, 0, 0) ).mesh(X,Y,Z);
```

where the `move` method creates a new camera object, the same as `cam`, but translated, and we invoke its `mesh` method.

There are lots of ways to solve this problem with the toolbox but perhaps the simplest way to solve this particular problem is

```
>> cam.mesh(X,Y,Z, 'cam', transl(0.2, 0, 0) );
```

To move the camera 20 cm upward, in the camera's negative y-direction

```
>> cam.mesh(X,Y,Z, 'cam', transl(0, -0.2, 0) );
```

and to rotate the camera about the z-axis

```
>> cam.mesh(X,Y,Z, 'cam', trotz(0.2) );
```

Next we wish to move the cube not the camera. An obvious approach is to generate another cube

```
>> [X,Y,Z] = mkcube(0.2, [0.2 0 1], 'edge');
```

where the centre has been moved in the x-direction, and viewing that

```
>> cam.mesh(X,Y,Z)
```

shows that the cube has moved to the right in the camera's field of view. Check that — as you move an object to the right it appears to your eyes to move to the right.

Just as for the case of camera motion above there is a simpler way of doing this in the toolbox. First we put the cube back to the origin

```
>> [X,Y,Z] = mkcube(0.2, [0 0 1], 'edge');
```

and then view it but with the object moved

```
>> cam.mesh(X,Y,Z, 'obj', transl(0.2, 0, 0) );
```

Q2 We will first create a cube at the origin and a camera object as in the example above

```
>> cam = CentralCamera('default');
```

and a smallish cube that is 20 cm on a side

```
>> [X,Y,Z] = mkcube(0.2, 'edge');
```

We will move the camera in a circular path of radius 1 m and with its z-axis pointing toward the origin. We achieve by first rotating the camera about its vertical- or y-axis and then 1 m in the negative z-direction. First let's see how this works

```
>> tranimate( trotz(0.2) );
>> tranimate( rotz(0.2), trotz(0.2)*trans(0,0,-1) );
```

Now we can create a loop to do this for all points around a circle

```
for theta=0:0.05:2*pi
  cam.mesh(X,Y,Z, 'cam', trotz(theta)*transl(0,0,-1) );
  pause(0.05);
end
```

Q3 Once again we create the camera and the cube

```
>> cam = CentralCamera('default');
```

and a smallish cube that is 20 cm on a side

```
>> [X,Y,Z] = mkcube(0.2, 'edge');
```

We will compute a camera trajectory from $(0,0,-5)$ to $(0,0,5)$ which will move right through the front face of the cube and out the back ¿¿ tg = ctraj(transl(0,0,-5), transl(0,0,5), 100); in 100 steps. To animate this we could use a loop for i=1:100 cam.mesh(X,Y,Z, 'cam', tg(:,:,i)); pause(0.05); end

Q6 The SI unit of solid angle, that is a 2-dimensional angle, is the steradian (abbreviation "sr"). One steradian is the angle subtended at the centre of a unit sphere by a unit area on the surface of a sphere. The maximum value is therefore 4π and corresponds

Solid angle is the 3-dimensional equaWe need to find the For a pyramid the solid angle is given by

$$\Omega = 4\sin^{-1}\left(\sin\frac{\theta_h}{2}\sin\frac{\theta_v}{2}\right)$$

Imagine a camera with horizontal field of view of 40° and a vertical field of view of 30°. Then the solid angle would be

```
>> deg = pi/180;
>> 4 * asin( sin(30*deg/2) * sin(40*deg/2) )
```

To a first approximation we can consider the pyramid to have unit height, in which case the width and height of the base would be

```
>> width = 2*tan(30*deg/2);
>> height = 2*tan(40*deg/2);
```

and the area of the base is

```
>> width*height
```

Chapter 12

Image Processing

Chapter 13

Feature extraction

Chapter 14

Multiple Views

Part V

Robotics and Vision

Chapter 15

Visual Servoing

Chapter 16

Advanced Visual Servoing

Bibliography

[1] M. Spong, S. Hutchinson, and M. Vidyasagar, *Robot modeling and control*. Wiley, 2006.