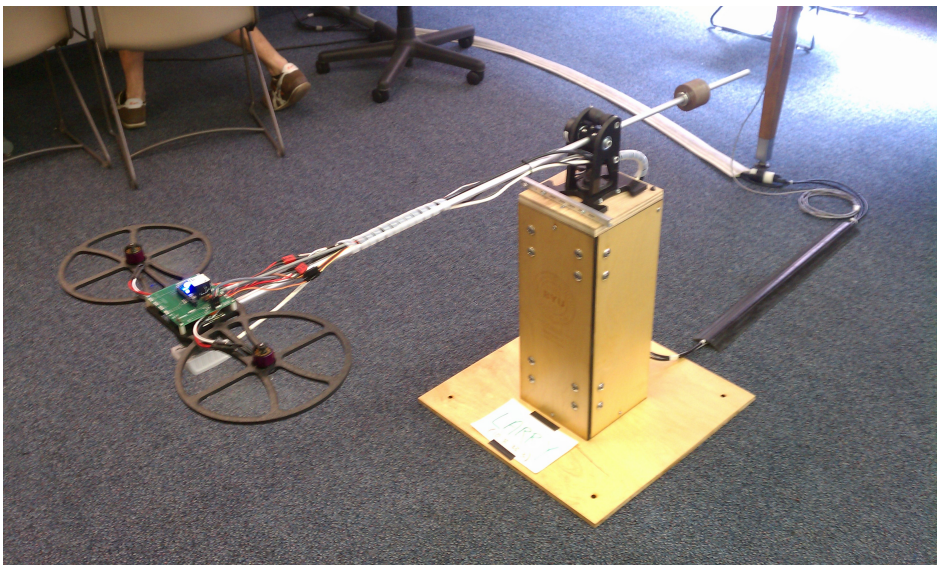
A Guide to the BYU Whirlybird
Version 3.0



Randal W. Beard, Tim McLain, Brandon Cannon, Everett Bryan
Brigham Young University

Updated: February 16, 2012

Contents

Chapter 1

Introduction

This objective of this document is to provide a detailed introduction to the BYU whirlybird including laboratory assignments that are intended to complement an introductory course in feedback control. The whirlybird is a three degree-of-freedom helicopter with complicated nonlinear dynamics, but the dynamics are amenable to linear analysis and design. The whirlybird has been designed with encoders on each joint that allow full-state feedback, but it is also equipped with an IMU (rate gyros and accelerometers) and a downward looking camera that resemble the types of sensors that will be on a real aerial robot. The sensor configuration allows estimated state information obtained from the IMU and camera to be compared to truth data obtained from the encoders, thereby decoupling the state feedback and state estimation problems, and allowing each to be tuned and analyzed independently. We believe that this decoupling is essential to help students understand the subtleties in feedback and estimator design.

The design process for control system is shown in Figure 1.1. For the whirlybird, the system to be controlled is a physical system with actuators (motors) and sensors (encoders, IMU, camera). The first step in the design process is to model the physical system using nonlinear differential equations. While approximations and simplifications will be necessary at this step, the hope is to capture in mathematics, all of the important characteristics of the physical system. The mathematical model of the system is usually obtained using Euler-Lagrange method, Newton's method, or other methods from physics and chemistry. The resulting model is usually high order and too complex for design. However, this model will be used to test later designs in simulation and is therefore called the Simulation Model, as shown in Fig-

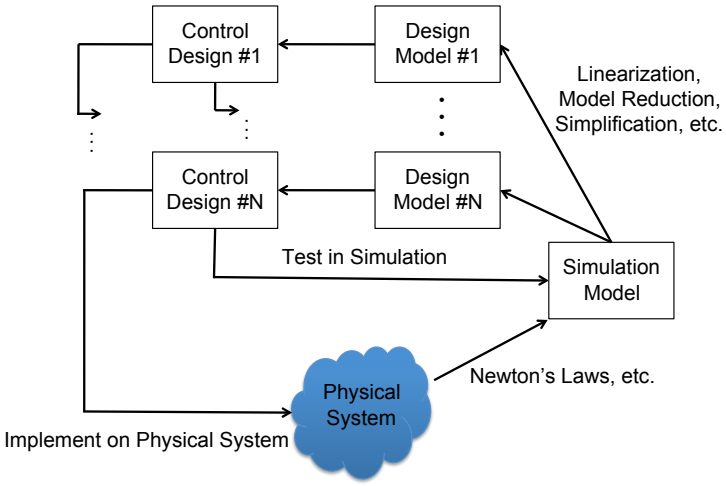


Figure 1.1: The design process. Using physics models the physical system is modeled with nonlinear differential equations resulting in the simulation model. The simulation model is simplified to create design models which are used for the control design. The control design is then tested and debugged in simulation and finally implemented on the physical system.

ure 1.1. To facilitate design, the simulation model is simplified and usually linearized to create lower-order (and usually linear) design models. For any physical system, there may be multiple design models that capture certain aspects of the design process. For the whirlybird, we will decompose the motion into longitudinal (pitching) motion and lateral (rolling and yawing) motion and we will have different design models for each type of motion. As shown in Figure 1.1, the design models are used to develop control designs. The control designs are then tested against the simulation, which sometimes requires that the design models be modified or enhanced if they have not captured the essential features of the system. After the control designs have been thoroughly tested in simulation, they are implemented on the physical system and are again tested and debugged, sometimes requiring that the simulation model be modified to more closely match the physical system.

The purpose of these notes, and the associated laboratory assignments, is to walk you through the design cycle for the whirlybird. We will consider three different design methodologies: successive loop closure, loopshaping, and observer-based control. The simulation model will be implemented in Matlab and Simulink. We believe that visualizing the physical system is an

important part of the simulation model. Therefore the first lab assignment described in Chapter 2 will describe how to draw animations in Simulink. Implementing complicated mathematical models in Simulink requires knowledge of s-functions, which are typically unfamiliar to students. Therefore the second lab assignment described in Chapter 3 shows how to create s-functions. A simulation model for the whirlybird is derived in Chapter 4 using the Euler-Lagrange method, and the associated lab is to implement this model in a Simulink s-function. The fourth lab, described in Chapter 5, is to develop the design models that will be used in subsequent labs. Developing physically realizable design specifications is an important part of the design process, and so the fifth lab assignment, described in Chapter 6 is to develop appropriate design specifications. The final three labs are to design implement controllers for the whirlybird using three different design techniques. These labs can be done in any order. In Chapter 7, successive loop closure using PID is used to control the whirlybird. In Chapter 8, the loopshaping method is used, and Chapter 9 walks the student through an observer-based design, where an extended Kalman filter is used to track a target in the camera plane.

Chapter 2

Animations in Simulink

We believe that visualizing the control design in the simulation stage helps the designer to develop physical intuition for the system and is an important debugging tool. Therefore, the first step in developing a simulation model for the whirlybird will be to create an animation in Matlab/Simulink. Students unfamiliar with Matlab and/or Simulink should read the `Getting Started` sections of the Matlab and Simulink documentation, which can be found by typing `helpdesk` at the Matlab prompt.

2.1 Handle Graphics in Matlab

When a graphics function like `plot` is called in Matlab, the function returns a *handle* to the plot. A graphics handle is similar to a pointer in C/C++ in the sense that all of the properties of the plot can be accessed through the handle. For example, the Matlab command

```
>> plot_handle=plot(t,sin(t))
```

returns a pointer, or handle, to the plot of `sin(t)`. Properties of the plot can be changed by using the handle, rather than reissuing the `plot` command. For example, the Matlab command

```
>> set(plot_handle,'YData',cos(t))
```

changes the plot to `cos(t)`, without redrawing the axes, title, label, and other objects that may be associated with the plot. If the plot contains

drawings of several objects, then a handle can be associated with each object. For example,

```
>> plot_handle1 = plot(t,sin(t))
>> hold on
>> plot_handle2 = plot(t,cos(t))
```

draws both `sin(t)` and `cos(t)` on the same plot, with a handle associated with each object. The objects can be manipulated separately without redrawing the other object. For example, to change `cos(t)` to `cos(2t)`, issue the command

```
>> set(plot_handle2,'YData',cos(2*t))
```

We can exploit this property to animate simulations in Simulink by only redrawing the parts of the animation that change in time, and thereby significantly reducing the simulation time. To show how handle graphics can be used to produce animations in Simulink, we will provide three detailed examples. In Section 2.2 we illustrate a 2D animation of an inverted pendulum using the fill command. In Section 2.3 we illustrate a 3D animation of a spacecraft using lines to produce a stick figure. In Section 2.4 we modify the spacecraft animation to use the vertices-faces data construction in Matlab.

2.2 Animation Example: Inverted Pendulum

Consider the image of the inverted pendulum shown in Figure 2.1, where the configuration is completely specified by the position of the cart y, and the angle of the rod from vertical θ. The physical parameters of the system are the rod length L, the base width w, the base height h, and the gap between the base and the track g. The first step in developing the animation is to determine the position of points that define the animation. For example, for the inverted pendulum in Figure 2.1, the four corners of the base are

$$(y + w/2, g), (y + w/2, g + h), (y - w/2, g + h), (y - w/2, g),$$

and the two ends of the rod are given by

$$(y, g + h), (y + L\sin\theta, g + h + L\cos\theta).$$

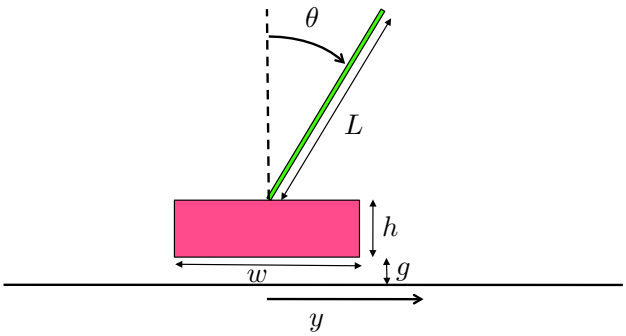


Figure 2.1: Drawing for inverted pendulum. The first step in developing an animation is to draw a figure of the object to be animated and identify all of the physical parameters.

Since the base and the rod can move independently, each will need its own figure handle. The `drawBase` command can be implemented with the following Matlab code:

```
1    function handle = drawBase(y,width,height,gap,handle,mode)
2      X = [y-width/2, y+width/2, y+width/2, y-width/2];
3      Y = [gap, gap, gap+height, gap+height];
4      if isempty(handle),
5        handle = fill(X,Y,'m','EraseMode',mode);
6      else
7        set(handle,'XData',X,'YData',Y);
8      end
```

Lines 2 and 3 define the X and Y locations of the corners of the base. Note that in Line 1, `handle` is both an input and an output. If an empty array is passed into the function, then the `fill` command is used to plot the base in Line 5. On the other hand, if a valid handle is passed into the function, then the base is redrawn using the `set` command in Line 7.

The Matlab code for drawing the rod is similar and is listed below.

```
1    function handle = drawRod(y,theta,L,gap,height,handle,mode)
2      X = [y, y+L*sin(theta)];
3      Y = [gap+height, gap + height + L*cos(theta)];
4      if isempty(handle),
5        handle = plot(X,Y,'g','EraseMode',mode);
6      else
```

```
7         set(handle,'XData',X,'YData',Y);
8       end
```

The input `mode` is used to specify the `EraseMode` in Matlab. The `EraseMode` can be set to `normal`, `none`, `xor`, or `background`. A description of these different modes can be found by looking under `Image Properties` in the Matlab Helpdesk.

The main routine for the pendulum animation is listed below.

```
1     function drawPendulum(u)
2         % process inputs to function
3         y        = u(1);
4         theta    = u(2);
5         t        = u(3);
6
7         % drawing parameters
8         L = 1;
9         gap = 0.01;
10        width = 1.0;
11        height = 0.1;
12
13        % define persistent variables
14        persistent base_handle
15        persistent rod_handle
16
17        % first time function is called, initialize plot and persistent vars
18        if t==0,
19            figure(1), clf
20            track_width=3;
21            plot([-track_width,track_width],[0,0],'k'); % plot track
22            hold on
23            base_handle = drawBase(y,width,height,gap,[],'normal');
24            rod_handle  = drawRod(y,theta,L,gap,height,[],'normal');
25            axis([-track_width,track_width,-L,2*track_width-L]);
26
27        % at every other time step, redraw base and rod
28        else
29            drawBase(y,width,height,gap,base_handle);
30            drawRod(y,theta,L,gap,height,rod_handle);
31        end
```

The routine `drawPendulum` is called from the Simulink file shown in Figure 2.2, where there are three inputs: the position y, the angle θ, and the time t. Lines 3-5 rename the inputs to y, θ, and t. Lines 8-11 define the

drawing parameters. We require that the handle graphics persist between function calls to `drawPendulum`. Since a handle is needed for both the base and the rod, we define two persistent variables in Lines 14 and 15. The `if` statement in Lines 18-31 is used to produce the animation. Lines 19–25 are called once at the beginning of the simulation, and draw the initial animation. Line 19 brings the figure 1 window to the front, and clears it. Lines 20 and 21 draw the ground along which the pendulum will move. Line 23 calls the `drawBase` routine with an empty handle as input, and returns the handle `base_handle` to the base. The `EraseMode` is set to `normal`. Line 24 calls the `drawRod` routine, and Line 25 sets the axes of the figure. After the initial time step, all that needs to be changed are the locations of the base and rod. Therefore, in Lines 29 and 30, the `drawBase` and `drawRod` routines are called with the figure handles as inputs.

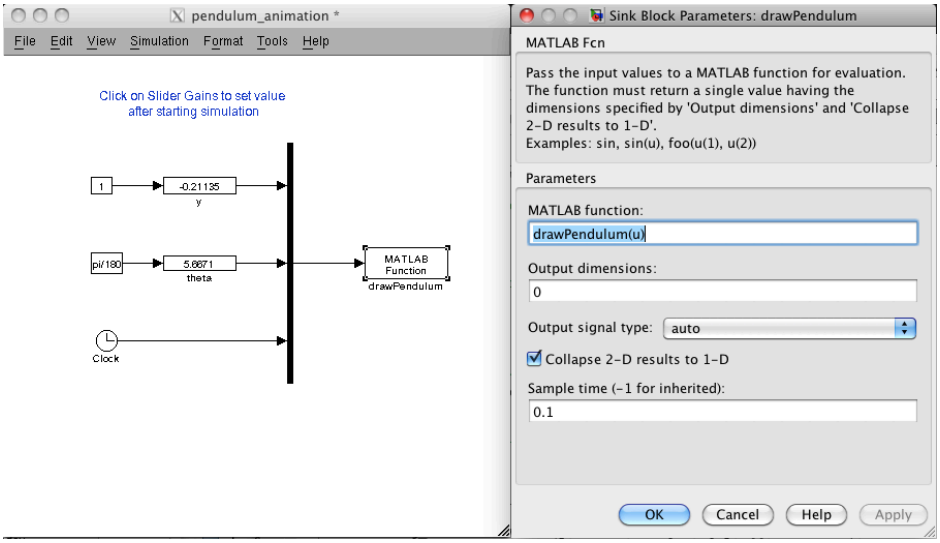


Figure 2.2: Simulink file for debugging the pendulum simulation. There are three inputs to the matlab m-file `drawPendulum`: the position y, the angle θ, and the time t. Slider gains for y and θ are used to verify the animation.

2.3 Animation Example: Spacecraft Using Lines

The previous section described a simple 2D animation. In this section we discuss a 3D animation of a spacecraft with six degrees of freedom. Figure 2.3

shows a simple line drawing of a spacecraft, where the bottom is meant to denote a solar panel that should be oriented toward the sun.

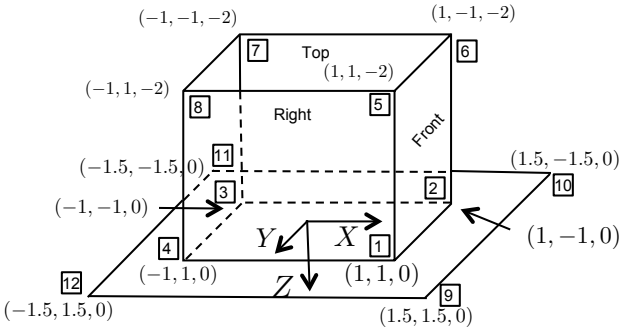


Figure 2.3: Drawing used to create spacecraft animation. Standard aeronautics body axes are used, where the x-axis points out the front of the spacecraft, the y-axis points to the right, and the z-axis point out the bottom of the body.

The first step in the animation process is to label the points on the spacecraft and to determine their coordinates in a body fixed coordinate system. We will use standard aeronautics axes with X pointing out the front of the spacecraft, Y pointing to the right, and Z pointing out the bottom. The points 1 through 12 are labeled in Figure 2.3 and specific coordinates are assigned to each label. To create a line drawing we need to connect the points in a way that draws each of the desired line segments. To do this as one continuous line, some of the segments will need to be repeated. To draw the spacecraft shown in Figure 2.3 we will transition through the following nodes $1-2-3-4-1-5-6-2-6-7-3-7-8-4-8-5-1-9-10-2-10-11-3-11-12-4-12-9$ Matlab code that defines the local coordinates of the spacecraft is given below.

```
function XYZ=spacecraftPoints;
  % define points on the spacecraft in local NED coordinates
  XYZ = [...
     1    1    0;... % point 1
     1   -1    0;... % point 2
    -1   -1    0;... % point 3
    -1    1    0;... % point 4
     1    1    0;... % point 1
     1    1   -2;... % point 5
```

```
          1   -1  -2;... % point 6
          1   -1   0;... % point 2
          1   -1  -2;... % point 6
         -1   -1  -2;... % point 7
         -1   -1   0;... % point 3
         -1   -1  -2;... % point 7
         -1    1  -2;... % point 8
         -1    1   0;... % point 4
         -1    1  -2;... % point 8
          1    1  -2;... % point 5
          1    1   0;... % point 1
          1.5  1.5 0;... % point 9
          1.5 -1.5 0;... % point 10
          1   -1   0;... % point 2
          1.5 -1.5 0;... % point 10
         -1.5 -1.5 0;... % point 11
         -1   -1   0;... % point 3
         -1.5 -1.5 0;... % point 11
         -1.5  1.5 0;... % point 12
         -1    1   0;... % point 4
         -1.5  1.5 0;... % point 12
          1.5  1.5 0;... % point 9
          ]';
```

The configuration of the spacecraft is given by the Euler angles ϕ, θ, and ψ, which represent the roll, pitch, and yaw angles respectively, and p_n, p_e, p_d, which represent the North, East, and Down positions respectively. The points on the spacecraft can be rotated and translated using the Matlab code listed below.

```
function XYZ=rotateBody(XYZ,phi,theta,psi);
  % define rotation matrix
  R_roll = [...
          1, 0, 0;...
          0, cos(phi), -sin(phi);...
          0, sin(phi), cos(phi)];
  R_pitch = [...
          cos(theta), 0, sin(theta);...
          0, 1, 0;...
          -sin(theta), 0, cos(theta)];
  R_yaw = [...
          cos(psi), -sin(psi), 0;...
          sin(psi), cos(psi), 0;...
          0, 0, 1];
```

```
15        R = R_yaw*R_pitch*R_roll;
16        % rotate vertices
17        XYZ = R*XYZ;
```

```
1     function XYZ = translateBody(XYZ,pn,pe,pd)
2       XYZ = XYZ + repmat([pn;pe;pd],1,size(XYZ,2));
```

Notice the order of the rotations performed. First the spacecraft is rolled about the North axis by the angle ϕ. Next it is pitched about the East axis by the angle θ. Finally, it is yawed about the Down axis by the angle ψ. Drawing the spacecraft at the desired location is accomplished using the following Matlab code.

```
1     function handle = drawSpacecraftBody(pn,pe,pd,phi,theta,psi,handle,mode)
2       % define points on spacecraft in local NED coordinates
3       NED = spacecraftPoints;
4       % rotate spacecraft by phi, theta, psi
5       NED = rotateBody(NED,phi,theta,psi);
6       % translate spacecraft to [pn; pe; pd]
7       NED = translateBody(NED,pn,pe,pd);
8        % transform vertices from NED to XYZ (for matlab rendering)
9        R = [...
10           0, 1, 0;...
11           1, 0, 0;...
12           0, 0, -1;...
13           ];
14       XYZ = R*NED;
15      % plot spacecraft
16      if isempty(handle),
17        handle = plot3(XYZ(1,:),XYZ(2,:),XYZ(3,:),'EraseMode', mode);
18      else
19         set(handle,'XData',XYZ(1,:),'YData',XYZ(2,:),'ZData',XYZ(3,:));
20        drawnow
21      end
```

Lines 9–14 are used to transform the coordinates from the North-East-Down (NED) coordinate frame, to the drawing frame used by Matlab which has the x-axis to the viewers right, the y-axis into the screen, and the z-axis up. The `plot3` command is used in Line 17 to render the original drawing, and the `set` command is used to change the `XData`, `YData`, and `ZData` in Line 19. A rendering of the spacecraft is shown in Figure 2.4.

The disadvantage of implementing the animation using the function `spacecraftPoints`

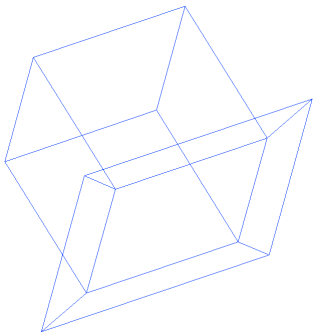


Figure 2.4: Rendering of the spacecraft using lines and the `plot3` command.

to define the spacecraft points, is that this function is called each time the animation is updated. Since the points are static, they only need to be defined once. The Simulink mask function can be used to define the points at the beginning of the simulation. Masking the `drawSpacecraft` m-file in Simulink, and then clicking on `Edit Mask` brings up a window like the one shown in Figure 2.5. The spacecraft points can be defined in the initialization window as shown in Figure 2.5 and passed to the `drawSpacecraft` m-file as a parameter.

2.4 Animation Example: Spacecraft using vertices and faces

The stick-figure drawing shown in Figure 2.4 can be improved visually by using the vertex-face structure in Matlab. Instead of using the `plot3` command to draw a continuous line, we will use the `patch` command to draw faces defined by vertices and colors. The vertices, faces, and colors for the spacecraft are defined in the Matlab code listed below.

```
1     function [V, F, patchcolors]=spacecraftVFC
2     % Define the vertices (physical location of vertices
3       V = [...
4           1     1     0;... % point 1
5           1    -1     0;... % point 2
6          -1    -1     0;... % point 3
7          -1     1     0;... % point 4
```

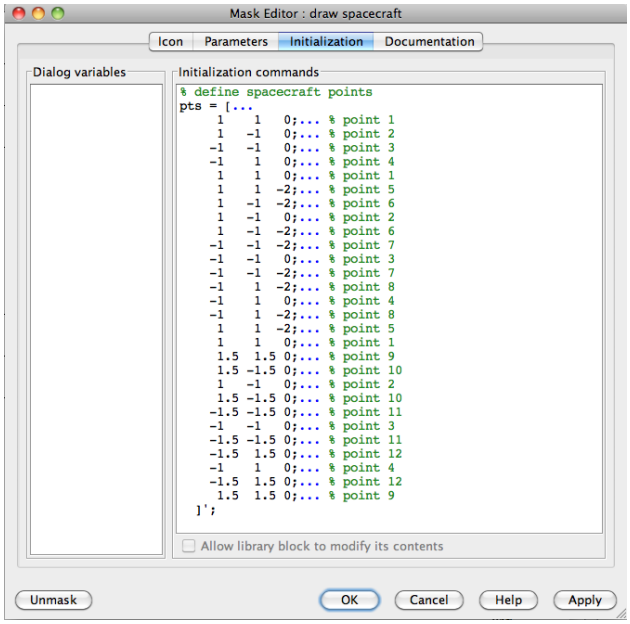


Figure 2.5: The mask function in Simulink allows the spacecraft points to be initialized at the beginning of the simulation.

```
8           1    1   -2;... % point 5
9           1   -1   -2;... % point 6
10         -1   -1   -2;... % point 7
11         -1    1   -2;... % point 8
12          1.5  1.5  0;... % point 9
13          1.5 -1.5  0;... % point 10
14         -1.5 -1.5  0;... % point 11
15         -1.5  1.5  0;... % point 12
16       ];
17     % define faces as a list of vertices numbered above
18       F = [...
19             1, 2,  6,  5;...  % front
20             4, 3,  7,  8;...  % back
21             1, 5,  8,  4;...  % right
22             2, 6,  7,  3;...  % left
23             5, 6,  7,  8;...  % top
24             9, 10, 11, 12;... % bottom
25             ];
26     % define colors for each face
27       myred    = [1, 0, 0];
28       mygreen  = [0, 1, 0];
```

```
29        myblue   = [0, 0, 1];
30        myyellow = [1, 1, 0];
31        mycyan   = [0, 1, 1];
32        patchcolors = [...
33          myred;...    % front
34          mygreen;...  % back
35          myblue;...   % right
36          myyellow;... % left
37          mycyan;...   % top
38          mycyan;...   % bottom
39          ];
```

The vertices are shown in Figure 2.3 and are defined in Lines 3–16. The faces are defined by listing the indices of the points that define the face. For example, the front face, defined in Line 19 consists of points $1-2-6-5$. Faces can be defined by N-points, where the matrix that defines the faces has N columns, and the number of rows is the number of faces. The color for each face is defined in Lines 32–39. Matlab code that draws the spacecraft body is listed below.

```
1     function handle = drawSpacecraftBody(pn,pe,pd,phi,theta,psi, handle, mode)
2       [V, F, patchcolors] = spacecraftVFC;  % define points on spacecraft
3       V = rotate(V', phi, theta, psi)'; % rotate spacecraft
4       V = translate(V', pn, pe, pd)'; % translate spacecraft
5       R = [...
6           0, 1, 0;...
7           1, 0, 0;...
8           0, 0, -1;...
9           ];
10      V = V*R;  % transform vertices from NED to XYZ (for matlab rendering)
11      if isempty(handle),
12      handle = patch('Vertices', V, 'Faces', F,...
13                     'FaceVertexCData',patchcolors,...
14                     'FaceColor','flat',...
15                     'EraseMode', mode);
16      else
17        set(handle,'Vertices',V,'Faces',F);
18      end
```

The transposes in Lines 3 and 4 are used because the physical positions in the vertices matrix V are along the rows instead of the columns. A rendering of the spacecraft using vertices and faces is given in Figure 2.6.

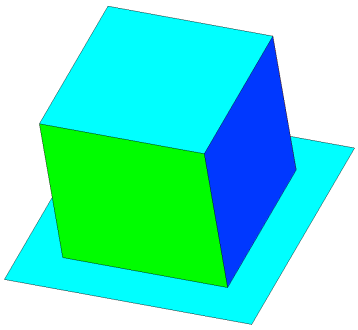


Figure 2.6: Rendering of the spacecraft using vertices and faces.

2.5 Lab 1: Assignment

The objective of this assignment is to create a 3D graphic of the whirlybird that is correctly rotated and translated to the desired configuration.

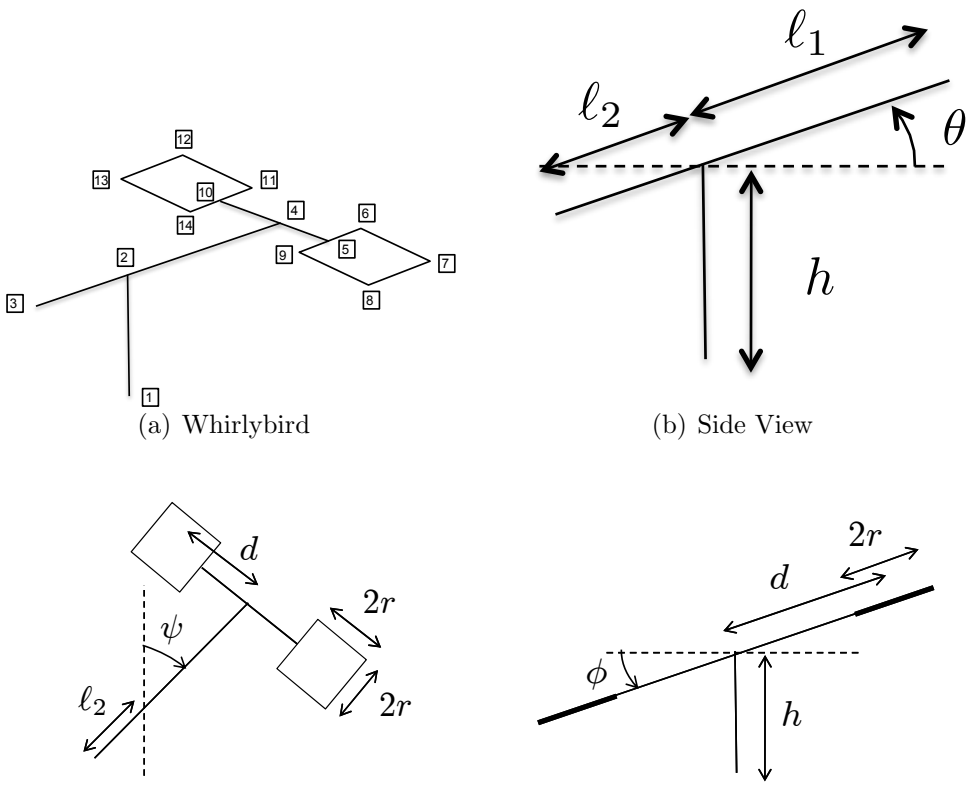
1. Create an animation drawing of the whirlybird shown in Figure 2.7. Use the parameters given Appendix A.

2. Create a Simulink model where the input to the animation file is the roll angle ϕ, the pitch angle θ, and the yaw angle ψ. Verify that the whirlybird is correctly rotated and translated in the animation. Hint: Point 2 in Figure 2.7(a) is the intersection of the three axes of rotation, ψ, θ, and ϕ. Making the origin of your NED reference frame coincide with point 2 simplifies the drawing the the whirlybird. Perform the rotations of the moving portions of the whirlybird in this order: (1) roll about North axis by ϕ, (2) pitch about East axis by θ, (3) yaw about Down axis by ψ .

3. Extra Credit: Improve the whirlybird animation so that it looks more realistic.

2.6 Lab 1: Hints

1. When drawing the whirlybird, it is important to correctly identify the orientation of the whirlybird axis. Pay close attention to Figure 2.3 and its explanation in the text to identify the whirlybird axis.



(a) Whirlybird

(b) Side View

(c) Top View

(d) Front View

Figure 2.7: Specifications for animation of whirlybird.

2. When using *handle graphics*, a separate handle must be defined for each object that moves independently. For example, the whirlybird base does not roll, pitch or yaw and therefore should have its own handle.

3. When using the *patch* function, the number of vertices that define a face must be 3 or greater and every face must have the same number of vertices. Faces with fewer vertices can be drawn by repeating the final point through the end of the row.

4. When using the *patch* function, the number faces must be equal to the number of face colors. If the number of faces differs from the number of face colors then object will not be drawn.

Chapter 3

S-functions in Simulink

This chapter assumes basic familiarity with the Matlab/Simulink environment. For additional information, please consult the Matlab/Simulink documentation. Simulink is essentially a sophisticated tool for solving interconnected hybrid ordinary differential equations and difference equations. Each block in Simulink has the structure

$$\dot{x}_c = f(t, x_c, x_d, u); \qquad x_c(0) = x_{c0} \tag{3.1}$$

$$x_d[k+1] = g(t, x_c, x_d, u); \qquad x_d[0] = x_{d0} \tag{3.2}$$

$$y = h(t, x_c, x_d, u) \tag{3.3}$$

where $x_c \in \mathbb{R}^{n_c}$ is a continuous state with initial condition x_{c0}, $x_d \in \mathbb{R}^{n_d}$ is a discrete state with initial condition x_{d0}, $u \in \mathbb{R}^m$ is the input to the block, $y \in \mathbb{R}^p$ is the output of the block, and t is the elapsed simulation time. An *s-function* is a Simulink tool for explicitly defining the functions f, g, and h and the initial conditions x_{c0} and x_{d0}. As explained in the Matlab/Simulink documentation, there are a number of methods for specifying an s-function. In this Chapter, we will overview two different methods: the level-1 m-file s-function, and a C-file s-function. The C-file s-function is compiled into C-code and executes much faster than m-file s-functions.

We will show how to implement the system specified by the standard second order transfer function

$$Y(s) = \frac{\omega_n^2}{s^2 + 2\zeta\omega_n s + \omega_n^2} U(s). \tag{3.4}$$

The first step is either case, is to represent (3.4) in state space form. Using

control canonical form we have

$$\begin{pmatrix}\dot{x}_1\\ \dot{x}_2\end{pmatrix} = \begin{pmatrix}-2\zeta\omega_n & -\omega_n^2\\ 1 & 0\end{pmatrix}\begin{pmatrix}x_1\\ x_2\end{pmatrix} + \begin{pmatrix}1\\ 0\end{pmatrix} u \tag{3.5}$$

$$y = \begin{pmatrix}0 & \omega_n^2\end{pmatrix}\begin{pmatrix}x_1\\ x_2\end{pmatrix}. \tag{3.6}$$

3.1 Level-1 M-file S-function

The code listing for an m-file s-function that implements system (3.5) and (3.6) is shown below. Line 1 defines the main m-file function. The inputs to this function are always the elapsed time t, the state x, which is a concatenation of the continuous state and discrete state, the input u, a `flag`, followed by user defined input parameters, which in this case are ζ and ω_n. The Simulink engine calls the s-function and passes the parameters t, x, u, and `flag`. When `flag==0`, the Simulink engine expects the s-function to return the structure `sys` which defines the block, initial conditions `x0`, an empty string `str`, and an array `ts` that defines the sample times of the block. When `flag==1` the Simulink engine expect the s-function to return the function $f(t, x, u)$, when `flag==2` the Simulink engine expects the s-function to return $g(t, x, t)$, and when `flag==3` it expects the s-function to return $h(t, x, u)$. The switch statement that calls the proper functions based on the value of `flag` is shown in Lines 2–11. The block setup and the definition of the initial conditions is shown in Lines 13–27. The number of continuous states, discrete states, outputs, and inputs are defined in Lines 16-19, respectively. The direct feedthrough term on Line 20 is set to one if the output depends explicitly on the input u: for example, if $D \neq 0$ in the linear state space output equation $y = Cx + Du$. The initial conditions are defined on Line 24. The sample times are defined on Line 27. The format for this line is `ts = [period offset]`, where `period` defines the sample period, and is 0 for continuous time, or -1 for inherited, and where `offset` is the sample time offset, which is typically 0. The function $f(t, x, u)$ is defined in Lines 30–32, and the output function $h(t, x, u)$ is defined in Lines 35–36.

```
1     function [sys,x0,str,ts] = second_order_m(t,x,u,flag,zeta,wn)
2         switch flag,
3         case 0,
4             [sys,x0,str,ts]=mdlInitializeSizes;  % initialize block
```

```
5           case 1,
6               sys=mdlDerivatives(t,x,u,zeta,wn);  % define xdot = f(t,x,u)
7           case 3,
8               sys=mdlOutputs(t,x,u,wn);           % define xup = g(t,x,u)
9           otherwise,
10              sys = [];
11          end
12
13       %=============================================================================
14      function [sys,x0,str,ts]=mdlInitializeSizes
15          sizes = simsizes;
16          sizes.NumContStates  = 2;
17          sizes.NumDiscStates  = 0;
18          sizes.NumOutputs     = 1;
19          sizes.NumInputs      = 1;
20          sizes.DirFeedthrough = 0;
21          sizes.NumSampleTimes = 1;     % at least one sample time is needed
22          sys = simsizes(sizes);
23
24          x0  = [0; 0];  %  define initial conditions
25          str = [];  % str is always an empty matrix
26          % initialize the array of sample times
27          ts  = [0 0];      % continuous sample time
28
29       %=============================================================================
30      function xdot=mdlDerivatives(t,x,u,zeta,wn)
31          xdot(1) = -2*zeta*wn*x(1) - wn^2*x(2) + u;
32          xdot(2) = x(1);
33
34       %=============================================================================
35      function y=mdlOutputs(t,x,u,wn)
36          y = wn^2*x(2);
```

3.2 C-file S-function

The code listing for a C-file s-function that implements system (3.5) and (3.6) is shown below. The function name must be specified as in Line 3. The number of parameters that are passed to the s-function is specified in Line 17, and macros that access the parameters are defined in lines 6 and 7. Line 8 defines a macro that allows easy access to the input of the block. The block structure is defined using `mdlInitializeSizes` in Line 15–36. The number of continuous states, discrete states, inputs, and outputs is defined in

Lines 21–27. The sample time and offset are specified in Lines 41–46. The initial conditions for the states are specified in Lines 52–57. The function $f(t, x, u)$ is defined in Lines 76–85, and the function $h(t, x, u)$ is defined in Lines 62–69. The C-file s-function is compiled using the matlab command `>> mex secondOrder_c.c`.

```
1   /*  File    : secondOrder_c.c
2    */
3   #define S_FUNCTION_NAME secondOrder_c
4   #define S_FUNCTION_LEVEL 2
5   #include "simstruc.h"
6   #define zeta_PARAM(S) mxGetPr(ssGetSFcnParam(S,0))
7   #define wn_PARAM(S)   mxGetPr(ssGetSFcnParam(S,1))
8   #define U(element) (*uPtrs[element])  /* Pointer to Input Port0 */
9
10  /* Function: mdlInitializeSizes
11   * Abstract:
12   *    The sizes information is used by Simulink to determine the S-function
13   *    blocks characteristics (number of inputs, outputs, states, etc.).
14   */
15  static void mdlInitializeSizes(SimStruct *S)
16  {
17      ssSetNumSFcnParams(S, 2);  /* Number of expected parameters */
18      if (ssGetNumSFcnParams(S) != ssGetSFcnParamsCount(S)) {
19          return; /* Parameter mismatch will be reported by Simulink */
20      }
21      ssSetNumContStates(S, 2);
22      ssSetNumDiscStates(S, 0);
23      if (!ssSetNumInputPorts(S, 1)) return;
24      ssSetInputPortWidth(S, 0, 1);
25      ssSetInputPortDirectFeedThrough(S, 0, 1);
26      if (!ssSetNumOutputPorts(S, 1)) return;
27      ssSetOutputPortWidth(S, 0, 1);
28      ssSetNumSampleTimes(S, 1);
29      ssSetNumRWork(S, 0);
30      ssSetNumIWork(S, 0);
31      ssSetNumPWork(S, 0);
32      ssSetNumModes(S, 0);
33      ssSetNumNonsampledZCs(S, 0);
34      /* Take care when specifying exception free code - see sfuntmpl_doc.c */
35      ssSetOptions(S, SS_OPTION_EXCEPTION_FREE_CODE);
36  }
37
38  /* Function: mdlInitializeSampleTimes
```

```
39      *    S-function is comprised of only continuous sample time elements
40      */
41     static void mdlInitializeSampleTimes(SimStruct *S)
42     {
43         ssSetSampleTime(S, 0, CONTINUOUS_SAMPLE_TIME);
44         ssSetOffsetTime(S, 0, 0.0);
45         ssSetModelReferenceSampleTimeDefaultInheritance(S);
46     }
47
48     #define MDL_INITIALIZE_CONDITIONS
49     /* Function: mdlInitializeConditions
50      *    Set initial conditions
51      */
52     static void mdlInitializeConditions(SimStruct *S)
53     {
54         real_T *x0 = ssGetContStates(S);
55         x0[0] = 0.0;
56         x0[1] = 0.0;
57     }
58
59     /* Function: mdlOutputs
60      *      output function
61      */
62     static void mdlOutputs(SimStruct *S, int_T tid)
63     {
64         real_T            *y    = ssGetOutputPortRealSignal(S,0);
65         real_T            *x    = ssGetContStates(S);
66         InputRealPtrsType uPtrs = ssGetInputPortRealSignalPtrs(S,0);
67         UNUSED_ARG(tid); /* not used in single tasking mode */
68         const real_T      *wn   = wn_PARAM(S);
69         y[0] = wn[0]*wn[0]*x[1];
70     }
71
72     #define MDL_DERIVATIVES
73     /* Function: mdlDerivatives
74      *      Calculate state-space derivatives
75      */
76     static void mdlDerivatives(SimStruct *S)
77     {
78         real_T            *dx   = ssGetdX(S);
79         real_T            *x    = ssGetContStates(S);
80         InputRealPtrsType uPtrs = ssGetInputPortRealSignalPtrs(S,0);
81         const real_T      *zeta = zeta_PARAM(S);
82         const real_T      *wn   = wn_PARAM(S);
83         dx[0] = -2*zeta[0]*wn[0]*x[0] - wn[0]*wn[0]*x[1] + U(0);
```

```
84          dx[1] = x[0];
85      }
86
87      /* Function: mdlTerminate
88       *    No termination needed, but we are required to have this routine.
89       */
90      static void mdlTerminate(SimStruct *S)
91      {
92          UNUSED_ARG(S); /* unused input argument */
93      }
94
95      #ifdef  MATLAB_MEX_FILE    /* Is this file being compiled as a MEX-file? */
96      #include "simulink.c"      /* MEX-file interface mechanism */
97      #else
98      #include "cg_sfun.h"       /* Code generation registration function */
99      #endif
```

3.3 Lab 2: Assignment

In the next chapter we will see that the equations of motion of the whirlybird, if the pitch angle is set to $\theta = 0$ are given by the equations

$$\ddot{\phi} = \frac{\dot{\psi}^2 \sin\phi\cos\phi(J_y - J_z) + d\,(f_l - f_r)}{J_x} \tag{3.7}$$

$$\ddot{\psi} = \frac{-2\dot{\psi}\dot{\phi}\sin\phi\cos\phi(J_y - J_z) + \ell_1\,(f_l + f_r)\sin\phi}{\left(m_1\ell_1^2 + m_2\ell_2^2 + J_y\sin^2\phi + J_z\cos^2\phi\right)}. \tag{3.8}$$

While we will not use these equations for the simulation model, but to give you practice in writing s-functions, in this lab we will implement these equations and connect the equations of motion to the animation that you created in Lab 1.

1. Rename your Simulation animation file `whirlysim.mdl`, and create a new a matlab file called `param.m`.

2. Enter all of the parameters from Appendix A into the parameter file as well as the feedback gain

$$K = \begin{pmatrix} 0.7216 & 0.1148 & 0.1 & 0.137 \end{pmatrix}.$$

3. Using Equation (3.7) and (3.8), and the equation

$$\ddot{\theta} = 0,$$

create a model of the whirlybird system using $u = f_l - f_r$ as the input and $x = (\phi, \dot{\phi}, \theta, \dot{\theta}, \psi, \dot{\psi})^T$ as the state and the output. Set the initial conditions to $\phi = 20$ deg, $\dot{\phi} = 0$ deg/s, $\theta = 0$ deg, $\dot{\theta} = 0$ deg/s, $\psi = 30$ deg, and $\dot{\psi} = 0$ deg/s. Assume that the sum of the left and right thrust forces is a constant value. To balance the weight of the whirlybird with the whirlybird in the horizontal plane, $f_l + f_r = 1.73\ N$. For simplicity, replace $f_l + f_r$ in Equation 3.8 with this value.

4. Add scopes to your simulation to plot each of the states and input torque.

5. Add a Matlab function that implements the feedback controller $u = -Kx$. Use a sample rate of $T_s = 0.01$. Simulate this controller to make sure that there is no motion on the whirlybird states.

6. Clean up the Simulink diagram so that it looks nice.

3.4 Lab 2: Hints

1. Your file called `param.m` should be a Matlab script. Matlab scripts do not have input or output arguments. Run the script before starting your simulink simulation. Type `help script` at the matlab prompt or view the Matlab online documentation for more information about scripts.

2. Remember to check the inputs of your function that plots the whirlybird. The input vector u was 3×1 for lab 1 and now it will be 7×1.

3. Inside your Simulink S-Function code you should be making changes to `mdlInitializeSizes`, `mdlDerivatives`, and `mdlOutputs`.

Chapter 4

Whirlybird Simulation Model

The objective of this Chapter is to develop the simulation model for the whirlybird using the Euler-Lagrange equation

$$\frac{d}{dt}\frac{\partial L}{\partial \dot{q}} - \frac{\partial L}{\partial q} = Q. \tag{4.1}$$

The variable L is called the Lagrangian, the variable q is a vector of generalized coordinates, and the variable Q is a vector of generalized forces.

The physical parameters of the whirlybird are given in Appendix A, where m_1 is the mass of the whirlybird head (including the rotors, IMU, and camera), m_2 is the mass of the counterweight, ℓ_1 is the length of the rod from the pivot point to the center of mass of the whirlybird head, ℓ_2 is the length of the rod from the pivot point to the counter weight, d is the distance from the center of mass of the whirlybird head to the motor, and h is the height of the pivot point above ground. We will assume that the counter weight is a point mass and that the rods are massless.

The unit vectors $\mathbf{i}_i$, $\mathbf{j}_i$, and $\mathbf{k}_i$ denote a right handed coordinate system centered at the pivot point and oriented so that $\mathbf{i}_i$ and $\mathbf{j}_i$ form a plane that is parallel to the ground, with $\mathbf{i}_i$ pointing along the rod and $\mathbf{j}_i$ pointing to the right when the pitch and yaw angles are both zero, and $\mathbf{k}_i$ pointing into the ground.

The yaw angle ψ is the positive rotation about $\mathbf{k}_i$, the pitch angle θ is the positive rotation about $\mathbf{j}_i$, and the roll angle ϕ is the positive rotation about $\mathbf{k}_i$. The forces exerted by the left and right motors are denoted as f_r and f_ℓ. A cartoon of the whirlybird is shown in Figure 4.1. To make the expressions easier to read, we will use the notation $c_\phi \triangleq \cos\phi$ and $s_\phi \triangleq \sin\phi$.

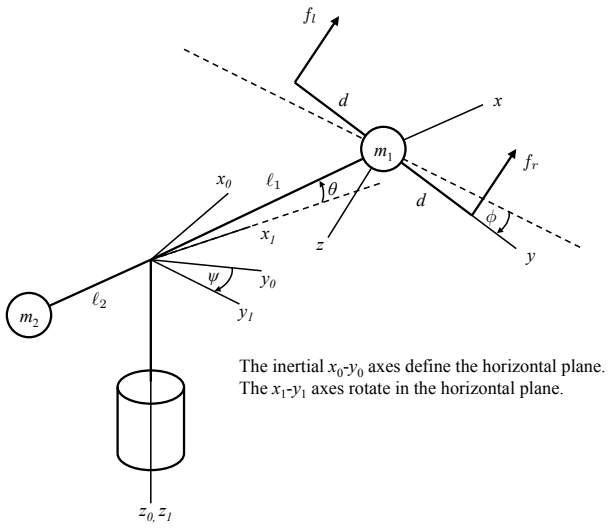


Figure 4.1: Whirlybird coordinate frames and nomenclature.

4.1 Kinetic and Potential Energy

The first step in the derivation of the equations of motion of the whirlybird is to find the total kinetic and potential energy of the system. The total kinetic energy of the system is given by

$$K = \sum_i \frac{1}{2} m_i \mathbf{v}_i^T \mathbf{v}_i + \frac{1}{2} \boldsymbol{\omega}_i^T \mathbf{J}_i \boldsymbol{\omega}_i,$$

where the sum is over all objects in the system, m_i is the mass of object i, $\mathbf{v}_i$ is the velocity of the center of mass of object i, $\boldsymbol{\omega}_i$ is the angular velocity of object i, and $\mathbf{J}_i$ is the inertia matrix of the i^{th} object. Since the velocity and angular velocity of the base are zero, and since we are assuming massless rods, they do not contribute to the kinetic energy of the system. Therefore, only the whirlybird head and the counterweight contribute to the potential energy. Since the counterweight is assumed to have a point mass, its inertia matrix is zero. Therefore, the kinetic energy of the whirlybird is given by

$$K = \frac{1}{2} m_1 \mathbf{v}_1^T \mathbf{v}_1 + \frac{1}{2} \boldsymbol{\omega}_1^T \mathbf{J}_1 \boldsymbol{\omega}_1 + \frac{1}{2} m_2 \mathbf{v}_2^T \mathbf{v}_2, \tag{4.2}$$

where $\mathbf{v}_1$, $\boldsymbol{\omega}_1$, and $\mathbf{J}_1$ are the velocity of the center of mass, angular velocity, and inertia of the whirlybird head respectively, and $\mathbf{v}_2$ is the velocity of the counter weight.

In the inertial frame, the position of the center of mass of the whirlybird head is given by

$$p_1 = R_z(\psi)R_y(\theta)\begin{pmatrix}\ell_1\\0\\0\end{pmatrix} = \begin{pmatrix}c_\psi & -s_\psi & 0\\ s_\psi & c_\psi & 0\\ 0 & 0 & 1\end{pmatrix}\begin{pmatrix}c_\theta & 0 & s_\theta\\ 0 & 1 & 0\\ -s_\theta & 0 & c_\theta\end{pmatrix}\begin{pmatrix}\ell_1\\0\\0\end{pmatrix} = \begin{pmatrix}\ell_1 c_\theta c_\psi\\ \ell_1 c_\theta s_\psi\\ -\ell_1 s_\theta\end{pmatrix}.$$

Similarly, the position of the counterweight is given by

$$p_2 = R_z(\psi)R_y(\theta)\begin{pmatrix}-\ell_2\\0\\0\end{pmatrix} = \begin{pmatrix}-\ell_2 c_\theta c_\psi\\ -\ell_2 c_\theta s_\psi\\ \ell_2 s_\theta\end{pmatrix}.$$

Differentiating with respect to time we get

$$\mathbf{v}_1 = \dot{\mathbf{p}}_1 = \begin{pmatrix}-\ell_1\dot\theta s_\theta c_\psi - \ell_1\dot\psi c_\theta s_\psi\\ -\ell_1\dot\theta s_\theta s_\psi + \ell_1\dot\psi c_\theta c_\psi\\ -\ell_1\dot\theta c_\theta\end{pmatrix}$$

$$\mathbf{v}_2 = \dot{\mathbf{p}}_2 = \begin{pmatrix}\ell_2\dot\theta s_\theta c_\psi + \ell_2\dot\psi c_\theta s_\psi\\ \ell_2\dot\theta s_\theta s_\psi - \ell_2\dot\psi c_\theta c_\psi\\ \ell_2\dot\theta c_\theta\end{pmatrix}.$$

The angular velocity of the whirlybird, expressed in the body frame, is

$$\boldsymbol{\omega} = \begin{pmatrix}p\\q\\r\end{pmatrix} = \begin{pmatrix}1 & 0 & -s_\theta\\ 0 & c_\phi & s_\phi c_\theta\\ 0 & -s_\phi & c_\phi c_\theta\end{pmatrix}\begin{pmatrix}\dot\phi\\\dot\theta\\\dot\psi\end{pmatrix} = \begin{pmatrix}\dot\phi - \dot\psi s_\theta\\ \dot\theta c_\phi + \dot\psi s_\phi c_\theta\\ -\dot\theta s_\phi + \dot\psi c_\phi c_\theta\end{pmatrix}.$$

The inertia matrix of the whirlybird head is given by

$$\mathbf{J} = \begin{pmatrix}J_x & 0 & 0\\ 0 & J_y & 0\\ 0 & 0 & J_z\end{pmatrix}.$$

Let $q = (\phi, \theta, \psi)^T$ and working out the expression for Equation (4.2) gives

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

where

$$M(q) \triangleq \begin{pmatrix}J_x & 0 & -J_x s_\theta\\ 0 & m_1\ell_1^2 + m_2\ell_2^2 + J_y c_\phi^2 + J_z s_\phi^2 & (J_y - J_z)s_\phi c_\phi c_\theta\\ -J_x s_\theta & (J_y - J_z)s_\phi c_\phi c_\theta & (m_1\ell_1^2 + m_2\ell_2^2 + J_y s_\phi^2 + J_z c_\phi^2)c_\theta^2 + J_x s_\theta^2\end{pmatrix}.$$

Since there are no springs in the system, the only source of potential energy is due to gravity. Therefore the potential energy is given by

$$P = m_1 g h_1 + m_2 g h_2 + P_0$$

where h_1 is the height of the center of gravity of the whirlybird head relative to zero pitch, h_2 is the height of the counterweight relative to zero pitch, and P_0 is the potential energy of the system at zero pitch. From geometry we have

$$P(q) = m_1 g \ell_1 s_\theta - m_2 g \ell_2 s_\theta + P_0.$$

4.2 Euler-Lagrange Equations

The Lagrangian is defined as

$$\begin{aligned} L &\triangleq K(q, \dot{q}) - P(q) \\ &= \frac{1}{2} \dot{q}^T M(q) \dot{q} - P(q). \end{aligned}$$

The generalized forces are

$$Q = \begin{pmatrix} d(f_l - f_r) \\ \ell_1 (f_l + f_r) c_\phi \\ \ell_1 (f_l + f_r) c_\theta s_\phi + d(f_r - f_l) s_\theta \end{pmatrix}.$$

Taking the partial of L with respect to q gives

$$\frac{\partial L}{\partial q} = \frac{1}{2} \begin{pmatrix} \dot{q}^T \frac{\partial M}{\partial q_1} \\ \dot{q}^T \frac{\partial M}{\partial q_2} \\ \dot{q}^T \frac{\partial M}{\partial q_3} \end{pmatrix} \dot{q} - \frac{\partial P}{\partial q}.$$

Also

$$\frac{d}{dt} \frac{\partial L}{\partial \dot{q}} = \frac{d}{dt} M(q) \dot{q} = M(q) \ddot{q} + \dot{M} \dot{q}.$$

Therefore, the Euler-Lagrange equation

$$\frac{d}{dt} \frac{\partial L}{\partial \dot{q}} - \frac{\partial L}{\partial q} = Q$$

gives

$$M(q)\ddot{q} + \left[\dot{M} - \frac{1}{2}\begin{pmatrix} \dot{q}^T \frac{\partial M}{\partial q_1} \\ \dot{q}^T \frac{\partial M}{\partial q_2} \\ \dot{q}^T \frac{\partial M}{\partial q_3} \end{pmatrix}\right]\dot{q} + \frac{\partial P}{\partial q} = Q.$$

If we define

$$c(q,\dot{q}) \triangleq \dot{M}\dot{q} - \frac{1}{2}\begin{pmatrix} \dot{q}^T \frac{\partial M}{\partial q_1} \\ \dot{q}^T \frac{\partial M}{\partial q_2} \\ \dot{q}^T \frac{\partial M}{\partial q_3} \end{pmatrix}\dot{q},$$

then the equations of motion are given by

$$M(q)\ddot{q} + c(q,\dot{q}) + \frac{\partial P}{\partial q} = Q.$$

Working through the math, we get

$$c(q,\dot{q}) = \begin{pmatrix} -\dot{\theta}^2(J_z - J_y)s_\phi c_\phi + \dot{\psi}^2(J_z - J_y)s_\phi c_\phi c_\theta^2 \\ -\dot{\theta}\dot{\psi}c_\theta\left[J_x - (J_z - J_y)(c_\phi^2 - s_\phi^2)\right] \\ --- \\ \dot{\psi}^2 s_\theta c_\theta\left[-J_x + m_1\ell_1^2 + m_2\ell_2^2 + J_y s_\phi^2 + J_z c_\phi^2\right] \\ -2\dot{\phi}\dot{\theta}(J_z - J_y)s_\phi c_\phi - \dot{\phi}\dot{\psi}c_\theta\left[-J_x + (J_z - J_y)(c_\phi^2 - s_\phi^2)\right] \\ --- \\ \dot{\theta}^2(J_z - J_y)s_\phi c_\phi s_\theta - \dot{\phi}\dot{\theta}c_\theta\left[J_x + (J_z - J_y)(c_\phi^2 - s_\phi^2)\right] \\ -2\dot{\phi}\dot{\psi}(J_z - J_y)c_\theta^2 s_\phi c_\phi + 2\dot{\theta}\dot{\psi}s_\theta c_\theta\left[J_x - m_1\ell_1^2 - m_2\ell_2^2 - J_y s_\phi^2 - J_z c_\phi^2\right] \end{pmatrix},$$

and

$$\frac{\partial P}{\partial q} = \begin{pmatrix} 0 \\ (m_1\ell_1 - m_2\ell_2)gc_\theta \\ 0 \end{pmatrix}.$$

4.2.1 Summary

The equations of motion are

$$M(q)\ddot{q} + c(q,\dot{q}) + \frac{\partial P}{\partial q} = Q, \tag{4.3}$$

where

$$M(q) = \begin{pmatrix} J_x & 0 & -J_x s_\theta \\ 0 & m_1\ell_1^2 + m_2\ell_2^2 + J_y c_\phi^2 + J_z s_\phi^2 & (J_y - J_z)s_\phi c_\phi c_\theta \\ -J_x s_\theta & (J_y - J_z)s_\phi c_\phi c_\theta & (m_1\ell_1^2 + m_2\ell_2^2 + J_y s_\phi^2 + J_z c_\phi^2)c_\theta^2 + J_x s_\theta^2 \end{pmatrix},$$

$$c(q,\dot{q}) = \begin{pmatrix} -\dot{\theta}^2(J_z - J_y)s_\phi c_\phi + \dot{\psi}^2(J_z - J_y)s_\phi c_\phi c_\theta^2 \\ -\dot{\theta}\dot{\psi}c_\theta\left[J_x - (J_z - J_y)(c_\phi^2 - s_\phi^2)\right] \\ --- \\ \dot{\psi}^2 s_\theta c_\theta\left[-J_x + m_1\ell_1^2 + m_2\ell_2^2 + J_y s_\phi^2 + J_z c_\phi^2\right] \\ -2\dot{\phi}\dot{\theta}(J_z - J_y)s_\phi c_\phi - \dot{\phi}\dot{\psi}c_\theta\left[-J_x + (J_z - J_y)(c_\phi^2 - s_\phi^2)\right] \\ --- \\ \dot{\theta}^2(J_z - J_y)s_\phi c_\phi s_\theta - \dot{\phi}\dot{\theta}c_\theta\left[J_x + (J_z - J_y)(c_\phi^2 - s_\phi^2)\right] \\ -2\dot{\phi}\dot{\psi}(J_z - J_y)c_\theta^2 s_\phi c_\phi + 2\dot{\theta}\dot{\psi}s_\theta c_\theta\left[J_x - m_1\ell_1^2 - m_2\ell_2^2 - J_y s_\phi^2 - J_z c_\phi^2\right] \end{pmatrix},$$

$$\frac{\partial P}{\partial q} = \begin{pmatrix} 0 \\ (m_1\ell_1 - m_2\ell_2)gc_\theta \\ 0 \end{pmatrix},$$

and

$$Q = \begin{pmatrix} d(f_l - f_r) \\ \ell_1(f_l + f_r)c_\phi \\ \ell_1(f_l + f_r)c_\theta s_\phi + d(f_r - f_l)s_\theta \end{pmatrix}.$$

4.3 Model of the Motor-Propeller

In the previous sections, we modeled the motors as producing a force perpendicular to the whirlybird x-y plane. The input to the motors is actually a pulse-width-modulation (PWM) command that regulates the duty cycle of the current supplied to the motor. The pulse width modulation command is constrained to be in the range $[0, 100]$, where 0 represents zero duty cycle, or zero current supplied to the motor, and 100 represents full current supplied to the motor.

While there are dynamics in the internal workings of the motor, we will neglect these dynamics and model the relationship between PWM command u and force as

$$F = k_m u.$$

To experimentally find the value of k_m we apply an equal PWM signal to each motor and increase this signal until the force balances the whirlybird at constant pitch angle. In the equilibrium position the forces cancel and we have

$$\ell_1 F = m_1\ell_1 g - m_2\ell_2 g.$$

Setting $F = f_l + f_r = k_m(u_l + u_r)$, and solving for k_m we get

$$k_m = \frac{m_1\ell_1 g - m_2\ell_2 g}{\ell_1(u_l + u_r)}.$$

While PWM is the input to the system, it is easier to think in terms of torque and force applied to the whirlybird. The relationship between PWM commands and force and torque are given by

$$\begin{pmatrix}\tau \\ F\end{pmatrix} = k_m \begin{pmatrix}d & -d \\ 1 & 1\end{pmatrix} \begin{pmatrix}u_l \\ u_r\end{pmatrix}.$$

Inverting the matrix and solving for PWM command gives

$$u_l = \frac{1}{2k_m}\left(F + \frac{\tau}{d}\right) \tag{4.4}$$

$$u_r = \frac{1}{2k_m}\left(F - \frac{\tau}{d}\right). \tag{4.5}$$

4.4 Lab 3: Assignment

1. Optional: Carefully work the derivation of the equations of motion for the whirlybird by hand to verify that they are correct. Please send corrections to the instructor.

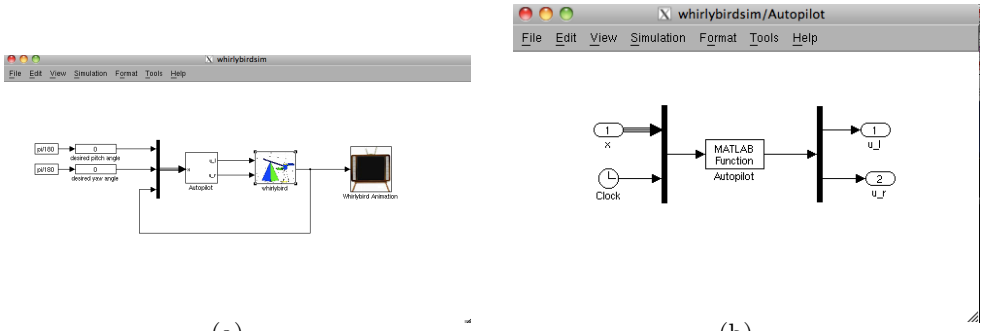
2. Modify the s-function from Lab 2 to implement the full equations of motion for the whirlybird. Let the state be given by $x = (\phi, \dot{\phi}, \theta, \dot{\theta}, \psi, \dot{\psi})^T$, and let the input be given by $u = (u_l, u_r)^T$.

3. Add your animation from Lab 1.

4. In hardware, the inputs to the motors will be pulse width modulation (PWM) comments, which we will denote by u_l, and u_r. The possible PWM signals are constrained to be between 0 and 100. Add if statements to the s-function that ensures that u_l and u_r are constrained between 0 and 100.

5. Add a matlab m-file called `autopilot.m` that will be used as the control block for the system. The output of the autopilot will be u_l and u_r. The inputs to the autopilot should be pitch command, yaw

command, and the full state of the system. See Figure 4.2. This autopilot block provides the framework to apply closed-loop feedback to our system. We are not quite ready for that yet. For the present, we will apply open-loop force and torque commands from within the autopilot block using Equations (4.4)–(4.5). Apply a torque of zero, and the right amount of force to cause the system to remain stationary when the initial conditions are $\phi(0) = 0$, $\theta(0) = 0$, and $\psi(0) = 0$.



(a) (b)

Figure 4.2: (a) Simulink model for the whirlybird system. (b) The control system is implemented in `autopilot.m` where the input is the system state x, and the current time t.

4.5 Lab 3: Hints

1.

Chapter 5

Whirlybird Design Models

The equations of motion developed in the previous chapter are too complex to be useful for control design. Therefore the objective of this chapter is to develop several different design models for the whirlybird that capture certain behaviors of the system. The essential idea is to notice that since the total force on the head of the whirlybird is $F = f_l + f_r$, the pitching motion can be controlled by manipulating F. Also, since the torque on the head of the whirlybird is $\tau = (f_l - f_r) * d$, the rolling motion can be controlled by manipulating τ. We also note that the yawing motion is driven primarily by the roll angle ϕ.

Accordingly, we define the *longitudinal* motion of the whirlybird, to be the motion involving the state variables $(\theta, \dot{\theta})$, and the *lateral* motion of the whirlybird to be the motion involving the state variables $(\phi, \dot{\phi}, \psi, \dot{\psi})$. While these motions are coupled, the coupling is weak and will be neglected in the design models. Design models for the longitudinal motion, assuming that $\phi = \dot{\phi} = \dot{\psi} = 0$ are derived in Section 5.1. Design models for the lateral motion, assuming that $\theta = \dot{\theta} = 0$ are derived in Section 5.2.

5.1 Design Models for the Longitudinal Dynamics

5.1.1 Nonlinear Model for Longitudinal Dynamics

In this section we will derive a nonlinear design model for the pitch, or longitudinal dynamics. The governing equation for θ is the second line of

Equation (4.3) which is

$$\begin{aligned}(m_1\ell_1^2 + m_2\ell_2^2 + J_y c_\phi^2 + J_z s_\phi^2)\ddot{\theta} + (J_y - J_z)s_\phi c_\phi c_\theta \ddot{\psi} \\ + \dot{\psi}^2 s_\theta c_\theta \left[-J_x + m_1\ell_1^2 + m_2\ell_2^2 + J_y s_\phi^2 + J_z c_\phi^2\right] \\ - 2\dot{\phi}\dot{\theta}(J_z - J_y)s_\phi c_\phi - \dot{\phi}\dot{\psi}c_\theta \left[-J_x + (J_z - J_y)(c_\phi^2 - s_\phi^2)\right] \\ + (m_1\ell_1 - m_2\ell_2)g c_\theta = \ell_1(f_l + f_r)c_\phi. \quad (5.1)\end{aligned}$$

Setting $\phi = \dot{\phi} = \ddot{\psi} = \dot{\psi} = 0$ gives

$$(m_1\ell_1^2 + m_2\ell_2^2 + J_y)\ddot{\theta} + (m_1\ell_1 - m_2\ell_2)g c_\theta = \ell_1(f_l + f_r).$$

Defining $F \triangleq f_l + f_r$ and solving for $\ddot{\theta}$ gives

$$\ddot{\theta} = \frac{(m_2\ell_2 - m_1\ell_1)g\cos\theta}{m_1\ell_1^2 + m_2\ell_2^2 + J_y} + \left(\frac{\ell_1}{m_1\ell_1^2 + m_2\ell_2^2 + J_y}\right)F, \quad (5.2)$$

which is the nonlinear design model for the longitudinal dynamics. In state-variable form we have

$$\dot{x}_{lon} = f_{lon}(x_{lon}, u_{lon}) \triangleq \begin{pmatrix} x_{lon}(2) \\ \frac{(m_2\ell_2 - m_1\ell_1)g\cos(x_{lon}(1))}{m_1\ell_1^2 + m_2\ell_2^2 + J_y} + \left(\frac{\ell_1}{m_1\ell_1^2 + m_2\ell_2^2 + J_y}\right)u_{lon} \end{pmatrix}, \quad (5.3)$$

where $x_{lon} = (\theta, \dot{\theta})^T$ and $u_{lon} = F$.

5.1.2 Linear State Space Model for Longitudinal Dynamics

The objective is to linearize the equations of motion around $\theta = \theta_e$, and the equilibrium force F_e Convince yourself that $\theta_e = 0$, $F_e = 0$ is not an equilibrium of the system. The first lab assignment is to find F_e so that θ_e is an equilibrium of the system. Defining the deviated state from equilibrium as

$$\tilde{x}_{lon} \triangleq (\theta, \dot{\theta})^T - (\theta_e, 0)^T$$

and the deviated control input from equilibrium as

$$\tilde{u}_{lon} = F - F_e,$$

then the linearization of Equation eq:(5.3) about the equilibrium is given by

$$\dot{\tilde{x}}_{lon} = A_{lon}\tilde{x}_{lon} + B_{lon}\tilde{u}_{lon},$$

where

$$A_{lon} = \frac{\partial f_{lon}}{\partial x_{lon}}(x_{lon,e}, F_e) = \begin{pmatrix} 0 & 1 \\ \frac{(m_1\ell_1 - m_2\ell_2)g\sin(\theta_e)}{m_1\ell_1^2 + m_2\ell_2^2 + J_y} & 0 \end{pmatrix}, \tag{5.4}$$

$$B_{lon} = \frac{\partial f_{lon}}{\partial u_{lon}}(x_{lon,e}, F_e) = \begin{pmatrix} 0 \\ \frac{\ell_1}{m_1\ell_1^2 + m_2\ell_2^2 + J_y} \end{pmatrix}. \tag{5.5}$$

If the output is $y_{lon} = \theta - \theta_e$, then the output equation is

$$y_{lon} = C_{lon}\tilde{x}_{lon},$$

where $C_{lon} = (1, 0)$.

Note that if we design a controller law $\tilde{u}_{lon}(\tilde{x}_{lon})$, then the commanded force will be

$$F = F_e + \tilde{u}_{lon}.$$

5.1.3 Linear Transfer Function Model for Longitudinal Dynamics

The transfer function from $\tilde{u}_{lon}$ to y_{lon} is given by

$$y_{lon}(s) = C_{lon}(sI - A_{lon})^{-1}B_{lon}\tilde{u}_{lon}(s) = \frac{\left(\frac{\ell_1}{m_1\ell_1^2 + m_2\ell_2^2 + J_y}\right)}{s^2 - \left(\frac{(m_1\ell_1 - m_2\ell_2)g\sin(\theta_e)}{m_1\ell_1^2 + m_2\ell_2^2 + J_y}\right)}\tilde{u}_{lon}(s).$$

If the desired equilibrium angle is $\theta_e = 0$, then the transfer function simplifies to

$$y_{lon}(s) = \frac{\left(\frac{\ell_1}{m_1\ell_1^2 + m_2\ell_2^2 + J_y}\right)}{s^2}\tilde{u}_{lon}(s). \tag{5.6}$$

5.2 Design Models for the Lateral Dynamics

5.2.1 Nonlinear Model for Lateral Dynamics

In this section we will derive a nonlinear design model for the side-to-side, or lateral dynamics. The governing equations for ϕ and ψ are the first and third lines of Equation (4.3) which, after setting $\theta = \dot{\theta} = \ddot{\theta} = 0$ are

$$J_x\ddot{\phi} + \dot{\psi}^2(J_y - J_z)s_\phi c_\phi = d(f_l - f_r)$$
$$(m_1\ell_1^2 + m_2\ell_2^2 + J_y s_\phi^2 + J_z c_\phi^2)\ddot{\psi} - 2\dot{\psi}\dot{\phi}(J_z - J_y)s_\phi c_\phi = \ell_1(f_l + f_r)s_\phi.$$

Solving for $\ddot{\phi}$ and $\ddot{\psi}$ gives

$$\ddot{\phi} = \frac{-\dot{\psi}^2(J_y - J_z)s_\phi c_\phi + d(f_l - f_r)}{J_x}$$
$$\ddot{\psi} = \frac{2\dot{\psi}\dot{\phi}(J_z - J_y)s_\phi c_\phi + \ell_1(f_l + f_r)s_\phi}{m_1\ell_1^2 + m_2\ell_2^2 + J_y s_\phi^2 + J_z c_\phi^2}.$$

To decouple the motion from the longitudinal states, we will assume that $f_l + f_r = F_e$, the equilibrium force required to maintain $\theta = \theta_e$. Defining the torque

$$\tau \triangleq d(f_l - f_r)$$

gives

$$\ddot{\phi} = \frac{-\dot{\psi}^2(J_y - J_z)s_\phi c_\phi + \tau}{J_x} \tag{5.7}$$
$$\ddot{\psi} = \frac{2\dot{\psi}\dot{\phi}(J_z - J_y)s_\phi c_\phi + \ell_1 F_e s_\phi}{m_1\ell_1^2 + m_2\ell_2^2 + J_y s_\phi^2 + J_z c_\phi^2}, \tag{5.8}$$

which are the nonlinear design models for the lateral dynamics. In state-variable form, we have

$$\dot{x}_{lat} = f_{lat}(x_{lat}, u_{lat}) \triangleq \begin{pmatrix} x_{lat}(2) \\ \frac{-x_{lat}(4)^2(J_y - J_z)\sin(x_{lat}(1))\cos(x_{lat}(1)) + u_{lat}}{J_x} \\ x_{lat}(4) \\ \frac{2x_{lat}(4)x_{lat}(2)(J_z - J_y)\sin(x_{lat}(1))\cos(x_{lat}(1)) + \ell_1 F_e \sin(x_{lat}(1))}{m_1\ell_1^2 + m_2\ell_2^2 + J_y\sin(x_{lat}(1))^2 + J_z\cos(x_{lat}(1))^2} \end{pmatrix},$$

where $x_{lat} = (\phi, \dot{\phi}, \psi, \dot{\psi})^T$, and $u_{lat} = \tau$. Note that f_{lat} is independent of the yaw angle ψ.

5.2.2 Linear State Space Model for Lateral Dynamics

Note that for the lateral dynamics, the equilibrium is given by $x_{lat,e} = (0, 0, 0, 0)$ and $u_{lat,e} = 0$. Therefore, to a first approximation, the lateral

dynamics are given by

$$\dot{x}_{lat} = A_{lat}x_{lat} + B_{lat}u_{lat}, \tag{5.9}$$

where

$$A_{lat} \triangleq \frac{\partial f_{lat}}{\partial x_{lat}}(x_{lat,e}, u_{lat,e}) = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ \frac{\ell_1 F_e}{m_1\ell_1^2+m_2\ell_2^2+J_z} & 0 & 0 & 0 \end{pmatrix} \tag{5.10}$$

$$B_{lat} \triangleq \frac{\partial f_{lat}}{\partial u_{lat}}(x_{lat,e}, u_{lat,e}) = \begin{pmatrix} 0 \\ \frac{1}{J_x} \\ 0 \\ 0 \end{pmatrix}. \tag{5.11}$$

Since both ϕ and ψ can be measured by the encoders, the output equation is two dimensional and is given by

$$y_{lat} = C_{lat}x_{lat}, \tag{5.12}$$

where

$$C_{lat} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}.$$

5.2.3 Linear Transfer Function Model for Lateral Dynamics

Taking the Laplace transform of the first two rows of Equation (5.9) gives

$$\phi(s) = \frac{(1/J_x)}{s^2}\tau(s). \tag{5.13}$$

Taking the Laplace transform of the second two rows of Equation (5.9) gives

$$\psi(s) = \frac{\left(\frac{\ell_1 F_e}{m_1\ell_1^2+m_2\ell_2^2+J_z}\right)}{s^2}\phi(s). \tag{5.14}$$

The structure of Equations (5.13) and (5.14) indicate that the linear equations for the lateral dynamics form a cascade structure as shown in Figure 5.1. As shown in Figure 5.1 the torque τ can be though of as the input to the roll

dynamics given by Equation (5.13). The roll angle ϕ which is the output of the roll dynamics can be thought of as the input to the yaw dynamics given by Equation (5.14). This cascade structure will be exploited in Chapter 7 and 8 where we will design control strategies for the roll dynamics and yaw dynamics separately and enforce suitable bandwidth separation to ensure a good design.

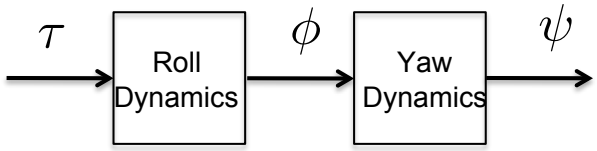


Figure 5.1: The lateral dynamics for the whirlybird can be thought of as a cascade system, where the roll angle drives the yaw dynamics, and the torque drives the roll dynamics.

5.3 Lab 4: Assignment

1. For the longitudinal state-variable equation given by Equation (5.3), find an equation for the equilibrium force F_e so that $x_{lon,e} = (\theta_e, 0)^T$ is an equilibrium of the system.
2. Derive the expressions for A_{lon} and B_{lon} given in Equations (5.4) and (5.5).
3. The lateral state space equations in Equation (5.9) and (5.12) are given in the original state x_{lat} and not a deviated state $\tilde{x}_{lat}$. Explain why this is appropriate for the lateral dynamics and not appropriate for the longitudinal dynamics.
4. Derive the expressions for A_{lat} and B_{lat} given in Equations (5.10) and (5.11).
5. Why does the cascade structure shown in Figure 5.1 makes sense physically.

5.4 Lab 4: Hints

1.

Chapter 6

Design Specifications and Limits of Performance

The desired performance of a closed-loop system is often given in terms of specifications like rise time, settling time, and maximum percent overshoot. In introductory courses like this one, these specifications are often given as part of the problem statement, i.e., design a feedback control law so that the rise time is less than 0.1 second, the settling time is less than 2 seconds and the maximum percent overshoot is less than 5%. However, for a given applications, these specifications must come from somewhere. Our objective in this lab is to develop a reasonable set of specifications for the whirlybird system.

The performance of a system is fundamentally limited by the amount of control effort available from the actuators in the system. In the case of the whirlybird, each of the propellor motors is limited by the maximum PWM command that can be supplied. In this section, we briefly describe how knowledge of the plant and controller transfer functions and the actuator saturation constraints can be used to develop performance specifications. We will use a second-order system to illustrate the process.

Given the second-order system shown in Figure 6.1 with proportional feedback on the output error and derivative feedback on the output, closed-loop transfer function is

$$\frac{y}{y^c} = \frac{b_0 k_p}{s^2 + (a_1 + b_0 k_d)s + (a_0 + b_0 k_p)}. \tag{6.1}$$

We can see that the closed-loop poles of the system are defined by the se-

lection of the control gains k_p and k_d. Further, we can see that the size of the actuator effort u is primarily governed by the sizes of the control error e, the error rate $\dot{e}$, and the control gains k_p and k_d. For a stable system with no transfer function zeros subjected to a step input command, the maximum value of the control effort will occur the instant that the step command is applied. For a maximum anticipated command input step size, we can calculate the calculate the maximum control effort as $u^{\max} = k_p(y^c - y(0))^{\max}$. If additional control loops are added (such as an integral control loop) that add zeros to the closed-loop transfer function, they may cause an increase in the maximum control effort. Under typical operating conditions, however, the actuator effort is dominated by the response of the proportional loop to large swings in the input command. Rearranging this expression, we find that the proportional control gain can be determined from the maximum anticipated output error and the saturation limits of the actuator as

$$k_p = \frac{u^{\max}}{(y^c - y(0))^{\max}}. \tag{6.2}$$

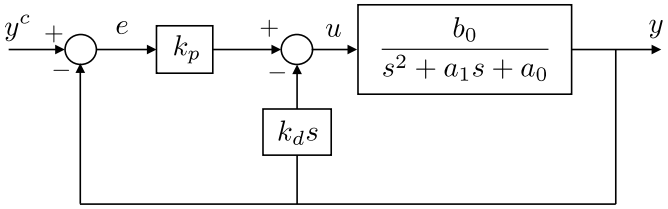


Figure 6.1: Control system example.

The canonical second-order transfer function with no zeros is given by the standard form

$$\frac{y}{y^c} = \frac{\omega_n^2}{s^2 + 2\zeta\omega_n s + \omega_n^2}, \tag{6.3}$$

where y^c is the commanded value, ζ is the damping ratio, and ω_n is the natural frequency. By comparing the coefficients of the denominator polynomials of the transfer function of the closed-loop system in Equation (6.1) and the canonical second-order system transfer function in Equation (6.3), and taking into account the saturation limits of the actuator, we can derive an expression for the achievable bandwidth and the rise time of the closed-loop

system. Equating the coefficients of the s^0 terms gives

$$\begin{aligned}\omega_n &= \sqrt{a_0 + b_0 k_p} \\ &= \sqrt{a_0 + b_0 \frac{u^{\max}}{(y^c - y(0))^{\max}}},\end{aligned}$$

which is an upper limit on the bandwidth of the closed-loop system ensuring that saturation of the actuator is avoided.

We have defined rise time to be the time it takes the system to transition from 10 percent to 90 percent of the steady-state value of the response. The rise time for typical values of the damping ratio is given by

$$t_r \approx \begin{cases} \frac{1.8}{\omega_n} & \zeta = 0.5 \\ \frac{2.2}{\omega_n} & \zeta = 0.7 \\ \frac{2.6}{\omega_n} & \zeta = 0.9 \end{cases}. \tag{6.4}$$

By specifying the damping ratio, we can determine the rise time that we would expect for a given system and saturation limit. For example, by specifying $\zeta = 0.7$, the rise time can be approximated as

$$\begin{aligned}t_r &\approx \frac{2.2}{\omega_n} \\ &\approx \frac{2.2}{\sqrt{a_0 + b_0 \frac{u^{\max}}{(y^c - y(0))^{\max}}}}.\end{aligned}$$

In the process of calculating the expected rise-time performance for the system, we calculated a value for the proportional gain k_p from the input step size and saturation limits as shown in Equation (6.2). We can calculate a suitable value for the derivative gain by equating the s^1 coefficients of the denominator polynomials of the transfer functions in Equations (6.1) and (6.3) to get

$$\begin{aligned}a_1 + b_0 k_d &= 2\zeta\omega_n \\ k_d &= \frac{2\zeta\omega_n - a_1}{b_0}.\end{aligned}$$

6.1 Example

The control block diagram of the pitch-axis proportional-derivative control is shown in Figure 6.2. From the block diagram, the transfer function for the

system is given by

$$y = \frac{b_0 k_p}{s^2 + b_0 k_d s + b_0 k_p} y^c + \frac{b_0}{s^2 + b_0 k_d s + b_0 k_p} F_e. \tag{6.5}$$

From Equation (5.6)

$$\begin{aligned} b_0 &= \frac{\ell_1}{m_1 \ell_1^2 + m_2 \ell_2^2 + J_y} \\ &= \frac{0.85 \text{ m}}{(0.535 \text{ kg})(0.85 \text{ m})^2 - (1 \text{ kg})(0.3048 \text{ m})^2 + 0.0014 \text{ kg-m}^2} \\ &= 1.7677 \text{ (kg-m}^{-1}\text{)}. \end{aligned}$$

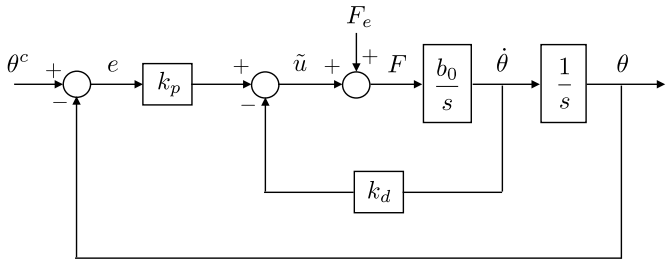


Figure 6.2: PD control design example.

To determine the expected natural frequency ω_n and rise time t_r, we must first determine the proportional gain k_p. To avoid saturation, we can select the proportional gain so that the maximum anticipated error $e_{\max}$ results in the maximum possible control effort $\tilde{u}_{\max}$. For a system driven by step inputs, the maximum error is the largest step size that the system will be given. From Figure 6.2, we get that

$$\tilde{u}_{\max} = k_p e_{\max}$$

at the instant that a step input is given. From this expression, we can solve for the proportional gain as

$$k_p = \frac{\tilde{u}_{\max}}{e_{\max}}.$$

The maximum control effort $\tilde{u}_{\max}$ in pitch is given by

$$\tilde{u}_{\max} = F_{\max} - F_e$$

where F_e is the equilibrium force required to balance the whirlybird in the equilibrium position and $F_{\max}$ is the maximum force available from the propellors.

The maximum force can be calculated as

$$\begin{aligned} F_{\max} &= k_m(u_{l,\max} + u_{r,\max}) \\ &= (0.0169 \text{ N/ct})(100 \text{ ct} + 100 \text{ ct}) \\ &= 3.38 \text{ N}. \end{aligned}$$

The equilibrium force can be calculated by performing a moment balance as

$$\begin{aligned} F_e &= (m_1\ell_1 - m_2\ell_2)\frac{g}{\ell_1} \\ &= [(0.535 \text{ kg})(0.85 \text{ m}) - (1 \text{ kg})(0.3048 \text{ m})]\,\frac{9.81 \text{ m/s}^2}{0.85 \text{ m}} \\ &= 1.73 \text{ N}. \end{aligned}$$

Thus,

$$\tilde{u}_{\max} = 1.65 \text{ N}.$$

Continuing on,

$$\begin{aligned} k_p &= \frac{1.65 \text{ N}}{(30 \text{ deg})\left(\frac{\pi \text{ rad}}{180 \text{ deg}}\right)} \\ &= 3.15\ \frac{\text{N}}{\text{rad}}. \end{aligned}$$

Comparing characteristic polynomial coefficients from Equations (6.3) and (6.5), we find that

$$\begin{aligned} \omega_n &= \sqrt{b_0 k_p} \\ &= \sqrt{\left(1.7677\ \tfrac{1}{\text{kg-m}}\right)\left(3.15\ \tfrac{\text{N}}{\text{rad}}\right)} \\ &= 2.36\ \frac{\text{rad}}{\text{s}} \end{aligned}$$

and

$$\begin{aligned} 2\zeta\omega_n &= b_0 k_d \\ k_d &= \frac{2\zeta\omega_n}{b_0}. \end{aligned}$$

Choosing $\zeta = 0.7$ to achieve a fast, but damped transient response, we get

$$\begin{aligned} k_d &= \frac{(2)(0.7)\left(2.36\ \frac{\text{rad}}{\text{s}}\right)}{1.7677\ \frac{1}{\text{kg-m}}} \\ &= 1.869\ \frac{\text{N-s}}{\text{rad}}. \end{aligned}$$

Finally, the corresponding rise time estimate is given by

$$\begin{aligned} t_r &\approx \frac{2.2}{\omega_n} \\ &\approx 0.93 \text{ s}. \end{aligned}$$

6.2 Full State Space Model

6.3 Lab 5: Assignment

1. Using the fact that the PWM command to the propellor motors is limited to $0 \leq PWM \leq 100$, develop specifications for the rise time of the whirlybird in roll and pitch. Use the physical parameters provided in Appendix A and the transfer functions derived in Lab 4 as the basis for your calculations. You will need to pick a reasonable step size for ϕ and θ (20 to 30 deg should be appropriate).

2. Calculate the proportional gains for the pitch and roll feedback loops to ensure that saturation does not occur for a step input of reasonable size. Based on your desired damping ratio, calculate derivative gains for the pitch and roll feedback loops.

3. For the pitch feedback loop, implement the control shown in Figure 6.1 using built-in Simulink blocks. Remember to add in the equilibrium force needed to hold the whirlybird in the horizontal position. Test the pitch feedback loop by giving it a variety of step inputs and verifying that it responds as expected.

4. For the roll feedback loop, implement the control shown in Figure 6.1 using built-in Simulink blocks. Command the pitch feedback loop to hold the pitch at 0 deg. Test the roll feedback loop by giving it a variety of step inputs and verifying that it responds as expected. Let the yaw angle move freely. We will implement yaw control in the next lab.

5. After implementing PD control for the pitch and roll loops and tuning them to verify their performance, add integral control to each loop. Use small integral gains to avoid degrading the transient response of the system.

6. **Bonus:** Instead of using a constant value for F_e in the control, implement a nonlinear feedforward control input based on Equation (5.1) using the commanded values for θ and ϕ. By setting the $\ddot{\theta}$, $\ddot{\psi}$, $\dot{\psi}$, and $\dot{\phi}$ terms to zero, the condition for static equilibrium is established. This can be used to calculate the feedforward command.

6.4 Lab 5: Hints

1.

Chapter 7

Successive Loop Closure using PID

The purpose of this lab is to design controllers for the whirlybird using PID control and successive loop closure to enable precise positioning of the whirlybird so that the performance specifications developed in Chapter 6 are satisfied.

7.1 PID Control

PID control laws are the most commonly used industrial control law for the following reasons:

- They are easy to understand,
- They can be tuned without an in-depth understanding of the underlying physics,
- A model of the system is not required.

Basic idea:

Proportional. Act according to the current output by pushing in a direction that is opposite to the error by an amount that is proportional to the error.

Integral. Act according to the past output by integrating the accumulated error and pushing by an amount that is proportional to the integrated error.

Derivative. Act according to the future predicted output by differentiating the error signal and pushing by an amount that is proportional to the derivative of the error.

A block diagram showing a standard PID implementation is shown in Figure 7.1. An alternate form of PID control is also commonly employed. As specified in Chapter 6, we will be implementing a PID loop where the derivative gain only acts on the output y and not on the error as shown in Figure 7.2. This eliminates a zero in the closed-loop transfer function and improves the transient response of the system to abrupt inputs.

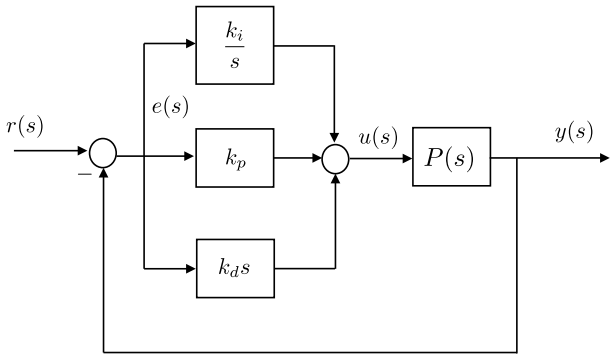


Figure 7.1: Standard implementation of PID controller.

Proportional control always appears in the controllers, but sometimes the integrator, differentiator, or both are not included. If the integrator and differentiator are not included, then the controller is referred as a proportional (P) controller. If the integrator is removed, then we have PD control, if the differentiator is removed, we have PI control.

In the time domain, PID control can be expressed as

$$u(t) = k_p e(t) + k_i \int_0^t e(\sigma)d\sigma + k_d \frac{de}{dt}(t).$$

In the frequency domain we have

$$u(s) = k_p + \frac{k_i}{s} + k_d s = \frac{k_d s^2 + k_p s + k_i}{s}. \tag{7.1}$$

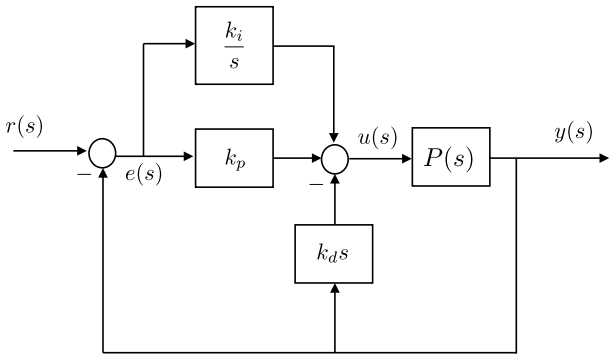


Figure 7.2: Alternative implementation of PID controller where differentiator acts on output only.

Since the order of the denominator is less than the order of the numerator, Equation (7.1) represents a non-causal system. Implementation requires the use of an approximate derivate as

$$u(s) = k_p + \frac{k_i}{s} + k_d \frac{s}{\tau s + 1} = \frac{(k_d + \tau k_p)s^2 + (k_p + \tau k_i)s + k_i}{s(\tau s + 1)}, \tag{7.2}$$

where τ is a small number. While PID control is implemented using Equation (7.2), in analysis, it is often easier to work with Equation (7.1) as a design model of the controller.

7.1.1 Discrete Implementation of a Differentiator

A practical differentiator is given by the equation

$$Z(s) = \frac{s}{\tau s + 1} W(s).$$

In the time-domain we have

$$\tau \dot{z} + z = \dot{w}.$$

Using the discrete-time approximation

$$\dot{z} \approx \frac{z[k] - z[k-1]}{T},$$

where $z[k]$ is the k^{th} sample of z and T is the sample rate, we get

$$\begin{aligned}
&\tau\left(\frac{z[k]-z[k-1]}{T}\right)+z[k]=\left(\frac{w[k]-w[k-1]}{T}\right)\\
\implies &(\tau+T)z[k]=\tau z[k-1]+(w[k]-w[k-1])\\
\implies &z[k]=\left(\frac{\tau}{\tau+T}\right)z[k-1]+\left(\frac{1}{\tau+T}\right)(w[k]-w[k-1])\\
\implies &z[k]=\left(\frac{\tau}{\tau+T}\right)z[k-1]+\left(\frac{T}{\tau+T}\right)\left(\frac{w[k]-w[k-1]}{T}\right).
\end{aligned}$$

Note the structure of the dirty derivative equation. When $\tau = 0$, we get

$$z[k]=\frac{w[k]-w[k-1]}{T},$$

which is a "pure" discrete-time derivative of the input signal w. As τ gets large $z[k] \approx z[k-1]$.

To gain insight into the structure of this equation, lets review the discrete-time implementation of a low-pass filter. In the Laplace domain, a low-pass filter is given by

$$Y(s)=\frac{a}{s+a}U(s).$$

Therefore

$$\begin{aligned}
&\dot{y}+ay=au\\
\implies &\frac{y[k]-y[k-1]}{T}+ay[k]=au[k]\\
\implies &(1+aT)y[k]=y[k-1]+aTu[k]\\
\implies &y[k]=\left(\frac{1}{1+aT}\right)y[k-1]+\left(\frac{aT}{1+aT}\right)u[k]\\
\implies &y[k]=\left(\frac{1/a}{1/a+T}\right)y[k-1]+\left(\frac{T}{1/a+T}\right)u[k].
\end{aligned}$$

Comparing this equation to the dirty derivative, it is obvious that the dirty derivative both differentiates w and low-pass filters the estimate. The bandwidth of the low-pass filter is given by $\frac{1}{\tau}$.

7.1.2 PID Implementation in Matlab

The longitudinal and lateral control strategies presented in this chapter consist of several proportional-integral-derivative (PID) control loops. In this section we briefly describe how PID loops can be implemented in discrete-time. A general PID control signal is given by

$$u(t) = k_p e(t) + k_i \int_{-\infty}^{t} e(\tau)d\tau + k_d \frac{de}{dt}(t),$$

where $e(t) = y^c(t) - y(t)$ is the error between the commanded output $y^c(t)$ and the current output $y(t)$. In the Laplace domain we have

$$U(s) = k_p E(s) + k_i \frac{E(s)}{s} + k_d s E(s).$$

Since a pure differentiator is not causal, the standard approach is to use a band-limited differentiator so that

$$U(s) = k_p E(s) + k_i \frac{E(s)}{s} + k_d \frac{s}{\tau s + 1} E(s).$$

To convert to discrete time, we use the Tustin or trapezoidal rule where the Laplace variable s is replaced with the z-transform approximation

$$s \mapsto \frac{2}{T_s}\left(\frac{1 - z^{-1}}{1 + z^{-1}}\right),$$

where T_s is the sample period [**?**]. Letting $I(s) \triangleq E(s)/s$, an integrator in the z domain becomes

$$I(z) = \frac{T_s}{2}\left(\frac{1 + z^{-1}}{1 - z^{-1}}\right) E(z).$$

Transforming to the time domain we have

$$I[n] = I[n-1] + \frac{Ts}{2}\left(E[n] + E[n-1]\right). \tag{7.3}$$

A formula for discrete implementation of a differentiator can be derived in a similar manner. Letting $D(s) \triangleq (s/(\tau s + 1))E(s)$, the differentiator in

the z domain is

$$D(z) = \frac{\frac{2}{Ts}\left(\frac{1-z^{-1}}{1+z^{-1}}\right)}{\frac{2\tau}{T_s}\left(\frac{1-z^{-1}}{1+z^{-1}}\right)+1}E(z)$$
$$= \frac{\left(\frac{2}{2\tau+T}\right)(1-z^{-1})}{1-\left(\frac{2\tau-T}{2\tau+T}\right)z^{-1}}E(z).$$

Transforming to the time domain we have

$$D[n] = \left(\frac{2\tau-T}{2\tau+T}\right)D[n-1] + \left(\frac{2}{2\tau+T}\right)(E[n]-E[n-1]). \qquad (7.4)$$

Matlab code that implements a general PID loop is shown below.

```
function u = pidloop(y_c, y, flag, kp, ki, kd, limit, Ts, tau)
  persistent integrator;
  persistent differentiator;
  persistent error_d1;
  if flag==1,  % reset (initialize) persistent variables when flag==1
      integrator = 0;
      differentiator = 0;
      error_d1   = 0; % _d1 means delayed by one time step
      y_d1 = 0;
  end
  error = y_c - y; % compute the current error
  integrator = integrator + (Ts/2)*(error + error_d1); % update integrator
  differentiator = (2*tau-Ts)/(2*tau+Ts)*differentiator...
      + 2/(2*tau+Ts)*(y - y_d1);  % update differentiator
                                  % differentiate output (not error)
  u = sat(...                    % implement PID control
      kp * error +...            % proportional term
      ki * integrator -...       % integral term
      kd * differentiator,...    % derivative term (minus derivative of outpu
      limit...                   % ensure abs(u)<=limit
      );
  error_d1 = error;  % update persistent variables
  y_d1 = y;

function out = sat(in, limit)
  if in > limit,         out = limit;
  elseif in < -limit;    out = -limit;
  else                   out = in;
  end
```

The inputs on Line 1 are the commanded output y_c, the current output y, a `flag` used to reset the integrator, the PID gains k_p, k_i, and k_d, the `limit` of the saturation command, the sample time T_s, and the derivative time constant τ. Line 11 implements Equation (7.3) and Lines 12-13 implement Equation (7.4).

A potential problem with the implementation of PID controllers using the Matlab code listed above is integrator wind-up. When the error $y_c - y$ is large and a large error persists for an extended period of time, the value of the integrator, as computed in Line 11 can become large, or "wind-up." A large integrator will cause u, as computed in Lines 14–19, to saturate, which will cause the system to push with maximum effort in the direction needed to correct the error. Since the value of the integrator will continue to wind up until the error signal changes sign, the control signal may not come out of saturation until well after the error has changed sign, which can cause a large overshoot and may potentially destabilize the system.

Since integrator wind up can destabilize the autopilot loops, it is important that each loop have an anti-wind-up scheme. A number of different anti-wind-up schemes are possible. A particularly simple scheme is to only allow the integrator to turn on when the error is below a certain threshold. When the error exceeds that threshold, the integrator is set to zero. While this scheme is simple to implement, it is not guaranteed to prevent the saturation phenomenon described in the previous paragraph. One scheme that has been suggested is to monitor the control signal u. When u exceeds the saturation limit, the value of the integrator is modified so that u is precisely at the saturation bound. As the error reduces, u will come out of saturation and the integrator will resume winding up until the error signal changes sign. This scheme will prevent the integrator from winding up while u is in saturation, and will reduce the large overshoots that can sometimes be caused by integrators.

7.2 Longitudinal Control using PID

Our goal with the longitudinal dynamics is to give a commanded pitch angle and have the whirlybird respond accordingly. This is accomplished by varying the force F exerted by the rotors. The use of an integrator for the longitudinal dynamics is very helpful because the equilibrium force is imprecisely known. This introduces a steady-state error. The integrator will eliminate this steady

state error. The block diagram in Figure 7.3 shows the general form of the closed-loop longitudinal dynamics when PID control is implemented.

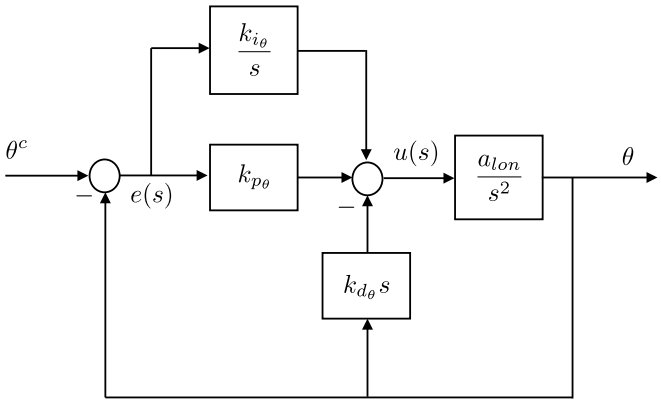


Figure 7.3: PID Diagram for Longitudinal Dynamics

The transfer function for this closed-loop system is given by

$$H_{\theta/\theta^c}(s) = \frac{k_{p_\theta} s + k_{i_\theta}}{\frac{1}{a_{lon}} s^3 + k_{d_\theta} s^2 + k_{p_\theta} s + k_{i_\theta}}. \tag{7.5}$$

where

$$a_{lon} = \frac{l_1}{m_1 l_1^2 + m_2 l_2^2 + J_y}.$$

In Chapter 6, we developed rise-time specifications for the longitudinal dynamics of the whirlybird utilizing the closed loop transfer function of the system without an integrator. In this lab, we will use the values of k_{p_θ} and k_{d_θ} we found to determine an integral gain, k_{i_θ}, that will reject steady-state error without introducing other adverse effects such as large overshoot, increased oscillation, or instability. We will use root locus methods to pick k_{i_θ}.

From Equation (7.5), we see that the closed-loop characteristic equation is given by

$$\frac{1}{a_{lon}} s^3 + k_{d_\theta} s^2 + k_{p_\theta} s + k_{i_\theta} = 0,$$

which can be placed in Evans form as

$$1 + k_{i_\theta} \left(\frac{a_{lon}}{s\left(s^2 + a_{lon} k_{d_\theta} s + a_{lon} k_{p_\theta}\right)} \right) = 0.$$

After substituting the values for a_{lon}, k_{p_θ}, and k_{d_θ}, the gain k_{i_θ} can be selected by looking at the root locus of this closed-loop system. Figure 7.4 shows an example of the root locus of the characteristic equation plotted as a function of k_{i_θ}.

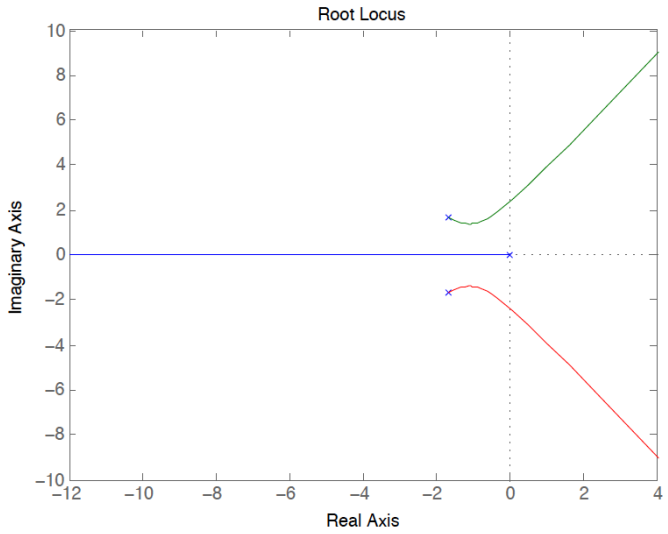


Figure 7.4: Pitch loop root locus as a function of the integral gain k_{i_θ}.

7.3 Successive Loop Closure

The primary goal in autopilot design is to control the attitude (ϕ, θ, ψ) of the whirlybird. Autopilots designed based on the assumption of decoupled dynamics will generally yield good performance. In the autopilot design discussion that follows, we will assume that the longitudinal dynamics (pitching motion) are decoupled from the lateral dynamics (rolling, yawing motions). This simplifies the development of the autopilot significantly and allows us to utilize a technique commonly used for autopilot design called successive loop closure.

The basic idea behind successive loop closure is to close several simple feedback loops in succession around the open-loop plant dynamics rather than designing a single (presumably more complicated) control system. To illustrate how this approach can be applied, consider the open-loop system

shown in Figure 7.5. The open-loop dynamics are given by the product of three transfer functions in series: $P(s) = P_1(s)P_2(s)P_3(s)$. Each of the transfer functions has an output (y_1, y_2, y_3) that can be measured and used for feedback. Typically, each of the transfer functions, $P_1(s)$, $P_2(s)$, $P_3(s)$, is of relatively low order – usually first or second order. In this case, we are interested in controlling the output y_3. Instead of closing a single feedback loop with y_3, we will instead close feedback loops around y_1, y_2, and y_3 in succession as shown in Figure 7.6. We will design the compensators $C_1(s)$, $C_2(s)$, and $C_3(s)$ in succession. A necessary condition in the design process is that the inner loop has the highest bandwidth, with each successive loop bandwidth a factor of 5 to 10 lower in frequency.

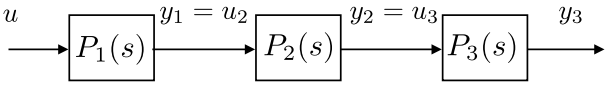


Figure 7.5: Open-loop transfer function modeled as a cascade of three transfer functions.

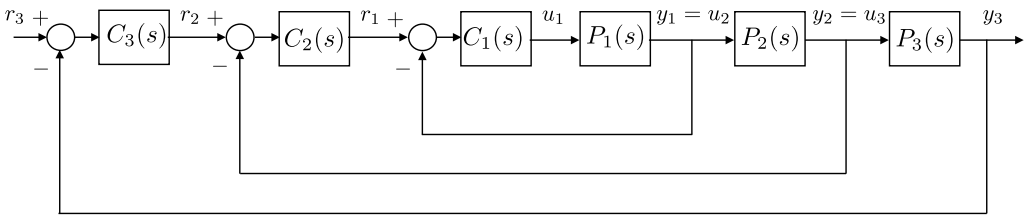


Figure 7.6: Three stage successive loop closure design.

Examining the inner loop shown in Figure 7.6, the goal is to design a closed-loop system from r_1 to y_1 having a bandwidth ω_{BW1}. The key assumption that we will make is that for frequencies well below ω_{BW1}, the closed-loop transfer function $y_1(s)/r_1(s)$ can be modeled as a gain of 1. This is depicted schematically in Figure 7.7. With the inner-loop transfer function modeled as a gain of 1, design of the second loop is simplified because it only includes the plant transfer function $P_2(s)$ and the compensator $C_2(s)$. The critical design step in closing the loops successively is to design the bandwidth of the next loop so that it is a factor of 5 to 10 slower than the preceding loop. In this case, we require $\omega_{\text{BW2}} < \frac{1}{5}\omega_{\text{BW1}}$. This ensures that

our unity gain assumption on the inner loop is not violated over the range of frequencies that the middle loop operates.

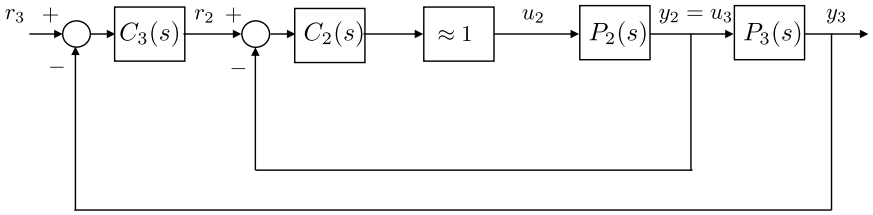


Figure 7.7: Successive loop closure design with inner loop modeled as a unity gain.

With the two inner loops operating as designed, $y_2(s)/r_2(s) \approx 1$ and the transfer function from $r_2(s)$ to $y_2(s)$ can be replaced with a gain of 1 for the design of the outermost loop, as shown in Figure 7.8. Again, there is a bandwidth constraint on the design of the outer loop: $\omega_{\text{BW3}} < \frac{1}{5}\omega_{\text{BW2}}$. Because each of the plant models $P_1(s)$, $P_2(s)$, $P_3(s)$ are first or second order, conventional PID or lead-lag compensators can be employed effectively. Transfer- function-based design methods such as root locus or loop shaping approaches are commonly used.

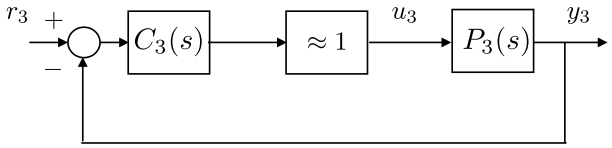


Figure 7.8: Successive loop closure design with two inner loops modeled as a unity gain.

The following section discusses the design of a lateral autopilot, using this concept of successive loop closure.

7.4 Lateral Control using Successive Loop Closure

Lateral control of the whirlybird will be slightly more difficult than the longitudinal control, as we have a cascading structure for the lateral dynamics

as shown in Figure 7.9.

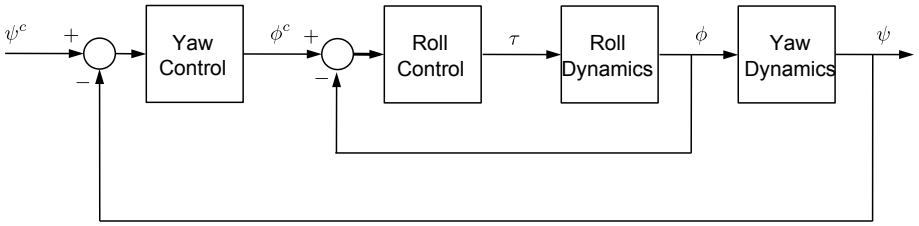


Figure 7.9: Controlled cascade system for the lateral whirlybird dynamics.

To develop a PID controller for the lateral dynamics, we first find the appropriate gains for the inner loop of the lateral dynamics, which is the roll attitude loop. We have already found values for k_{p_ϕ} and k_{d_ϕ} in Chapter 6. We will use those gains and the root locus method to find a value for k_{i_ϕ} that gives a good response.

The closed-loop roll dynamics are described by the following transfer function

$$H_{\phi/\phi^c}(s) = \frac{k_{p_\phi} s + k_{i_\phi}}{J_x s^3 + k_{d_\phi} s^2 + k_{p_\phi} s + k_{i_\phi}}, \tag{7.6}$$

which was obtained from the block diagram shown in Figure 7.10. From this transfer function, the closed-loop characteristic equation can be written as

$$J_x s^3 + k_{d_\theta} s^2 + k_{p_\theta} s + k_{i_\theta} s = 0.$$

This can also be represented in Evans form as

$$1 + k_{i_\phi} \left(\frac{1}{s\left(J_x s^2 - k_{d_\phi} s + k_{p_\phi}\right)} \right) = 0.$$

With this representation, we can generate a root locus plot in MATLAB to determine an appropriate k_{i_ϕ}.

After finding k_{d_ϕ}, k_{p_ϕ}, k_{i_ϕ}, we can then begin to develop gains for our yaw control loop. To simplify this problem, we make the key assumption that for frequencies well below ω_{n_ϕ}, the closed-loop transfer function ϕ/ϕ^c can be modeled as a gain of 1. This is depicted in Figure 7.11. Now, we can develop PID control for the yaw dynamics. We implement a PID controller following the same steps that we used in Chapter 6. First calculate the k_{p_ψ} and k_{d_ψ},

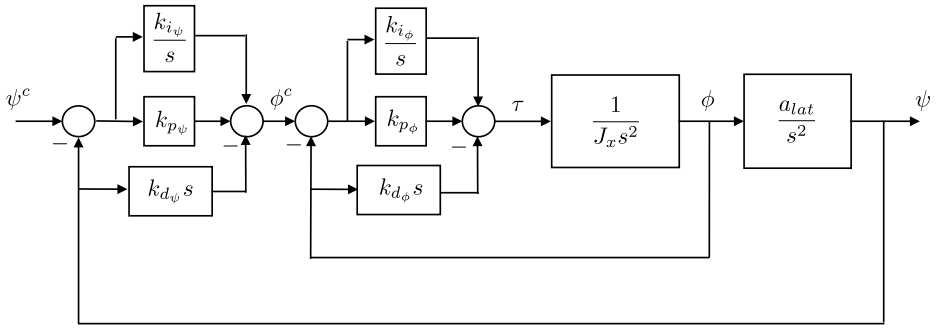


Figure 7.10: Cascading Structure of Lateral Dynamics with PID control

keeping in mind the constraint

$$\omega_{n_\psi} < \frac{1}{5}\omega_{n_\phi}.$$

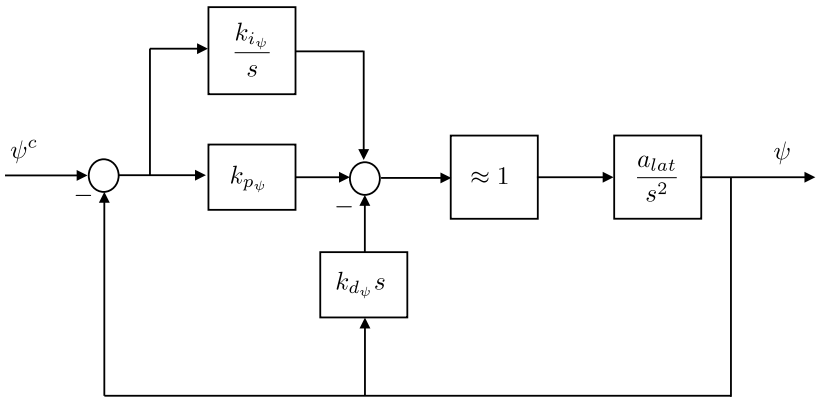


Figure 7.11: Roll Loop Modeled as a Gain of 1.

As can be seen in Figure 7.11, the output ψ can be described by the transfer function

$$H_{\psi/\psi^c}(s) = \frac{k_{p_\psi}s + k_{i_\psi}}{\frac{1}{a_{lat}}s^3 + k_{d_\psi}s^2 + k_{p_\psi}s + k_{i_\psi}} \tag{7.7}$$

where

$$a_{lat} = \frac{l_1 F_e}{m_1 l_1^2 + m_2 l_2^2 + J_z}.$$

This characteristic equation can be placed in Evans form so that we can generate a root locus plot in MATLAB to determine an appropriate value for k_{i_ψ}.

7.5 Lab 6: Assignment

1. Using the approach described in 7.1.2, modify your simulation from Lab 5 to implement your PD control using a Matlab function in Simulink. Use the gain values determined in Lab 5 and verify that the performance of the PD control implementation is the same. Set the time constant of your derivative filter to correspond to a cut-off frequency of 30 Hz ($1/\tau = (30)(2\pi)$ rad/s).

2. Find the gain k_{i_θ} using root locus techniques as described in this chapter. Implement a PID longitudinal controller in simulation. Give the whirlybird a commanded pitch angle and observe its response. Verify that you achieve zero steady-state error using a scope on θ.

3. Find the gain k_{i_ϕ} using root locus techniques as described in this chapter. Implement a PID controller for the roll loop in simulation. Verify that it performs as intended. From its rise time, estimate the bandwidth of the roll loop.

4. From the bandwidth of the roll loop, choose the bandwidth of the yaw loop to be about a factor of ten slower ($\omega_{n_\psi} \approx \omega_{n_\phi}/10$. Using the methods from Chapter 6, find the gains $k_{p_{psi}}$ and $k_{d_{psi}}$. Instead of calculating $k_{p_{psi}}$ from the saturation limit, you will calculate it from ω_{n_ψ}. Find a suitable value for the gain k_{i_ψ} using root locus techniques described in this chapter. Implement lateral control in simulation. Specify a commanded yaw angle and demonstrate that your simulated whirlybird responds as expected. Verify its performance with scopes on ψ and ϕ. Before continuing on to the final step, pass off steps 1 through 4 with the TA.

5. Implement complete design on whirlybird hardware. Carefully study Appendix B for details on how to implement your controller with the whirlybird. *Please ensure that you have a TA with you the first time you operate one of the whirlybirds. They can be dangerous!* To avoid

injuries and damage to the hardware, please be cautious and careful as you use the hardware.

7.6 Lab 6: Hints

1.

Chapter 8

Loopshaping Design

8.1 Lead-lag Control Overview

For this lab, we will design a lead-lag compensator to control the pitch motion of the whirlybird. The pitch motion control diagram that we will use for this lab is shown in Figure 8.1. The parameter b_0 is given by

$$b_0 = \frac{\ell_1}{m_1\ell_1^2 + m_2\ell_2^2 + J_y}.$$

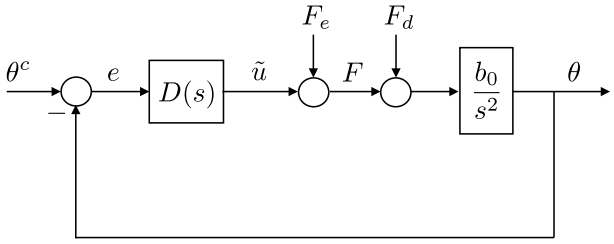


Figure 8.1: Pitch control block diagram showing equilibrium force and disturbance force.

The disturbance force F_d is the effective force acting on the head of the whirlybird due to the gravitational forces acting on the system, which can be calculated as

$$F_d = (m_2\ell_2 - m_1\ell_1)\frac{g\cos\theta}{\ell_1}.$$

To compensate for this gravity-based disturbance that pulls the whirlybird down to the ground, we have applied a feedforward force command or equilibrium force input to the system:

$$F_e = (m_1\ell_1 - m_2\ell_2)\frac{g\cos\theta^c}{\ell_1}.$$

If the equilibrium force model includes all of the physical force acting, the model parameters (m_1, m_2, ℓ_1, ℓ_2) are perfectly known, $\theta = \theta^c$, and the implementation of the feedforward force command by the thrusters is perfect, then the feedforward of the equilibrium force F_e will cancel the disturbance force F_d. Since this is unlikely to happen, a net disturbance force will remain, which we will denote as

$$\bar{F}_d = F_d + F_e.$$

This is depicted graphically in Figure 8.2. In this lab, we will be concerned with how the system responds to the command input θ^c and the disturbance input $\bar{F}_d$.

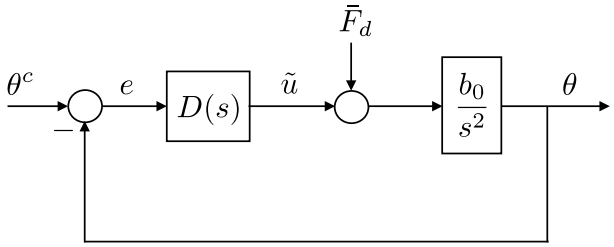


Figure 8.2: Simplified pitch control block diagram.

In this lab, we will implement lead and lead-lag compensators of the form

$$D_{\text{lead}}(s) = K\frac{T_1 s + 1}{\alpha_1 T_1 s + 1}$$
$$D_{\text{lead-lag}}(s) = \left(K\frac{T_1 s + 1}{\alpha_1 T_1 s + 1}\right)\left(\alpha_2\frac{T_2 s + 1}{\alpha_2 T_2 s + 1}\right)$$

where $\alpha_1 < 1$ and $\alpha_2 > 1$. From Figure 8.2, the error due to command and disturbance inputs can be calculated as

$$e = \frac{s^2}{s^2 + b_0 D(s)}\,\theta^c - \frac{b_0}{s^2 + b_0 D(s)}\,\bar{F}_d.$$

Since the lead and lead-lag compensators we will implement do not have any free s terms in the denominator, we can see that this system is Type 2 with respect to reference inputs and Type 0 with respect to disturbance inputs.

8.2 Discrete Compensator Implementation

You will design a continuous lead and a continuous lead-lag compensator using the methods taught in class. You will implement these compensators in your Simulink animation of the whirlybird. To implement your compensators on the whirlybird hardware, you will first convert them to discrete transfer functions and then to difference equations that can be implemented in the LabVIEW control VI. Here are the steps:

1. Convert the continuous $D(s)$ to a discrete $D(z)$ in Matlab using the `c2d` command: `Dd = c2d(D,Ts,'tustin')`. `Ts` is the sample period and `'tustin'` denotes that Tustin's method will be used to make the discrete approximation.

2. The resulting transfer function will be of the form

$$\begin{aligned} D(z) = \frac{\tilde{U}(z)}{E(z)} &= \frac{b_0 z^n + b_1 z^{n-1} \cdots + b_{n-1} z + b_n}{z^n + a_1 z^{n-1} \cdots + a_{n-1} z + a_n} \\ &= \frac{b_0 + b_1 z^{-1} \cdots + b_{n-1} z^{-(n-1)} + b_n z^{-n}}{1 + a_1 z^{-1} \cdots + a_{n-1} z^{-(n-1)} + a_n z^{-n}} \end{aligned}$$

3. Convert $D(z)$ to a difference equation that can be implemented in your control loop by taking the inverse Z-transform and recognizing that z^{-1} can be thought of as a one-time-step delay operator. Accordingly,

$$(1 + a_1 z^{-1} \cdots + a_{n-1} z^{-(n-1)} + a_n z^{-n}) \tilde{U}(z) = (b_0 + b_1 z^{-1} \cdots + b_{n-1} z^{-(n-1)} + b_n z^{-n}) E(z)$$

$$\tilde{u}_k + a_1 \tilde{u}_{k-1} \cdots + a_{n-1} \tilde{u}_{k-(n-1)} + a_n \tilde{u}_{k-n} = b_0 e_k + b_1 e_{k-1} \cdots + b_{n-1} e_{k-(n-1)} + b_n e_{k-n},$$

resulting in

$$\tilde{u}_k = +b_0 e_k + b_1 e_{k-1} \cdots + b_{n-1} e_{k-(n-1)} + b_n e_{k-n} - a_1 \tilde{u}_{k-1} \cdots - a_{n-1} \tilde{u}_{k-(n-1)} - a_n \tilde{u}_{k-n}.$$

For a second-order compensator transfer function (like our lead-lag design), $n = 2$ and the difference equation simplifies to

$$\tilde{u}_k = +b_0 e_k + b_1 e_{k-1} + b_2 e_{k-2} - a_1 \tilde{u}_{k-1} - a_2 \tilde{u}_{k-2}.$$

The subscript k denotes the current sample period, $k-1$ denotes the previous sample period, $k-2$ denotes two sample periods prior, and so forth.

8.3 Lab 7: Assignment

Check Appendix A for the latest, greatest values of the whirlybird parameters. Steps 1 through 4 of the assignment will be due after one week. Step 5 will be due after two weeks.

1. Design a lead compensator for the pitch axis of the whirlybird. Set the crossover frequency at $\omega_c = 5$ rad/s and set the phase margin to be 60 degrees. Using your Simulink animation, simulate the response of the system to a 30 degree step input in commanded pitch and to a -1 N force disturbance. Use the simulation from Lab 5 and replace the PID design with a continuous transfer-function block modeling your lead transfer function. (This disturbance for can be applied to the system at the same location as the equilibrium force or feedforward force from your controller.) Observe the steady-state error to the two types of inputs.

2. Design a lag compensator to complement your lead compensator from Step 1. The goal of the lag compensator is to reduce the steady-state error in response to force disturbances by a factor of 20. How has the response of the system to pitch commands and disturbance forces changed?

3. Convert your continuous lead-lag transfer function to its discrete equivalent using the `c2d` command in Matlab. Use the bilinear approximation (Tustin's method) and a sample period of 0.005 s (200 Hz). Implement this controller using a discrete transfer-function block in your Simulink animation. Verify that the performance is similar to that obtained in Step 2.

   Caution: The discrete transfer function implementation is sensitive to numerical round-off error. For this reason, you should format your Matlab output to include greater precision than the standard four decimal places. Use the `format long` command to do this. Print your discrete transfer function coefficients using the command `[n,d]=tfdata(sys,'v')`.

The `'v'` option prints the numerator and denominator coefficients in vector form. You can then cut and paste them directly into the discrete transfer function dialog box in Simulink.

4. Convert your discrete lead-lag transfer function into a difference equation with the current value of the control $\tilde{u}_k$ expressed as a function of past values of the control ($\tilde{u}_{k-1}$ and $\tilde{u}_{k-2}$) and current and past values of the error (e_k, e_{k-1}, and e_{k-2}). Implement this pitch control law in your Matlab autopilot.m function from Lab 6 in place of the PID control design. The sample time for the autopilot.m function should be set to 0.005 s. Test the performance of the system and verify that it performs as anticipated (it should be identical to the performance of Step 3).

   Note that the difference equation calculations for $\tilde{u}_k$ and the aging of the past control values $\tilde{u}_{k-1}$ and $\tilde{u}_{k-2}$ should not be mixed with the feedforward force calculation – this will mess things up badly.

5. Implement the control equations from your Matlab autopilot.m function into the LabVIEW whirlybird control VI. Start with small step inputs initially as larger step sizes will cause saturation. Provide small disturbance inputs to the system by pushing down on the whirlybird shaft (Keep your hands away from the spinning props!). Observe the response and feel the "stiffness" of the position control. Implement difference equations corresponding to your original lead design and test the response of the system. How does the disturbance response compare to the lead-lag implementation?

8.4 Lab 7: Hints

1.

Chapter 9

Full State Feedback

The purpose of this lab is to design and implement full state feedback control strategies to satisfy the specifications developed in Chapter 6. An integrator will be used on the longitudinal dynamics to compensate for inaccuracies in modeling the equilibrium force needed to compensate for gravity.

9.1 Lateral Control using Linear State Feedback

In Chapter 6 you picked desired pole locations for the roll and yaw modes. The objective of lateral control is to design a linear state feedback of the form

$$\tau = -K_{lat} x_{lat} + k^r_{lat} \psi^c$$

that places the poles of the lateral dynamics at the desired pole locations. Let

$$p_{lat} = [p_{roll}, p_{yaw}]$$

be the desired pole locations for the lateral dynamics, where p_{roll} are the desired desired pole locations for roll and p_{yaw} are the desired pole locations for yaw. The gain K_{lat} can be found using the Matlab command

```
>> K_lat = place(A_lat, B_lat, p_lat)
```

where A_{lat} and B_{lat} are given in Equations (5.10) and (5.11) respectively. The feedforward gain k^r_{lat} is given by the formula

$$k^r_{lat} = -\frac{1}{C_{yaw}(A_{lat} - B_{lat}K_{lat})^{-1}B_{lat}},$$

where $C_{yaw} = (0, 0, 1, 0)$.

9.2 Longitudinal Control using Linear State Feedback and an Integrator

The form of the controller for the longitudinal controller is

$$F = F_e - K_{lon}\tilde{x}_{lon} + k^r_{lon}\theta^c + k_i \int_0^t (\theta^c - \theta(\sigma))\, d\sigma.$$

The first term is the feedforward equilibrium force found in Chapter 5. The second term is the state feedback control based on the linearized state. The third term is a feedforward term that ensures that the DC-gain of the closed-loop system is one. Since F_e is constant, when the commanded pitch angle is not equal to θ_e, F_e will not exactly cancel gravity, and the result will be a steady state pitch error. To remove this error we add the integrator as the last term of the controller.

The control design requires the selection of three poles: two associated with the longitudinal dynamics as selected in Chapter 6, and one associated with the integrator. A good rule of thumb for the initial selection of the integrator pole is to select it to be one tenth of the real part of the longitudinal poles. Let

$$p_{lon} = [p_{pitch}, p_{integrator}]$$

be the desired longitudinal poles. The gains K_{lon} and k_i can be found by using the Matlab command

```
>> KK_lon = place(AA_lon, BB_lon, p_lon)
>> K_lon = KK_lon(1:2)
>> k_i = KK_lon(3)
```

where

$$AA_{lon} = \begin{pmatrix} A_{lon} & 0 \\ C_{lon} & 0 \end{pmatrix}$$

$$BB_{lon} = \begin{pmatrix} B_{lon} \\ 0 \end{pmatrix},$$

and where A_{lon} and B_{lon} are given in Equations (5.4) and (5.5) respectively. The feedforward gain k^r_{lon} is given by the formula

$$k^r_{lon} = -\frac{1}{C_{lon}(A_{lon} - B_{lon}K_{lon})^{-1}B_{lon}}.$$

9.3 Linear Quadratic Regulator

The state feedback gain K and the reference input gain k_r can be designed using linear quadratic regulator (LQR) theory. With LQR we assume state space models of the form

$$\dot{x} = Ax + Bu \tag{9.1}$$
$$y = Cx \tag{9.2}$$
$$z = Gx + Hu \tag{9.3}$$

where $x \in \mathbb{R}^n$ is the state, $u \in \mathbb{R}^m$ is the input, $y \in \mathbb{R}^p$ is the measured output, and $z \in \mathbb{R}^\ell$ is the controlled output. The objective is to select $u(t)$ to minimize the cost function [**?**]

$$J(x_0) = \int_0^\infty z^\top(t)\bar{Q}z(t) + \rho u^\top(t)\bar{R}u(t)\,dt$$

where $\bar{Q}$ and $\bar{R}$ are symmetric and positive definite. Since $z = Gx + Hu$ we have that

$$z^\top\bar{Q}z + \rho u^\top\bar{R}u = x^\top G^\top\bar{Q}Gx + u^\top\left(\rho I + H^\top\bar{Q}H\right)u + u^\top(2H^\top\bar{Q}G)x.$$

Therefore defining $Q \triangleq G^\top\bar{Q}G$, $R \triangleq \rho I + H^\top\bar{Q}H$, and $N \triangleq 2H^\top\bar{Q}G$, we get that

$$J(x_0) = \int_0^\infty x^\top(t)Qx(t) + u^\top(t)Ru(t) + u^\top(t)Nx(t)\,dt. \tag{9.4}$$

In Matlab, the LQR problem defined by Equations (9.1) and (9.4) is solved using the command

`[K,S,E]=lqr(A,B,Q,R,N)`

where the optimal control is given by $u^*(t) = -Kx(t)$.

9.3.1 Reference Input

In order to force y to track a reference input $r \in \mathbb{R}^p$, the optimal control is given by $u^*(t) = -Kx(t) + K^r r(t)$, where the reference gain K^r is selected so that the DC gain from r to y is the identity. In other words, K^r is selected so that

$$\lim_{s\to 0} C(sI - (A - BK))^{-1}BK^r = -C(A - BK)^{-1}BK^r = I,$$

where we have assumed that there are no closed loop poles at the origin. Therefore, the reference gain is given by

$$K^r = -\left[C(A - BK)^{-1}B\right]^{-1}. \tag{9.5}$$

9.3.2 Adding an Integrator

The discussion in this section follows [**?**]. Suppose the system is subjected to an unknown constant disturbance $\bar{w}$ so that the state equations become

$$\dot{x} = Ax + Bu + \bar{w}.$$

An interesting consequence is that x and u cannot be zero in equalibrium, but rather must be a non-zero constant to cancel the effect of $\bar{w}$. Therefore, the cost criteria (9.4) cannot be finite.

Let $\bar{x}$ and $\bar{u}$ be the equilibrium values of x and u. Then, to be able to cancel the disturbance $\bar{w}$ we must have that

$$0 = A\bar{x} + B\bar{u} + \bar{w}.$$

If A^{-1} exists then

$$\bar{x} + A^{-1}B\bar{u} = -A^{-1}\bar{w}.$$

Assume that there exists a D such that in equilibrium we have $D\bar{x} = 0$. Then

$$DA^{-1}B\bar{u} = -DA^{-1}\bar{w}.$$

Therefore, if $DA^{-1}B$ is invertible, then the equilibrium control value is

$$\bar{u} = -(DA^{-1}B)^{-1}DA^{-1}\bar{w}.$$

Since $Dx(t) \to 0$, we can define the augmented state

$$x_I(t) = \begin{pmatrix} \dot{x} \\ Dx \end{pmatrix},$$

where

$$\dot{x}_I = \begin{pmatrix} \ddot{x} \\ D\dot{x} \end{pmatrix} = \begin{pmatrix} A\dot{x} + B\dot{u} \\ D\dot{x} \end{pmatrix} = \begin{pmatrix} A & 0 \\ D & 0 \end{pmatrix} x_I + \begin{pmatrix} B \\ 0 \end{pmatrix} v,$$

and where $v = \dot{u}$. Defining

$$A_I = \begin{pmatrix} A & 0 \\ D & 0 \end{pmatrix}$$
$$B_I = \begin{pmatrix} B \\ 0 \end{pmatrix}$$
$$Q_I = \begin{pmatrix} Q_{\dot{x}} & 0 \\ 0 & Q_{Dx} \end{pmatrix},$$

we can solve the LQR problem

$$\int_0^\infty x_I^\top Q_I x_I + v^\top R_I v dt,$$

to obtain

$$K_I = \begin{pmatrix} K_{\dot{x}} & K_{Dx} \end{pmatrix}$$

and the associated optimal control (derivative)

$$v^* = \dot{u}^* = -K_I x_I = -K_{\dot{x}} \dot{x} - K_{Dx} Dx,$$

which implies that the optimal control is given by

$$u^*(t) = -K_{\dot{x}} x(t) - K_{Dx} \int_0^t Dx(\tau)\, d\tau,$$

which has the structure of PI control where $K_{\dot{x}}$ is the proportional gain and K_{Dx} is the integral gain.

For PI control, the reference gain is given by Equation (9.5) where K is $K_{\dot{x}}$.

9.4 Lab 8: Assignment

1. The pole placement technique is only guaranteed to work if the system is controllable. Verify that the systems (A_{lat}, B_{lat}) and (A_{lon}, B_{lon}) are controllable. Also verify that the system (AA_{lon}, BB_{lon}) is controllable.

2. Find the gains K_{lat} and k_{lat}^r for the lateral state feedback controller that place the poles at the desired locations specified in Chapter 6. Set up your parameter file so that the gains can be tuned by selecting ζ_{roll}, $t_{r,roll}$, ζ_{yaw}, and $t_{r,yaw}$.

3. Find the gains K_{lon}, k^r_{lon}, and k_i for the longitudinal state feedback controller that place the poles at the desired locations specified in Chapter 6. Set up your parameter file so that the gains can be tuned by selecting ζ_{pitch}, $t_{r,pitch}$, and $p_{integrator}$.

4. Implement the longitudinal and lateral controllers in your Simulink simulation of the whirlybird and verify that you satisfy your design specifications in terms of rise time. Also verify that the actuators do not saturate. Tune the controllers until you are satisfied with their performance.

5. Implement the longitudinal and lateral controllers on the whirlybird hardware. Test to verify that they meet your specifications. Tune the controllers until you are satisfied with their performance.

9.5 Lab 8: Hints

1.

Chapter 10

Observer Design and Visual Feedback

To this point the labs have been somewhat contrived in the sense that they have used the angle encoders for full state feedback. On actual flying machines, the configuration angles of the platform are not directly available from sensors and must be estimated using alternative sensors. The state-of-the art is to use interoceptive sensors like rate gyros and accelerometers in combination with exteroceptive sensors like GPS, magnetometers, or cameras, to estimate attitude and position of the platform. State estimation for flying machines is an extremely difficult problem in general and we cannot hope to discuss all of the relevant issues in an introductory course. However, the whirlybird provides a nice platform to introduce this topic. Accordingly, the purpose of this lab is to use an extended Kalman filter to blend measurements from the rate gyros and the camera to estimate the yaw angle relative to a color dot on the ground, and the roll angle of the whirlybird. The encoders will not be used for lateral control, however the pitch-axis encoder will be used to control the longitudinal motion using the longitudinal controller developed in Chapter 9.

10.1 Relative Yaw Angle

In the previous labs we commanded an absolute yaw angle based on a predefined location for $\psi = 0$. Using the encoders we were able to measure the absolute yaw angle and control the whirlybird to match that angle. However,

the encoders provide information that is not available on real flight systems. Our objective is to use only gyro and camera information to track the position of a visually distinct (red) target in the camera field of view. Therefore, rather than track an absolute yaw angle, the control objective is to regulate the relative yaw angle between the whirlybird and the target to zero. Let ψ be the yaw angle of the whirlybird, and let ψ_t be the yaw angle of the target, and define

$$\delta \triangleq \psi - \psi_t$$

as the relative yaw angle between the whirlybird and the target.

Since the motion of the target cannot be distinguished from the motion of the whirlybird, we will assume that the target is not moving, i.e., we will assume that

$$\dot{\psi}_t = \ddot{\psi}_t = 0.$$

Therefore

$$\dot{\delta} = \dot{\psi}$$
$$\ddot{\delta} = \ddot{\psi}$$

Using the lateral equations of motion we get the following state-space model for the relative lateral dynamics:

$$\begin{pmatrix} \dot{\phi} \\ \ddot{\phi} \\ \dot{\delta} \\ \ddot{\delta} \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ \frac{\ell_1 F_e}{m_1\ell_1^2+m_2\ell_2^2+J_z} & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} \phi \\ \dot{\phi} \\ \delta \\ \dot{\delta} \end{pmatrix} + \begin{pmatrix} 0 \\ 1/J_x \\ 0 \\ 0 \end{pmatrix} \tau. \tag{10.1}$$

10.2 Sensor Models

10.2.1 Rate Gyros

A MEMS rate gyro contains a small vibrating lever. When the lever undergoes an angular rotation, Coriolis effects change the frequency of the vibration, thus detecting the rotation. A brief description of the physics of rate gyros can be found at http://www.xbow.com/Support/Support_pdf_files/RateSensorAppNote.pdf.

The output of the rate gyro is given by

$$y_{gyro} = k_{gyro}\Omega + \beta_{gyro} + \eta_{gyro},$$

where y_{gyro} is in Volts, k_{gyro} is a gain, Ω is the angular rate in radians per second, β_{gyro} is a bias term, and η_{gyro} is zero mean white noise. The gain k_{gyro} should be given on the spec sheet of the sensor. However, due to variations in manufacturing it is imprecisely known. The bias term β_{gyro} is strongly dependent on temperature and should be calibrated prior to each flight.

The rate gyros attached to the whirlybird are temperature compensated and calibrated to units of radians/sec. The gyros are aligned with the x, y, and z axes of the whirlybird head and measure the angular rates of the head, which are not equal to $\dot{\phi}$, $\dot{\theta}$, and $\dot{\psi}$ unless $\phi = \theta = 0$. To understand why this is true, consider the case where the head is rolled by $\phi = \pi/2$. The rotation rate $\dot{\psi}$ will now be measured as a rotation about the y-axis (i.e., the pitch axis) of the head and not the z-axis. Let p, q, and r denote the rotation about the x, y, and z axis of the whirlybird head. The relationship between p, q, and r and $\dot{\phi}$, $\dot{\theta}$, and $\dot{\psi}$ is given by

$$\begin{pmatrix} p \\ q \\ r \end{pmatrix} = \begin{pmatrix} 1 & 0 & -\sin\theta \\ 0 & \cos\phi & \sin\phi\cos\theta \\ 0 & -\sin\phi & \cos\phi\cos\theta \end{pmatrix} \begin{pmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{pmatrix}. \tag{10.2}$$

The output of the rate gyros is given by

$$\begin{aligned} y_{gyro,x} &= k_{gyro,x} p + \beta_{gyro,x} + \eta_{gyro,x} \\ y_{gyro,y} &= k_{gyro,y} q + \beta_{gyro,y} + \eta_{gyro,y} \\ y_{gyro,z} &= k_{gyro,z} r + \beta_{gyro,z} + \eta_{gyro,z}. \end{aligned}$$

For simulation purposes, we may assume that the gains are identical and equal to one, and that the biases have been estimated and subtracted from the measurements to produce

$$\begin{aligned} \hat{y}_{gyro,x} &= \dot{\phi} - \dot{\psi}\sin\theta + \eta_{gyro,x} \\ \hat{y}_{gyro,y} &= \dot{\theta}\cos\phi + \dot{\psi}\sin\phi\cos\theta + \eta_{gyro,y} \\ \hat{y}_{gyro,z} &= -\dot{\theta}\sin\phi + \dot{\psi}\cos\phi\cos\theta + \eta_{gyro,z}. \end{aligned} \tag{10.3}$$

In this lab we will use encoders for longitudinal control and can therefore ignore the y-axis gyro and assume that $\dot{\theta} = 0$ to obtain the following "design model" for the gyros:

$$\hat{y}_{gyro,x} = \dot{\phi} - \dot{\psi}\sin\theta + \eta_{gyro,x} \stackrel{\triangle}{=} h_{gyro,x}(x) + \eta_{gyro,x} \tag{10.4}$$

$$\hat{y}_{gyro,z} = \dot{\psi}\cos\phi\cos\theta + \eta_{gyro,z} \stackrel{\triangle}{=} h_{gyro,z}(x) + \eta_{gyro,z}. \tag{10.5}$$

10.2.2 Camera

Our next objective is to model the output of the camera. The camera data is processed by a Labview software module that returns the location of the centroid of certain colors in the image plane. In the simulator, we will return the pixel locations of a target in the camera field of view. The geometry of the camera is shown in Figure 10.1

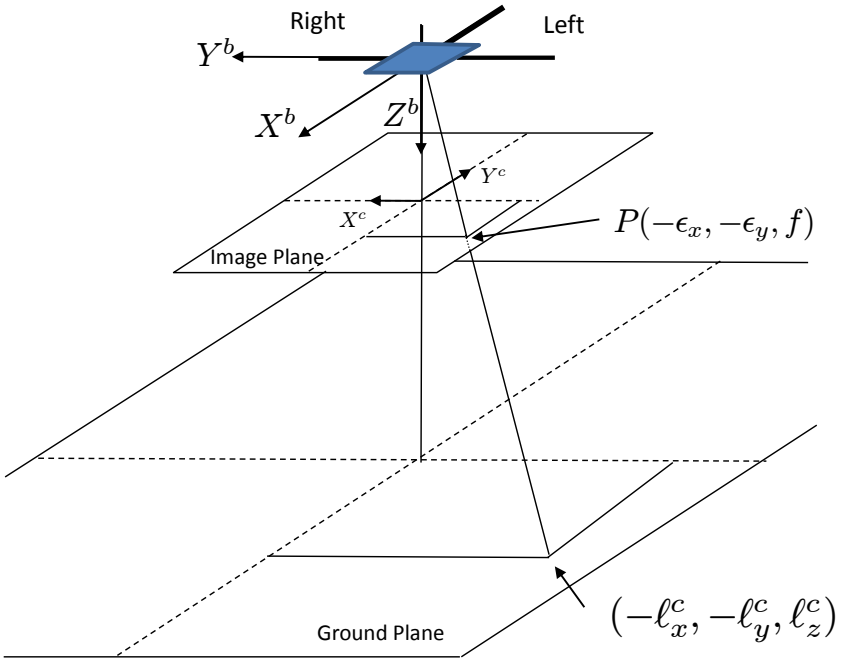


Figure 10.1: Camera model. The position of the target in the ground plane is given by (p_x, p_y). The pixel locations of the target in the image plane is given by (ϵ_x, ϵ_y)

In the computer vision world, the camera x-axis is typically defined to the right in the image, and the y-axis is defined down in the image. Therefore, the z-axis is along the optical axis. For the whirlybird the transformation from the body frame to the camera frame is given by

$$R_b^c = \begin{pmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

Let $\mathbf{p}^i_{target}$ be the inertial position of the target expressed in inertial coordinates, and let $\mathbf{p}^i_{whirlybird}$ be the position of the whirlybird expressed in inertial coordinates, and let $\tilde{\mathbf{p}}^i = \mathbf{p}^i_{target} - \mathbf{p}^i_{whirlybird}$ be the relative position of the target expressed in inertial coordinates. The relative position expressed in camera coordinates is given by

$$\tilde{\mathbf{p}}^c = R^c_b R^b_i \tilde{\mathbf{p}}^i,$$

where the rotation matrix from the inertial to the body frame is given by

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

In component form, let

$$\tilde{\mathbf{p}}^c = \begin{pmatrix} \ell^c_x \\ \ell^c_y, \\ \ell^c_z \end{pmatrix}.$$

The geometry in the camera frame is shown in Figure 10.1, where f is the focal length in units of pixels, and P converts pixels to meters. To simplify the discussion, we will assume that the pixels and the pixel array are square. If the width of the square pixel array is M and the field-of-view of the camera η is known, then the focal length f can be expressed as

$$f = \frac{M}{2\tan\left(\frac{\eta}{2}\right)}. \tag{10.6}$$

The position of the projection of the object onto the image plane and expressed in the camera frame is given by $(P\epsilon_x, P\epsilon_y, Pf)$, where ϵ_x and ϵ_y are the pixel location (in units of pixels) of the object. The position of the object expressed in the camera frame is given by $(\ell_x, \ell_y, \ell_z)^\top$ as shown in Figure 10.1.

Using similar triangles to express the relationship between pixel location and relative position gives

$$\frac{\epsilon_x}{f} = \frac{\ell^c_x}{\ell^c_z}$$
$$\frac{\epsilon_y}{f} = \frac{\ell^c_y}{\ell^c_z}.$$

The simulation model for the camera can therefore be expressed as

$$\epsilon_x = f\frac{\ell_x^c}{\ell_z^c} + \eta_{\epsilon_x} \tag{10.7}$$

$$\epsilon_y = f\frac{\ell_y^c}{\ell_z^c} + \eta_{\epsilon_y}, \tag{10.8}$$

where η_{ϵ_x} and η_{ϵ_y} are zero mean Gaussian processes.

Equation (10.7) represents the simulation model for the camera. However, it is too complex for design. An adequate design model can be obtain by considering the 2D model shown in Figure 10.2, where the pitch angle is approximately equal to zero. From similar triangles we have that

$$\frac{\epsilon_x}{f} = \frac{d + h\tan\phi}{h},$$

where $d = \ell_1\delta$. Therefore, a simplified camera model is

$$\epsilon_x = \frac{f\ell_1}{h}\delta + f\tan\phi + \eta_{pixel}.$$

10.3 State Estimation using an Extended Kalman filter

To estimate the relative state $x_{lat,rel} \stackrel{\triangle}{=} (\phi, \dot{\phi}, \delta, \dot{\delta})^T$ we will use an extended Kalman filter (EKF) with a continuous-time prediction step and a sampled-data measurement update. The EKF algorithm is summarized in Algorithm 1, where we assume the dynamic model

$$\dot{x} = f(x, u) + \xi$$

and the measurement model

$$y_k = h(x) + \eta$$

where ξ is a zero mean Gaussian process with covariance Q, and η is a zero mean Gaussian process with covariance R.

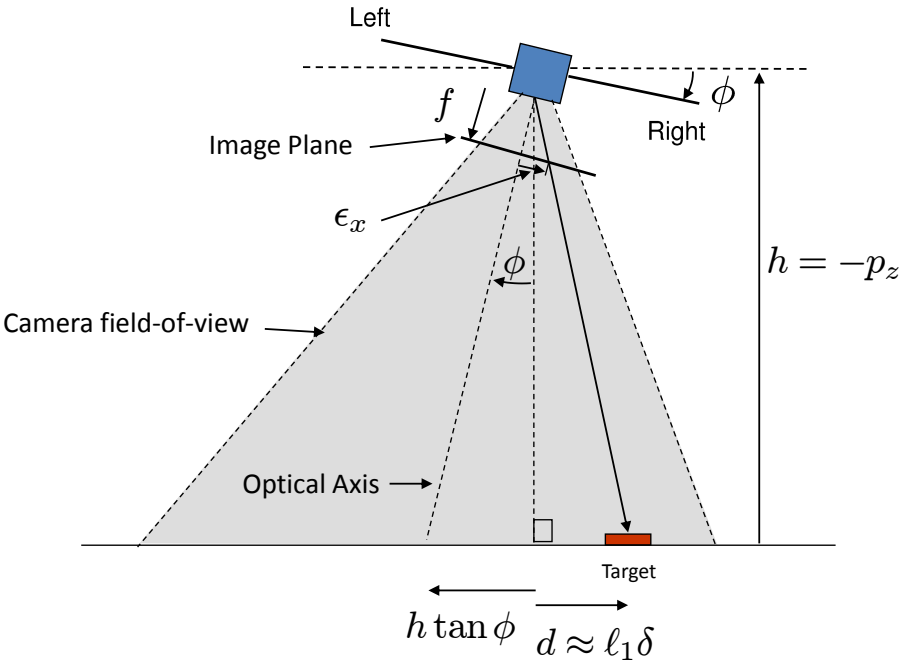


Figure 10.2: 2D camera geometry looking down the whirlybird arm. The camera focal length in pixels is f, and projection of the position of the target onto the image plane is ϵ_x. The distance of the whirlybird head to the target is given by $\ell_1 \delta$.

Algorithm 1 Summary of Extended Kalman Filter

1: Initialize: $\hat{x} = 0$.
2: Pick an output sample rate T_{out} which is much less than the sample rates of the sensors.
3: At each sample time T_{out}:
4: **for** $i = 1$ to N **do** {Propagate the equations.}
5: $\quad \hat{x} = \hat{x} + \left(\frac{T_{out}}{N}\right) f(\hat{x}, u)$
6: $\quad A = \frac{\partial f}{\partial x}(\hat{x}, u)$
7: $\quad P = P + \left(\frac{T_{out}}{N}\right)\left(AP + PA^T + Q\right)$
8: **end for**
9: **if** Measurement y received, **then** {Measurement Update}
10: $\quad C = \frac{\partial h}{\partial x}(\hat{x}, u)$
11: $\quad L = PC^T(R + CPC^T)^{-1}$
12: $\quad P = (I - LC)P$
13: $\quad \hat{x} = \hat{x} + L\left(y - h(\hat{x}, u)\right)$.
14: **end if**

10.4 Lab 9: Assignment

1. In your whirlybird simulator, add a sensors block to the output of the whirlybird dynamics. The input to the block should be the states of the whirlybird, and the output will be three gyro measurements, and the x and y pixel locations. Use Equations (10.3) to simulate the gyros, and use Equation (10.7) to simulate the pixel locations of the target. Note that the Matlab `randn` command can be used to synthesize a Gaussian random variable with zero mean and variance equal to one. To get a variance of σ, multiply `randn` by σ.

2. Observer design is predicated on the observability of the system. The observability of the system can be analyzed by using the output vector

$$y \stackrel{\triangle}{=} \begin{pmatrix} \hat{y}_{gyro,x} \\ \hat{y}_{gyro,z} \\ \epsilon_x \end{pmatrix} = \begin{pmatrix} h_{gyro,x}(x) \\ h_{gyro_z}(x) \\ h_{\epsilon_x}(x) \end{pmatrix} + \eta \stackrel{\triangle}{=} h(x) + \eta. \tag{10.9}$$

Using h defined above, compute $C = \frac{\partial h}{\partial x}$, and find the rank of the observability matrix, where the A matrix is obtained from Equation (10.1). Are the states ϕ, $\dot{\phi}$, δ, $\dot{\delta}$ observable using these three outputs?

3. In your whirlybird simulator, implement an extended Kalman filter to estimate the state $x = (\phi, \dot{\phi}, \delta, \dot{\delta})$ using the linear model in (10.1) for the prediction step, and the nonlinear sensor model given in (10.9) for the correction step. To avoid inverting a 3×3 matrix, break the correction step into three distinct correction substeps, one for each sensor. Using Q, tune the filter to give good performance in simulation. At this stage, use the actual state, and NOT the estimated state, for feedback control.

4. Implement the Kalman filter on the whirlybird hardware and tune the Kalman filter using the input gains. Use the state returned by the encoders for feedback control. Note that the system will not follow the red dot, but that the roll and relative yaw angles should be accurately estimated as long as the red dot remains in the camera field of view.

5. In your whirlybird simulator, modify the autopilot to use the estimated state of the relative lateral dynamics. Tune the Kalman filter and control gains as required to get good performance.

6. Implement visual control on the whirlybird hardware and tune the filter and controller until you are satisfied with the performance.

10.5 Lab 9: Hints

1.

Chapter 11

Nonlinear Control Design

In this chapter we outline a design project to be completed in ECE 774: Nonlinear Control.

11.1 Equations of Motion

The equations of motion for the whirlybird system are given by Equations (5.2), (5.7), and (5.8). If we define

$$
\begin{aligned}
a_\theta &\triangleq \frac{(m_2\ell_2 - m_1\ell_1)g}{m_1\ell_1^2 + m_2\ell_2^2 + J_y} \\
b_\theta &\triangleq \frac{\ell_1}{m_1\ell_1^2 + m_2\ell_2^2 + J_y} \\
b_\psi &\triangleq \frac{2\dot\psi\dot\phi(J_z - J_y)c_\phi + \ell_1 F_e}{m_1\ell_1^2 + m_2\ell_2^2 + J_y s_\phi^2 + J_z c_\phi^2} \\
a_\phi &\triangleq \frac{(J_z - J_y)}{J_x} \\
b_\phi &\triangleq \frac{1}{J_x}
\end{aligned}
$$

where

$$
F_e = \frac{(m_1\ell_1 - m_2\ell_2)g}{\ell_1},
$$

then the equation of motion for the longitudinal dynamics is given by

$$
\ddot\theta = a_\theta \cos\theta + b_\theta F, \tag{11.1}
$$

where the input is the force F, and the equations of motion for the lateral dynamics are given

$$\ddot{\psi} = b_\psi \sin\phi \tag{11.2}$$

$$\ddot{\phi} = a_\phi \dot{\psi}^2 \sin\phi\cos\phi + b_\phi \tau, \tag{11.3}$$

where the input is the torque τ.

11.2 Assignment

1. *(Parameters)* Using the parameters given in Appendix A, compute nominal values for the parameters a_θ, b_θ, b_ψ, a_ϕ, b_ϕ. To compute b_ψ assume that $\phi = \dot{\phi} = \dot{\psi} = 0$.

2. *(Feedback Linearization)*

   (a) Using output feedback linearization with $y = \theta$, design a force controller (F) for the longitudinal dynamics given in Equation (11.1) so that the closed loop system has the transfer function

   $$\Theta(s) = \frac{\omega_{n_\theta}^2}{s^2 + 2\zeta_\theta \omega_{n_\theta} s + \omega_{n_\theta}^2} \Theta^c(s)$$

   where $\zeta = 0.707$ and ω_n is a design parameter.

   (b) Using output feedback linearization with $y = \psi$, design a torque controller (τ) for the lateral dynamics given in (11.2) and (11.3) so that the closed-loop system has the transfer function

   $$\Psi(s) = \frac{\omega_{n_\phi}^2 \omega_{n_\psi}^2}{(s^2 + 2\zeta_\phi \omega_{n_\phi} s + \omega_{n_\phi}^2)(s^2 + 2\zeta_\psi \omega_{n_\psi} s + \omega_{n_\psi}^2)} \Psi^c(s),$$

   where $\zeta_\phi = \zeta_\psi = 0.707$, and ω_{n_ϕ} and ω_{n_ψ} are design parameters.

   (c) Download the files `whirlybird.zip` from the class wiki and modify the function `autopilot.m` to implement the feedback linearizing controllers. Tune ω_{n_*} until are satisfied with the performance. Since the roll angle ϕ is an inner loop to the yaw angle ψ, you should expect that ω_{n_ψ} is at least five times smaller than ω_{n_ϕ}.

(d) Download the file `controller.vi` from the class wiki and modify to implement the feedback linearizing controllers. Compile and execute the functions on the whirlybird hardware and tune the parameters to get acceptable performance.

3. *(Adaptive Control)*

(a) Design an adaptive controller for the longitudinal dynamics given in Equation (11.1) assuming that a_θ and b_θ are not precisely known, so that $\theta(t) \to \theta^c$ where θ^c is a constant pitch command. Using Lyapunov theory and Barbalat's lemma, argue that $\tilde{\theta} = \theta - \theta^c$, $\dot{\theta}$, and the parameter errors remain bounded and that $\theta(t) \to \theta^c$, and $\dot{\theta} \to 0$.

(b) Design an adaptive controller for the lateral dynamics given in Equations(11.2) and (11.3) assuming that b_ψ, a_ϕ and b_ϕ are not precisely known so that $\psi(t) \to \psi^c$ where ψ^c is a constant yaw command. Using Lyapunov theory and Barbalat's lemma, argue that $\tilde{\psi} = \psi - \psi^c$, $\dot{\psi}$, $\tilde{\phi} = \phi - \phi^c$, $\dot{\phi}$,and the parameter errors remain bounded and that $\psi(t) \to \psi^c$, and $\dot{\psi}$, ϕ, and $\dot{\phi}$ all approach zero.

(c) In Simulink, modify `autopilot.m` to implement the adaptive controllers, and tune the control parameters to get performance similar to feedback linearization.

(d) Implement the adaptive controllers on the whirlybird hardware and tune the parameters to get acceptable performance.

4. *(Sliding Mode Control)* In this problem, assume that the parameters given in Appendix A are only accurate to within 10% of their nominal values.

(a) Find upper and lower bounds for a_θ, b_θ, b_ψ, a_ϕ, b_ϕ.

(b) Design a sliding mode controller for the longitudinal dynamics given in Equation (11.1) so that $\theta(t) \to \theta^c$ where θ^c is a constant pitch command. Using Lyapunov theory argue that $\tilde{\theta} = \theta - \theta^c$ and $\dot{\theta} \to 0$. Replace the `sign` function with a `sat` function and argue that $\tilde{\theta}$ and $\dot{\theta}$ are uniformly ultimately bounded.

(c) Design a sliding mode controller for the lateral dynamics given in Equations(11.2) and (11.3) so that $\psi(t) \to \psi^c$ where ψ^c is a

constant yaw command. Using Lyapunov theory argue that $\tilde{\psi} = \psi - \psi^c$, and that $\dot{\psi}$, ϕ, and $\dot{\phi}$ all approach zero. Replace the `sign` function with a `sat` function and argue that $\tilde{\psi}$, $\dot{\psi}$, ϕ, and $\dot{\phi}$ are uniformly ultimately bounded.

(d) In Simulink, modify `autopilot.m` to implement the sliding mode controllers, and tune the control parameters to get performance similar to feedback linearization.

(e) Implement the sliding mode controllers on the whirlybird hardware and tune the parameters to get acceptable performance.

11.3 Lab 10: Hints

1.

Appendix A

Whirlybird Parameters

The whirlybird in the lab has the following parameters:

g	9.81	m/s^2
ℓ_1	0.85	m
ℓ_2	0.3048	m
m_1	0.891	kg
m_2	1	kg
d	0.178	m
h	0.65	m
r	0.12	m
J_x	0.0047	kg-m^2
J_y	0.0014	kg-m^2
J_z	0.0041	kg-m^2
k_m	0.0546	N/PWM
σ_{gyro}	8.7266×10^{-5}	rad
σ_{pixel}	0.05	pixel

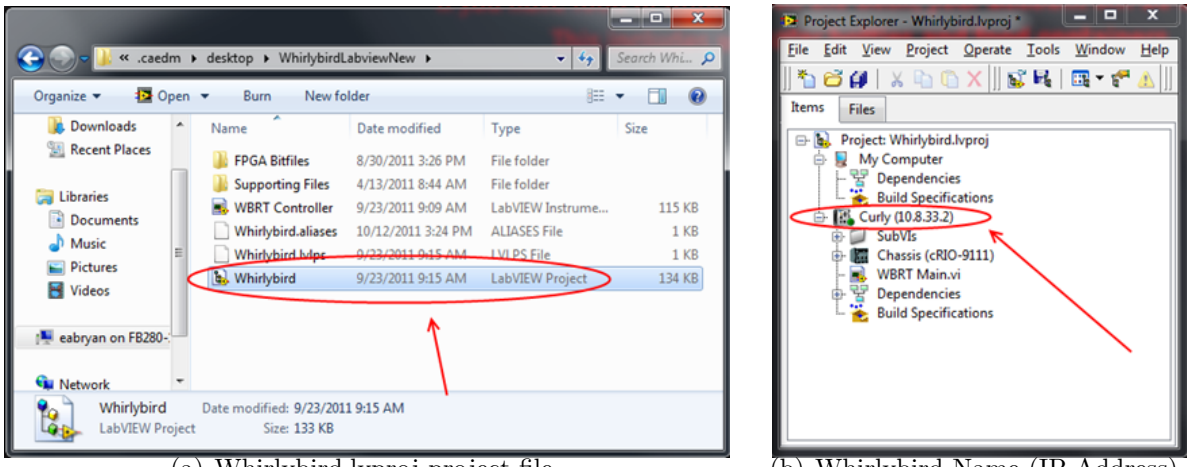
Appendix B

Using the Hardware

RWB: We may need to add a quick introduction to Labview and Labview graphical programming. Perhaps in another appendix.

B.1 Getting Started

1. Download the whirlybird project file from the website.



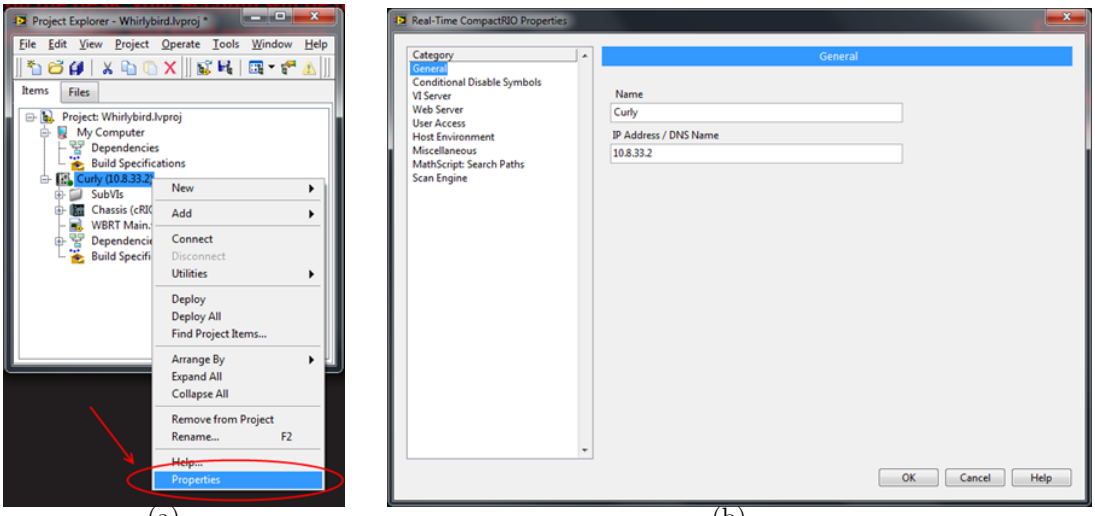
(a) Whirlybird.lvproj project file (b) Whirlybird Name (IP Address)

Figure B.1: Open LabVIEW project file

2. Locate the LabVIEW project file **Whirlybird.lvproj**. Double-click and it will open in LabVIEW. Figure B.1(a).

3. Verify that the name and IP address of the whirlybird you intend to control is listed in the LabVIEW project tree. Figure B.2(b).

4. If the name and IP address are not correct, right-click on the name, select properties, and then change the 'Name' and 'IP Address/DNS Name' entries. See figures B.2(a) and B.2(b)



(a) (b)

Figure B.2: Change whirlybird IP address and name

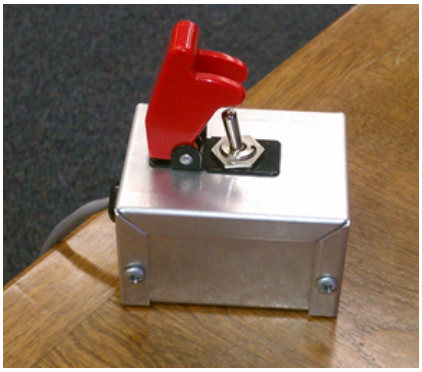


Figure B.3: Safety switch

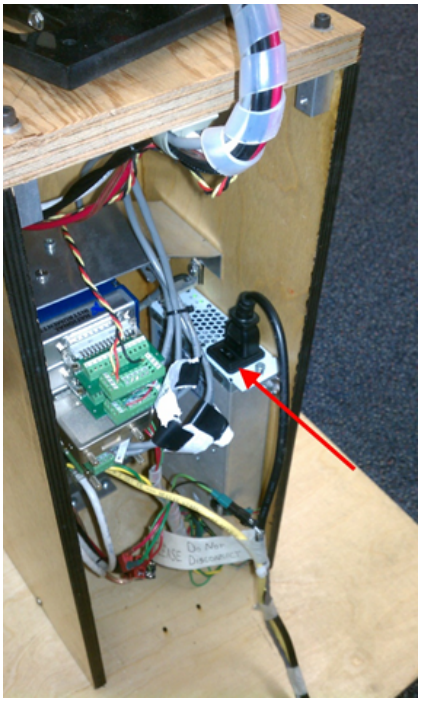


Figure B.4: Power supply switch

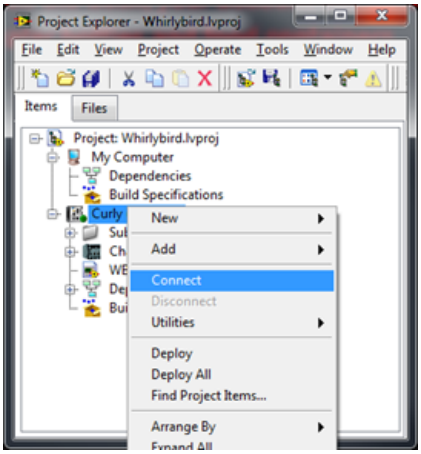


Figure B.5: Connect to whirlybird

5. Verify that the toggle safety switch cover is up and the switch is closed. Figure B.3.

6. Turn on the whirlybird power supply. Figure B.4.

7. Connect LabVIEW to the whirlybird by right clicking on the name of the whirlybird and selecting "Connect". Figure B.5.

   - If LabVIEW is unable to connect to the whirlybird, check to see that the whirlybird is connected to power and the network cable is plugged in.
   - Be sure the safety switch is closed when the whirlybird power supply is turned on.

8. Once the whirlybird is connected to LabVIEW, return to the project tree and open 'WBRT Main.vi'. Figure B.6

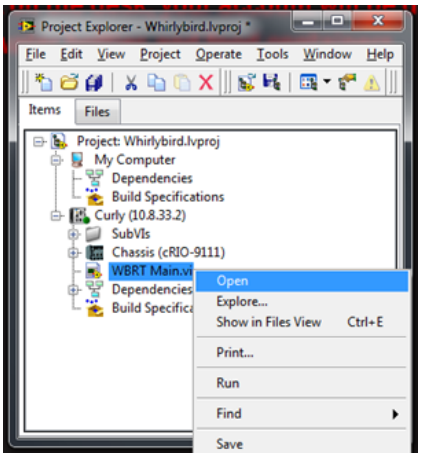


Figure B.6: Open 'WBRT Main.vi'

9. Wait for the 'Main' window to open. Figure B.7

10. To start the hardware, click on the arrow in the upper left corner of the 'Main' window (Figure B.8). If you are asked to save any files, click yes.

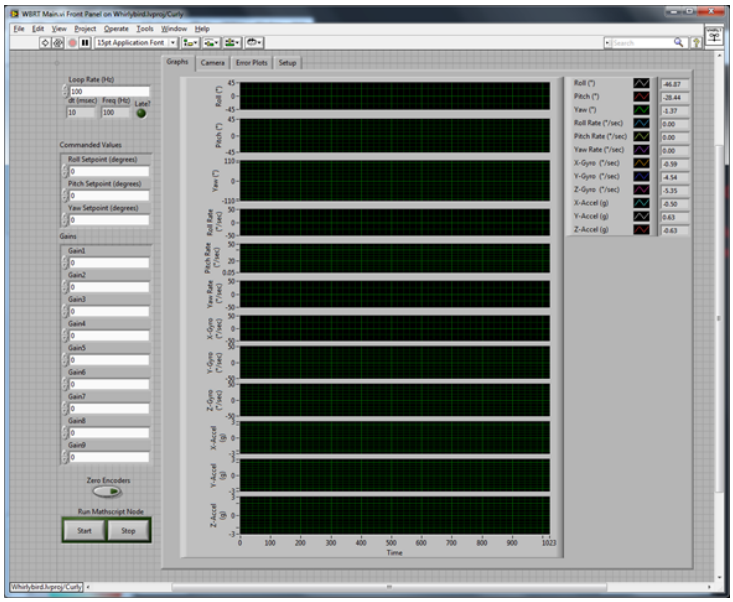


Figure B.7: WBRT Main.vi

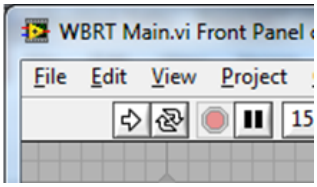


Figure B.8: Click the arrow to begin

11. Once the software is running you will see that the center of the Main.vi window contains state plots that are now plotting in real time. The encoders need to be set to zero. Prop the head up on a chair making sure that the pitch, roll and yaw angles are all at the zero point, as seen in Figure B.9. Zero the encoders by clicking on the button labeled "Zero Encoders" (Figure B.12).

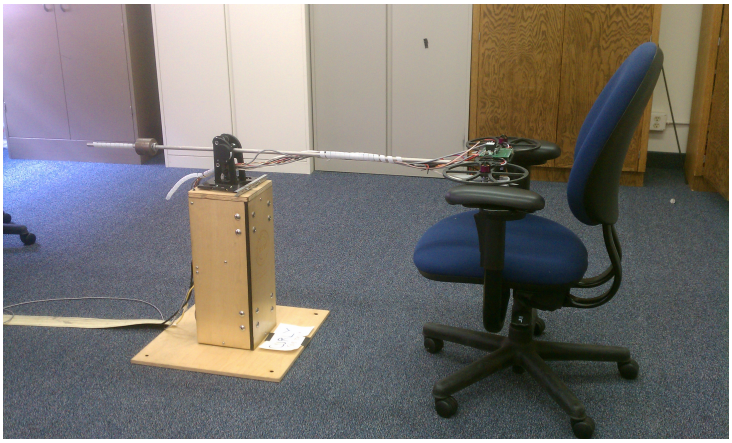


Figure B.9: Setting encoders equal to zero.

12. Now that the encoders are set to zero, remove the chair. It is helpful to set the field labeled "Pitch Setpoint (degrees)" to -20 degrees, and all other Setpoints to zero degrees.

13. Make sure that the roll angle is zero before starting the whirlybird and that the gains are correctly entered.. Click on the Start button to start the whirlybird.

B.2 WBRT Main.vi Front Panel

The WBRT Main.vi Front Panel can be divided into three parts. The left most part contains the run-time variable fields and buttons to start the Mathscript Code and zero the encoders (Figures B.10, B.11, & B.12). Figure B.10 shows the fields where the gains can be entered and adjusted while the whirlybird is flying.

From the Main.vi Front Panel the user may adjust the loop rate. The loop rate can be changed before or during the simulation. The default Loop Rate is 100Hz (Figure B.11). The center of the Main.vi Front Panel is filled

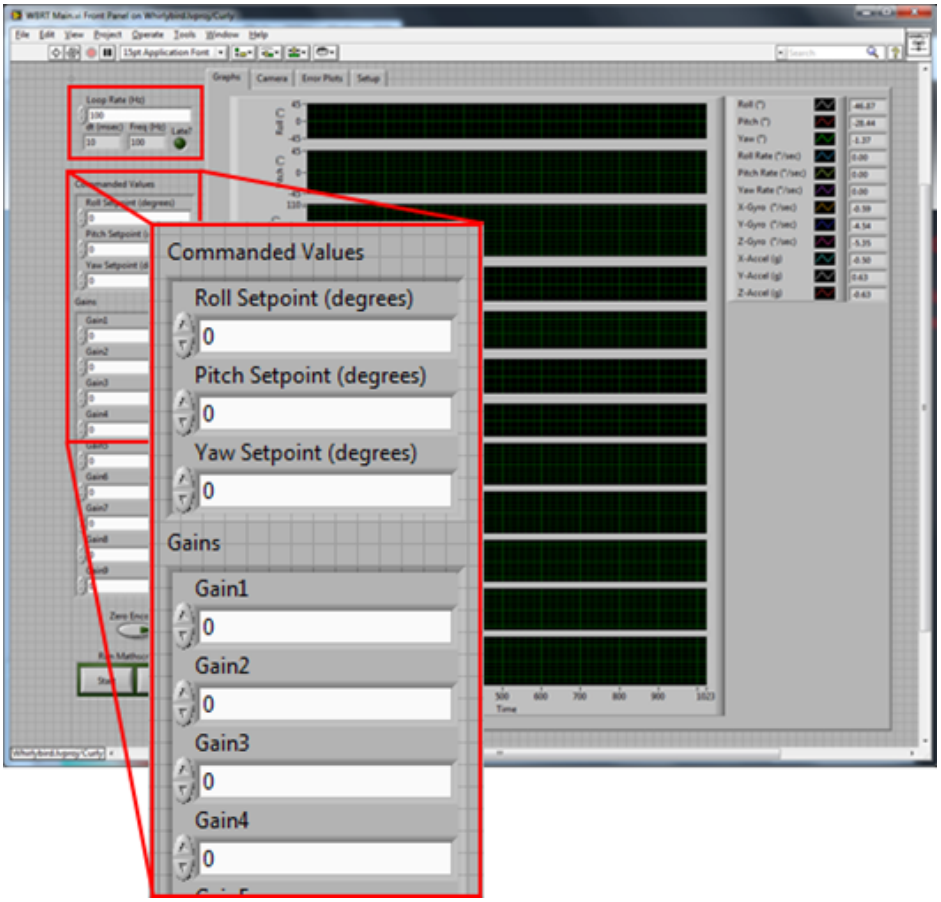


Figure B.10:

with scrolling plots of the whirlybird states, gyro, and accel measurements.

The right side of the Main.vi Front Panel has the state, gyro, and accel instantaneous values (Figure B.13).

B.3 Turn off / Emergency Shut Down

The whirlybird has built-in "power up" and "power down" motor behaviors to reduce the possibility of damaging the whirlybird when turning it on or

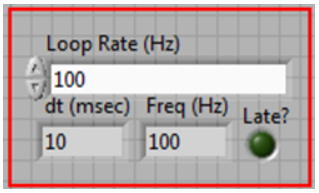


Figure B.11: Loop Rate of Controller.

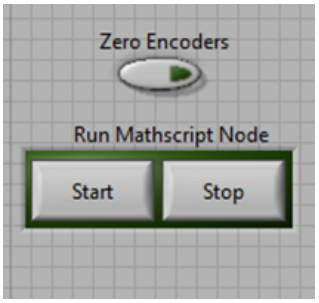


Figure B.12: Zero Encoders & Start/Stop Buttons

off. Unless there is risk of bodily injury or damage of equipment, follow the steps in section B.3.1.

B.3.1 Turn Off

To turn off the whirlybird:

1. Click on the "Stop" button (Figure B.12) and wait for the whirlybird motors to stop.
2. Once they have stopped, disconnect from the whirlybird by right clicking on the name of the whirlybird in the project tree and selecting "Disconnect".
3. Turn off the power supply.

B.3.2 Emergency Shut Down

There are various reasons why you may need to turn the whirlybird immediately off, possible bodily injury or damage of equipment to name two. To turn off the whirlybird immediately, close the red cover over the toggle safety

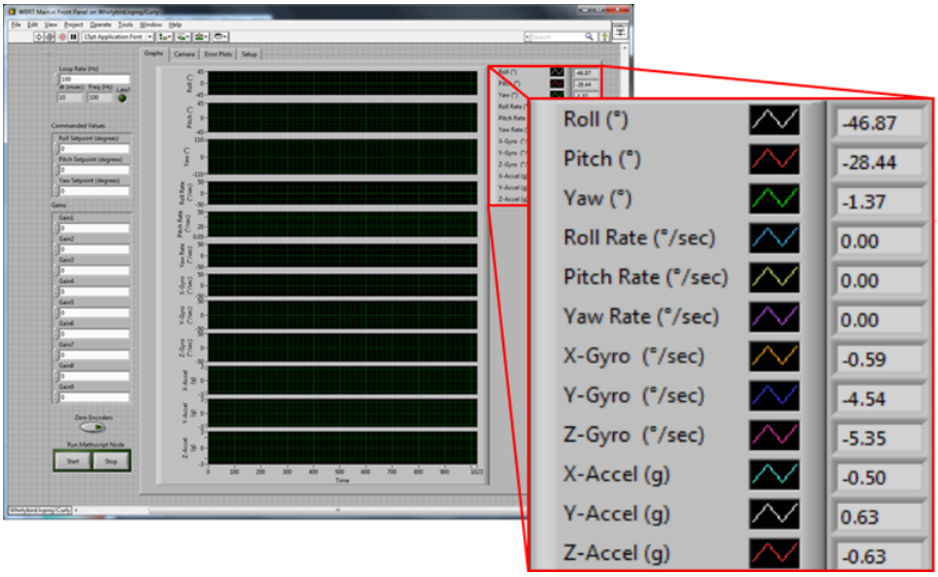


Figure B.13: Main.vi Front Panel: Whirlybird States

switch. The motors will immediately stop spinning and the whirlybird will disconnect.

When the emergency switch is used to shut the whirlybird down, there is a short procedure to follow to reconnect and begin again. To restart the whirlybird after an emergency shutdown,

1. Open the safety switch cover and close the switch.

2. Cycle the power supply by turning it off and waiting for the internal fan to stop spinning, then turning it back on.

3. Reconnect LabVIEW to the whirlybird. Go to the Project Explorer and right-click on the name of the whirlybird and select "Connect".

B.4 WBRT Controller.vi Block Diagram

B.4.1 Open Controller

1. To open the controller, double click on 'WBRT Controller.vi' in the Whirlybird LabVIEW directory (Figure B.14).

2. The controller front panel will open in LabVIEW (Figure B.15).

3. Press 'Control + E' to open the block diagram. The block diagram should look something like figure B.16.

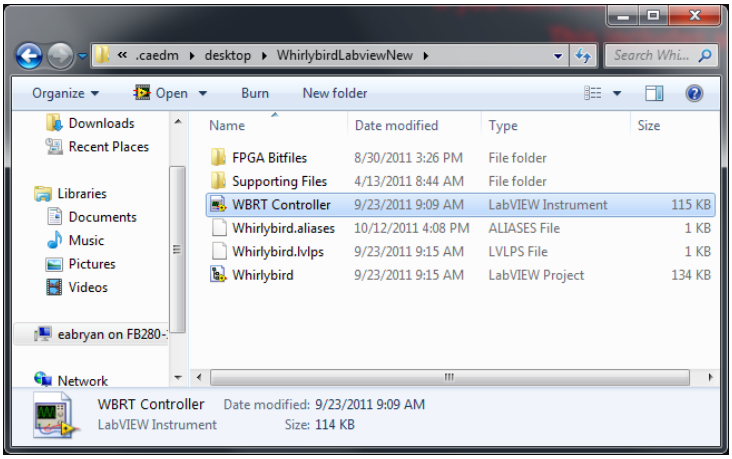


Figure B.14:

B.4.2 Controller Block Diagram explanation

Inside the block diagram (Figure B.16) there is a blue outlined text box where the control code is entered. The programming language is MathScript and is very similar to MATLAB. Add your control code into the Controller.vi Block Diagram at the location indicated in the script file.

You will notice that the input variables and the output variables are on the left and right side of the text box, respectively.

- The first 6 variables on the left-hand side of the script window describe the states of the whirlybird in units of radians and radians/sec.

- The next 6 variables describe the whirlybird state information as obtained from the onboard inertial measurement unit (IMU).

- The variables labeled **RollSetpoint**, **PitchSetpoint**, and **YawSetpoint** are to be used as the roll, pitch, and yaw commanded values. You can set these up to be in radians or degrees. These values are read in from the 'Main.vi' window.

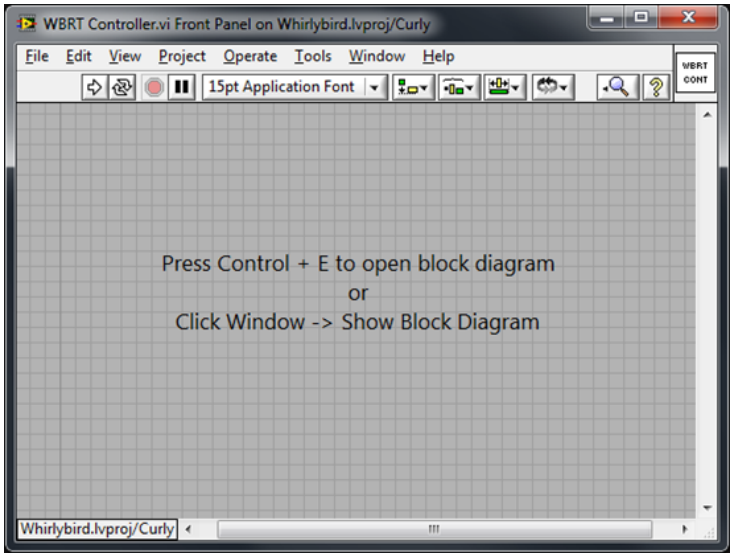


Figure B.15:

- **TrajectoryData** is meant to be a vector of desired attitude values to be read in from a text file. We are not currently using this variable.
- **dt** is set in the Main.vi by changing the 'Loop Rate (Hz)' variable and is in units of milliseconds.
- **ElapsedTime** is the time since your program has been running in units of milliseconds
- **Gain1** through **Gain9** are for your own use. These values, like **dt**, are read in from the 'Main.vi' window and can be changed while the whirlybird is running.
- **Variable1** through **Variable4** are for your own use. These variables behave like persistent variables and will retain their values between calls.
- **CommandL** and **CommandR** are the PWM outputs to the motors, these need to be in the range of 0-100.

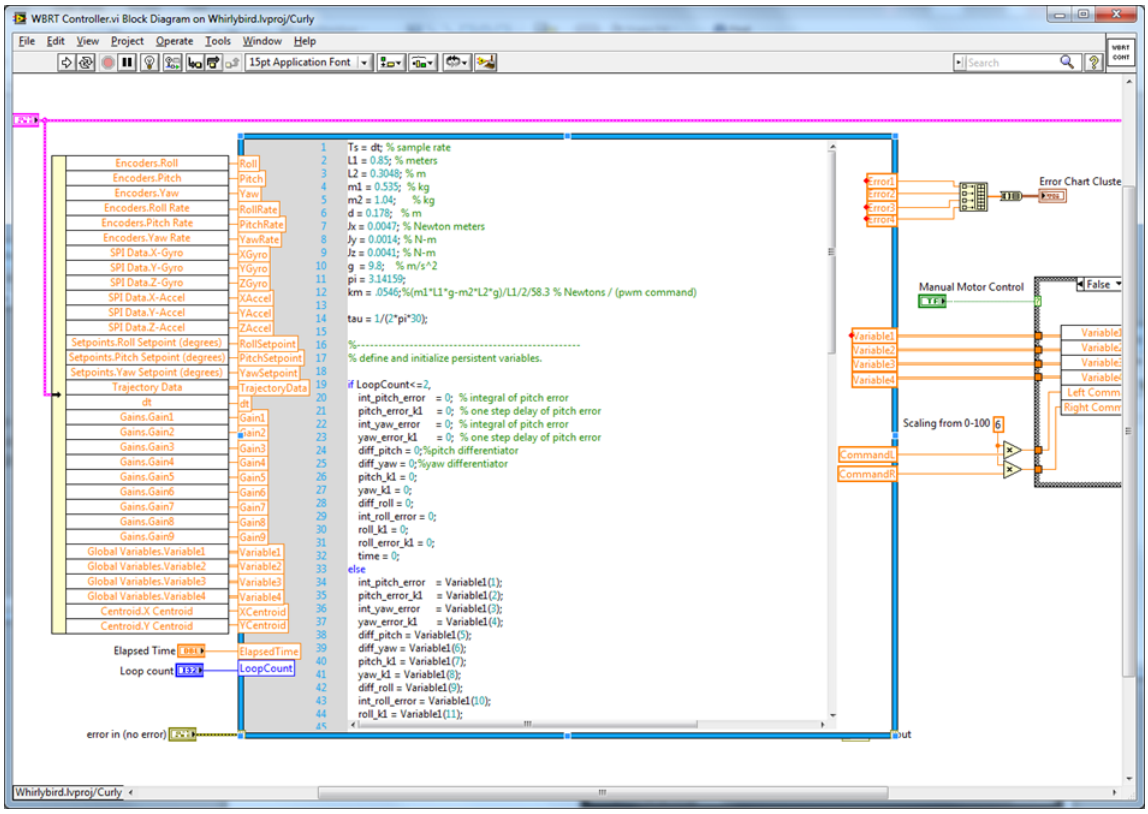


Figure B.16: WBRT Controller.vi

Bibliography