CHAPTER 10

Fault Detection

Introduction

Fault detection is the process of detecting failures, also known as faults, in a dynamical system. It is an important area for systems that are supposed to operate without human supervision. There are many ways of detecting failures. The simplest is using boolean logic to check against fixed thresholds. For example, you might check an automobile's speed against a speed limit. Other methods include fuzzy logic, parameter estimation, expert systems, statistical analysis, and parity space methods. In this section, we will implement one type of fault detection system, a detection filter. This is based on linear filtering. The detection filter is a state estimator tuned to detect specific failures. We will design a detection filter system for an air turbine. We will also show how to build a graphical user interface (GUI) as a front end to the fault detection simulation.

10.1 Modeling an Air Turbine

Problem

We need to make a numerical model of an air turbine to demonstrate detection filters.

Solution

Write the equations of motion for an air turbine. We will use a linear model of the air turbine to simplify the detection filter design. This will allow us to model the system with a linear state space model.

M. Paluszek and S. Thomas, *MATLAB Recipes*,
https://doi.org/10.1007/978-1-4842-6124-8_10

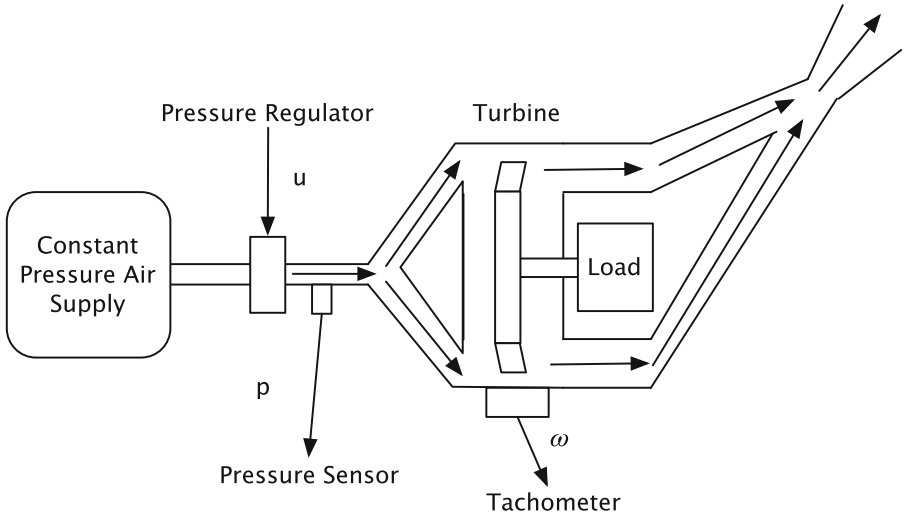

Figure 10.1: *Air turbine. The arrows show the airflow. The air flows through the turbine blade tips causing it to turn.*

How It Works

Figure 10.1 shows an air turbine.[1] It has a constant pressure air supply. We can control the valve from the air supply, the pressure regulator, to control the speed of the turbine. The air flows past the turbine blades causing it to turn. The control needs to adjust the air pressure to handle variations in the load. We measure the air pressure p downstream from the valve, and we also measure the rotational speed of the turbine ω with a tachometer.

The dynamical model for the air turbine is

$$\begin{bmatrix} \dot{p} \\ \dot{\omega} \end{bmatrix} = \begin{bmatrix} -\frac{1}{\tau_p} & 0 \\ \frac{K_t}{\tau_t} & -\frac{1}{\tau_t} \end{bmatrix} \begin{bmatrix} p \\ \omega \end{bmatrix} + \begin{bmatrix} \frac{K_p}{\tau_p} \\ 0 \end{bmatrix} u \tag{10.1}$$

This is a state space system:

$$\dot{x} = ax + bu \tag{10.2}$$

where

$$a = \begin{bmatrix} -\frac{1}{\tau_p} & 0 \\ \frac{K_t}{\tau_t} & -\frac{1}{\tau_t} \end{bmatrix} \tag{10.3}$$

$$b = \begin{bmatrix} \frac{K_p}{\tau_p} \\ 0 \end{bmatrix} \tag{10.4}$$

[1] PhD thesis of Jere Schenck Meserole, "Detection Filters for Fault-Tolerant Control of Turbofan Engines," Massachusetts Institute of Technology, Department of Aeronautics and Astronautics, 1981.

The state vector is

$$\begin{bmatrix} p \\ \omega \end{bmatrix} \tag{10.5}$$

The pressure downstream from the regulator is equal to K_{pu} when the system is in equilibrium. τ_p is the regulator time constant, and τ_t is the turbine time constant. The turbine speed is K_{tp} when the system is in equilibrium. The tachometer measures ω, and the pressure sensor measures p. The load is folded into the time constant for the turbine.

The code for the right-hand side of the dynamical equations is shown in the following. Only one line of code is needed. The rest returns the default data structure. The simplicity of the model is due to its being a state space model. The number of states could be large, yet the code would not change.

RHSAirTurbine.m

```
27  function xDot = RHSAirTurbine( ~, x, d )
28
29  % Default data structure
30  if( nargin < 1 )
31    kP   = 1;
32    kT   = 2;
33    tauP = 10;
34    tauT = 40;
35    c    = eye(2);
36    b    = [kP/tauP;0];
37    a    = [-1/tauP 0; kT/tauT -1/tauT];
38
39    xDot = struct('a',a,'b',b,'c',c,'u',0);
40    if( nargout == 0)
41      disp('RHSAirTurbine struct:');
42    end
43    return
44  end
```

The response to a step input for u is shown in Figure 10.2. The pressure settles faster than the turbine speed. This is due to the turbine time constant and the lag in the pressure change. The residuals are very small because there are no failures.

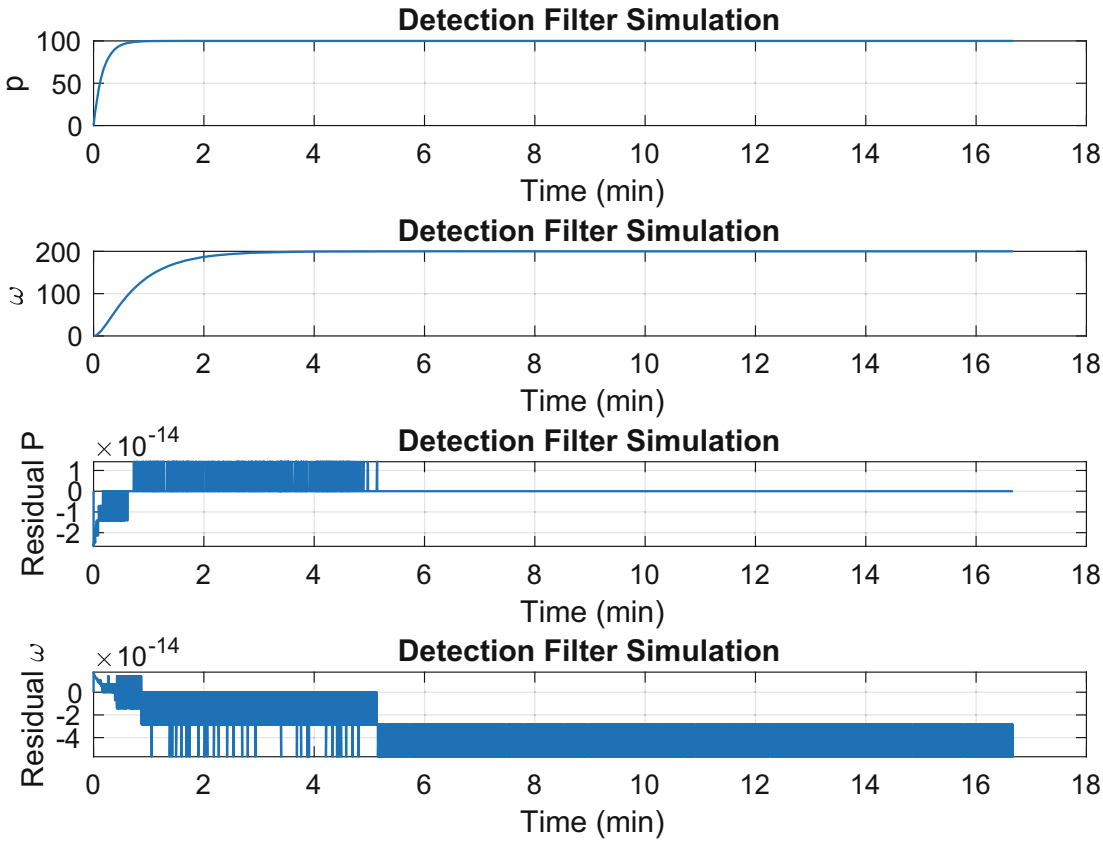

Figure 10.2: *Air turbine response to a step pressure regulator input. The residuals are zero as expected.*

10.2 Building a Detection Filter

Problem

We want to build a system to detect failures in our air turbine using the linear model developed in the previous recipe.

Solution

We will build a detection filter that detects pressure regulator failures and tachometer failures. Our plant model (continuous a, b, and c state space matrices) will be an input to the filter building function.

How It Works

The detection filter is an estimator with a specific gain matrix that multiplies the residuals. The residuals are the difference between the estimated outputs and the outputs:

$$\begin{bmatrix} \dot{\hat{p}} \\ \dot{\hat{\omega}} \end{bmatrix} = \begin{bmatrix} -\frac{1}{\tau_p} & 0 \\ \frac{K_t}{\tau_t} & -\frac{1}{\tau_t} \end{bmatrix} \begin{bmatrix} \hat{p} \\ \hat{\omega} \end{bmatrix} + \begin{bmatrix} \frac{K_p}{\tau_p} \\ 0 \end{bmatrix} u + \begin{bmatrix} d_{11} & d_{12} \\ d_{21} & d_{22} \end{bmatrix} \begin{bmatrix} p - \hat{p} \\ \omega - \hat{\omega} \end{bmatrix} \tag{10.6}$$

where $\hat{p}$ is the estimated pressure and $\hat{\omega}$ is the estimated angular rate of the turbine. The D matrix is the matrix of detection filter gains. These feedback the residuals, the difference between the measured and estimated states, into the detection filter. The residual vector is

$$r = \begin{bmatrix} p - \hat{p} \\ \omega - \hat{\omega} \end{bmatrix} \tag{10.7}$$

The residuals are the difference between the measured values and the estimated values. The D matrix needs to be selected so that this vector tells us the nature of the failure. The gains should be selected so that

1. The filter is stable.
2. If the pressure regulator fails, the first residual $p - \hat{p}$ is nonzero, but the second remains zero.
3. If the turbine fails, the second residual $\omega - \hat{\omega}$ is nonzero, but the first remains zero.

A gain matrix is

$$D = a + \begin{bmatrix} \frac{1}{\tau_1} & 0 \\ 0 & \frac{1}{\tau_2} \end{bmatrix} \tag{10.8}$$

The time constant τ_1 is the pressure residual time constant. The time constant τ_2 is the tachometer residual time constant. In effect, we cancel out the dynamics of the plant and replace them with decoupled detection filter dynamics. These time constants should be shorter than the time constants in the dynamical model so that we detect failures quickly. However, they need to be at least twice as long as the sampling period to prevent numerical instabilities.

We will write a function with three actions, an initialize case, an update case, and a reset case. `varargin` is used to allow the three cases to have different input lists. The function signature is

DetectionFilter.m

```
49  function d = DetectionFilter( action, varargin )
```

The header and syntax for `DetectionFilter` are shown as follows. We used LaTeX equations to describe the function.

```
%% DETECTIONFILTER Builds and updates a linear detection filter.
%% Forms
%   d = DetectionFilter( 'initialize', d, tau, dT )
%   d = DetectionFilter( 'update', u, y, d )
%   d = DetectionFilter( 'reset', d )
%
%% Description
% The detection filter gain matrix d is designed during the initialize
% action. The continuous matrices are then discretized using the internal
% function CToDZOH. The esimated state and residual vectors are initialized
% to the size dictated by a. During the update action, the residuals and
% new estimated state are calculated and stored in the data structure d.
%
% The residuals calculation is
%
% $$r   = y - c\hat{x}$$
%
% The estimated state calculated with the detection filter gains is
%
% $$\hat{x}_{k+1} = a*\hat{x} + +b*u + d*r$$
%
%% Inputs
%   action      (1,:) 'initialize' or 'update'
%   d           (.)   Data structure
%                       .a (:,:) State space continuous a matrix
%                       .b (:,1) State space continuous b matrix
%                       .c (:,:) State space continuous c matrix
%   tau         (:,1) Vector of time constants
%   dT          (1,1) Time step
%   u           (:,1) Actuation input
%   y           (:,1) Measurement vector
%
%% Outputs
%   d           (.)   Updated data structure
%                       .a (:,:) State space discrete a matrix
%                       .b (:,1) State space discrete b matrix
%                       .c (:,:) State space discrete c matrix
%                       .d (:,:) Detection filter gain matrix
%                       .x (:,1) Estimated states
%                       .r (:,1) Residual vector
```

The filter is built and initialized in the following code in `DetectionFilter`. The continuous state space model of the plant, in this case, our linear air turbine model, is an input. The selected time constants τ are also an input, and they are added to the plant model as in Equation 10.8. The function discretizes the plant `a` and `b` matrices and the computed detection filter gain matrix `d`.

```
48
49 function d = DetectionFilter( action, varargin )
50
51 switch lower(action)
52   case 'initialize'
53     d   = varargin{1};
54     tau = varargin{2};
55     dT  = varargin{3};
56
57     % Design the detection filter
58     d.d = d.a + diag(1./tau);
59
60     % Discretize both
61     d.d        = CToDZOH( d.d, d.b, dT );
62     [d.a, d.b] = CToDZOH( d.a, d.b, dT );
63
64     % Initialize the state
66     d.x = zeros(m,1);
67     d.r = zeros(m,1);
```

The update for the detection filter is in the same function, as the next action in the `switch` statement. Note the equations implemented as described in the header.

```
69   case 'update'
70     u   = varargin{1};
71     y   = varargin{2};
72     d   = varargin{3};
73     r   = y - d.c*d.x;
74     d.x = d.a*d.x + +d.b*u + d.d*r;
75     d.r = r;
```

Finally, we create a reset action to allow us to reset the residual and state values for the filter in between simulations. After this action, we end the `switch` statement.

```
77   case 'reset'
78     d   = varargin{1};
79     m   = size(d.a,1);
80     d.x = zeros(m,1);
81     d.r = zeros(m,1);
82 end
```

10.3 Simulating the Fault Detection System

Problem

We want to simulate a failure in the plant and demonstrate the performance of the failure detection.

Solution

We will build a MATLAB script that designs the detection filter using the function from the previous recipe and then simulates it with a user selectable pressure regulator or tachometer failure. The failure can be total or partial.

How It Works

The script designs a detection filter using `DetectionFilter` from the previous recipe and implements it in a loop. Runge-Kutta integration propagates the continuous domain right-hand side of the air turbine, `RHSAirTurbine`. The detection filter is discrete time.

The script has two scale factors `uF` and `tachF` that multiply the regulator input and the tachometer output to simulate failures. Setting a scale factor to zero is a total failure, and setting it to one indicates that the device is working perfectly. If we fail one, we expect the associated residual to be nonzero and the other to stay at zero. Failures can be any number between zero and one. Partial failures are not necessarily related to a specific mechanical failure but are useful for testing the system.

DetectionFilterSim.m

```
%% Simulation of a detection filter
% Simulates detecting failures of an air turbine.
% An air turbine has a constant pressure air source that sends air
% through a duct that drives the turbine blades. The turbine is
% attached to a load.
%
% The air turbine model is linear. Failures are modeled by multiplying
% the regulator input and tachometer output by a constant. A constant
% of 0 is a total failure and 1 is perfect operation.

%% User inputs

% Failures. Set to any number betweem 0 and 1 is 0 is total failure. 1
    is working perfectly.
% uF scales the actuation u. tachF scales the rate measurement.
uF    = 0;
tachF = 1;

% Time constants for failure detection
tau1 = 0.3; % sec
```

```
tau2 = 0.3; % sec

% End time
tEnd = 1000; % sec

% State space system
d = RHSAirTurbine;

%% Initialization
dT = 0.02; % sec
n  = ceil(tEnd/dT);

% Initial state
x = [0;0];

%% Detection Filter design
dF = DetectionFilter('initialize',d,[tau1;tau2],dT);

%% Run the simulation

% Control. This is the regulator input.
u = 100;

% Plotting array
xP = zeros(4,n);
t  = (0:n-1)*dT;

for k = 1:n
  % Measurement vector including measurement failure
  y       = [x(1);tachF*x(2)]; % Sensor failure
  xP(:,k) = [x;dF.r];

  % Update the detection filter
  dF = DetectionFilter('update',u,y,dF);

  % Integrate one step
  d.u = uF*u; % Actuator failure
  x   = RungeKutta( @RHSAirTurbine, t(k), x, dT, d );
end

%% Plot the states and residuals
[t,tL] = TimeLabel(t);
yL     = {'p' '\omega' 'Residual P' 'Residual \omega' };
tTL    = 'Detection Filter Simulation';
PlotSet( t, xP,'x label',tL,'y label',yL,'plot title',tTL,'figure title
    ',tTL)
```

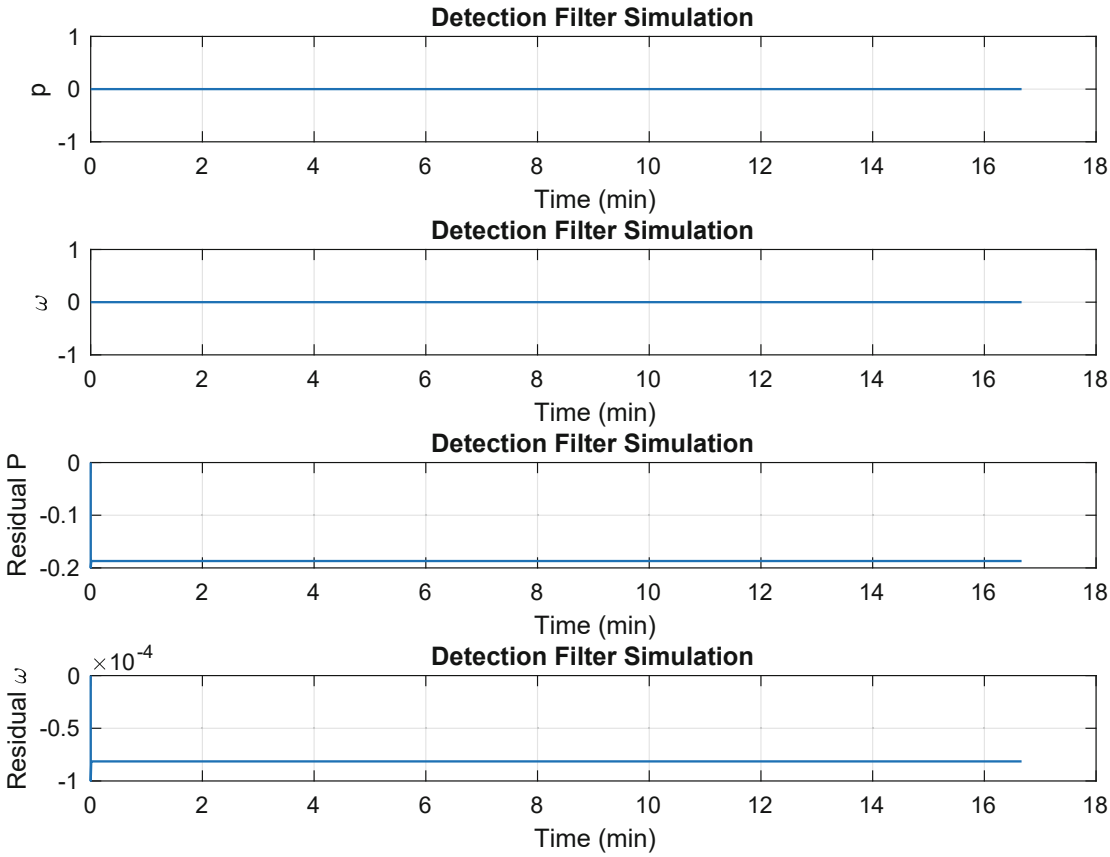

Figure 10.3: *Air turbine response to a failed regulator.*

In Figure 10.3, the regulator fails and its residual is nonzero. In Figure 10.4, the tachometer fails and its residual is nonzero. The residuals show what has failed clearly. Simple boolean logic (i.e., if end statements) are all that is needed.

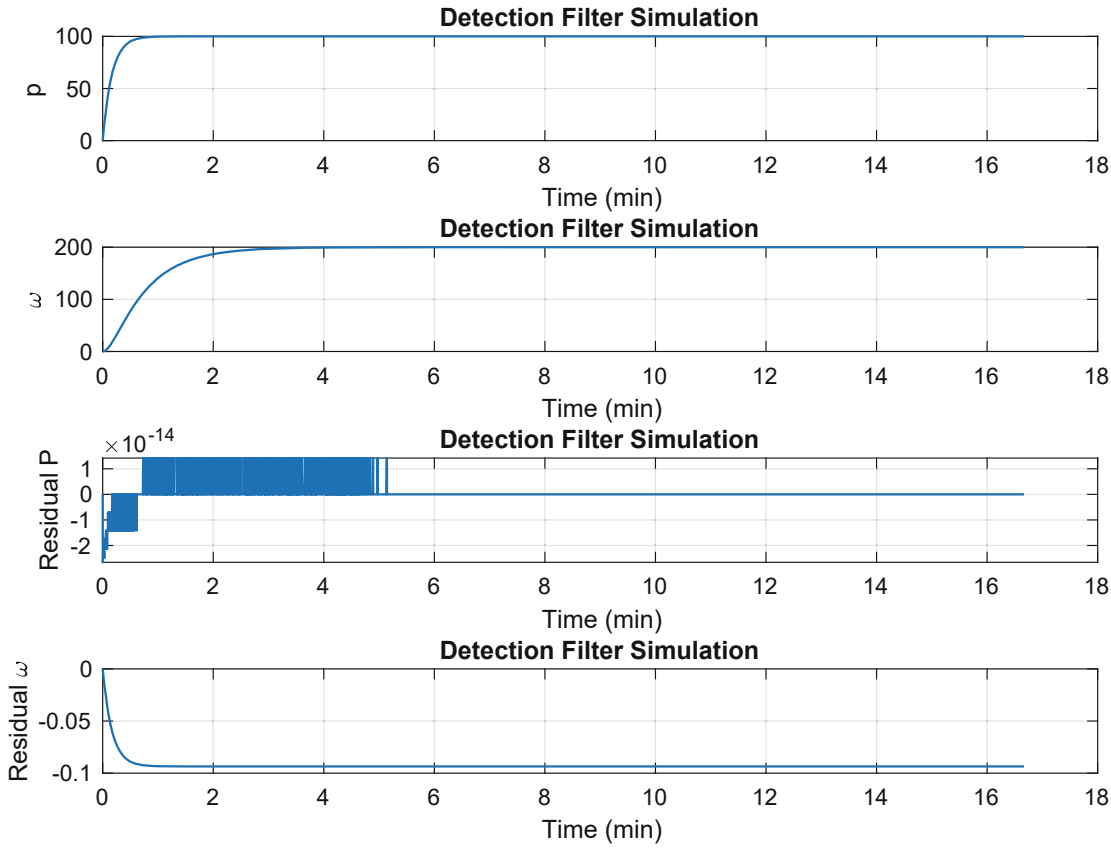

Figure 10.4: *Air turbine response to a failed tachometer.*

10.4 Building a GUI for the Detection Filter Simulation

Problem

We want a GUI to provide a graphical interface to the fault detection simulation that will allow us to evaluate the filter's performance.

Solution

We will use the MATLAB App Designer to build a GUI that will allow us to

1. Set the residual time constants
2. Set the end time for the simulation

3. Set the pressure regulator input
4. Introduce a pressure regulator or tachometer fault at any time
5. Display the states and residuals in a plot

How It Works

The MATLAB App Designer is invoked by typing `appdesigner` at the command line. There are several options for GUI templates, or a blank GUI; we will start from the GUI with `uicontrols`. First, let's make a list of the controls we will need from our desired features list earlier:

- Edit boxes for the simulation duration, residual time constants τ_1 and τ_2, pressure regulator setting u
- Edit boxes for the pressure regulator and tachometer fault parameters, with buttons for sending the newly commanded values to the simulation
- Text box for displaying the calculated detection filter gains
- Run button for starting a simulation
- Two plot axes

In order to change the fault parameters while the simulation is running, we will need the loop to be checking a variable that can be externally set by the GUI. We can do this using global variables.

There are several templates that we can use. We will start with the basic blank template. Type appdesigner in the command window. Figure 10.5 shows the interface.

Double-click the blank app template.

Add the app `DFGUI`. It will appear in your folder as `DFGUI.mlapp`.

Add the following to the blank template:

1. Parameter input boxes
   (a) Duration
   (b) Input
   (c) Tau 1
   (d) Tau 2
   (e) Gains (2-by-2 matrix)

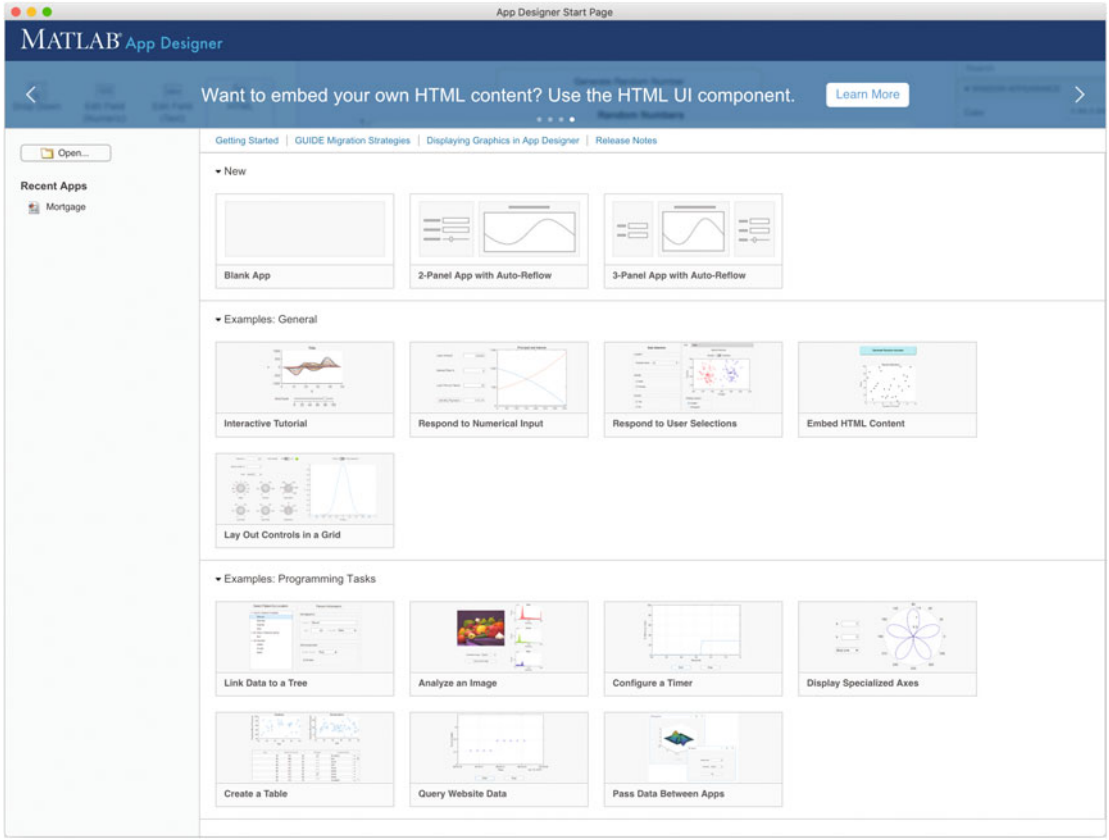

Figure 10.5: *The interface to appdesigner.*

2. Failure input boxes
    (a) Tachometer
    (b) Input
    (c) Send button for tachometer
    (d) Send button for input

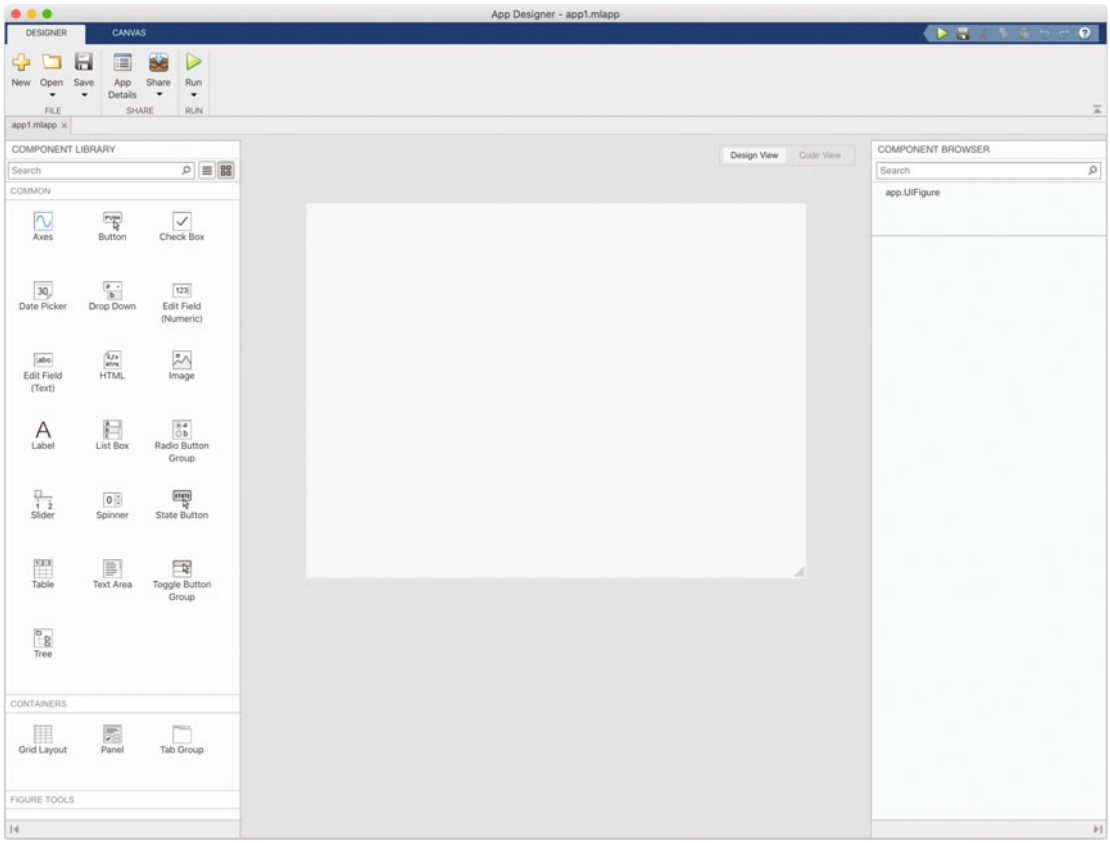

Figure 10.6: *Snapshot of the blank app.*

3. Calculate button
4. Reset button
5. State plot
6. Residual plot

You add items by dragging and dropping them on the window from the items on the left-hand side. We use numeric for the input text boxes. Figure 10.7 shows the completed interface. There are four push buttons.

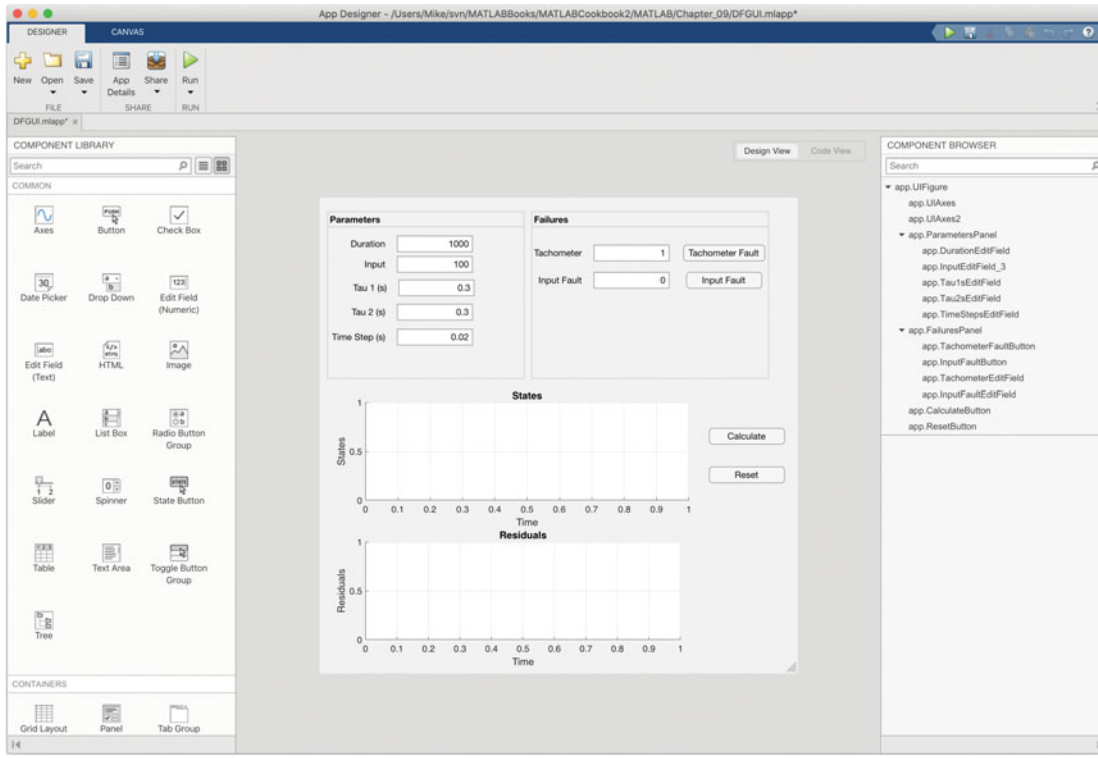

Figure 10.7: *Snapshot of the app after the interface is done.*

You can add information about the app. Figure 10.8 shows the window for app information. The app appears in the app menu as shown in Figure 10.9.

We select the callback for calculate. The App Designer highlights where the code should go. We copy relevant code from the simulation script. We get the inputs from the text boxes.

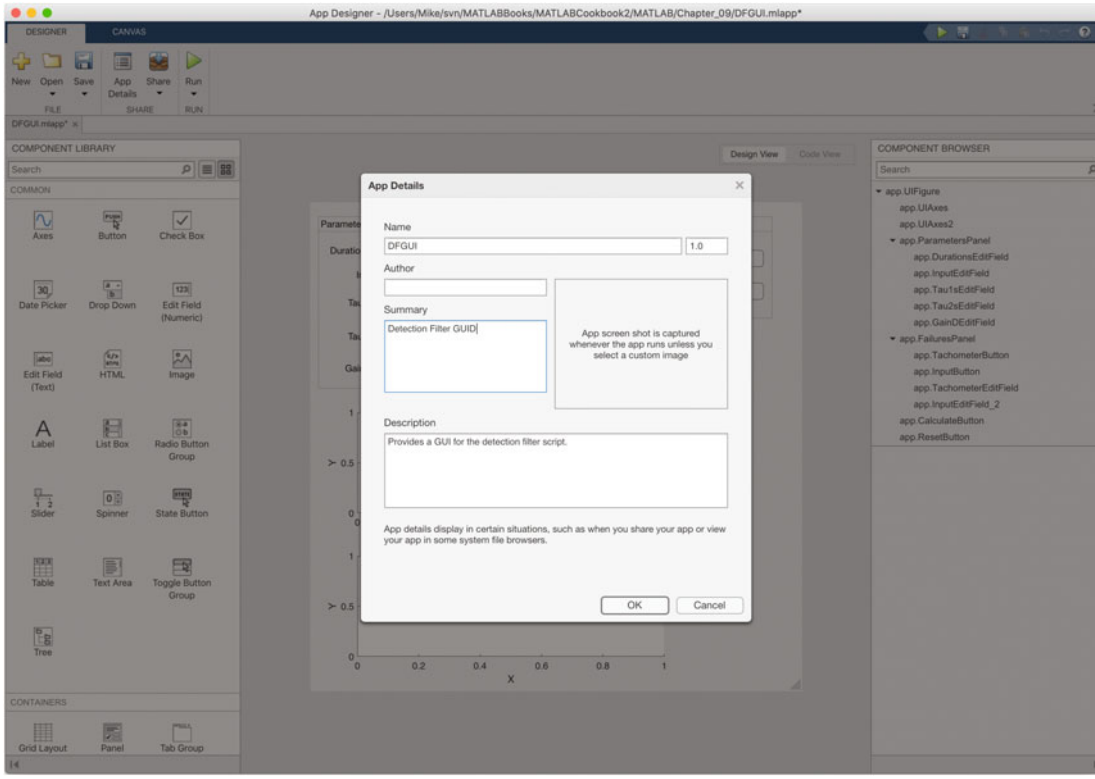

Figure 10.8: *App Details let you add information about the app for users.*

Figure 10.10 shows the code. You access parameters from the text boxes using `app.xxx.Value`. For all plot-related functions, you need to add the axes handle using `app.UAxes`. or `app.UAxe2s`.

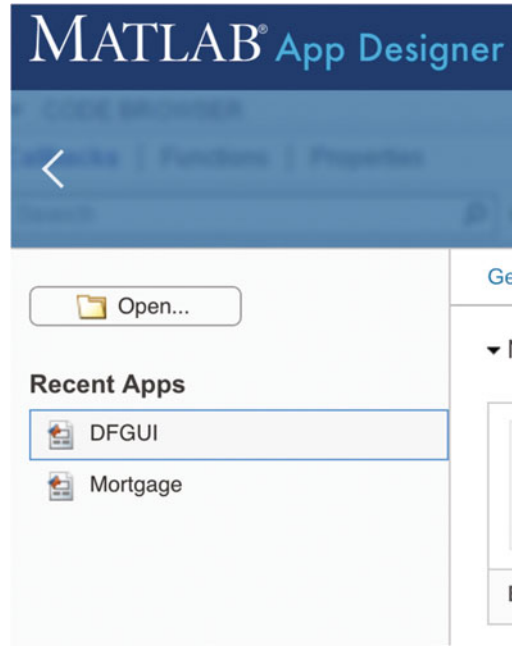

Figure 10.9: *The app appears in the app menu. You get to this window by hitting the design app button.*

Figure 10.11 shows a debugger breakpoint. You have full access to the debugger in App Designer. You will also see MATLAB warnings on the right.

Figure 10.12 shows the app after a run.

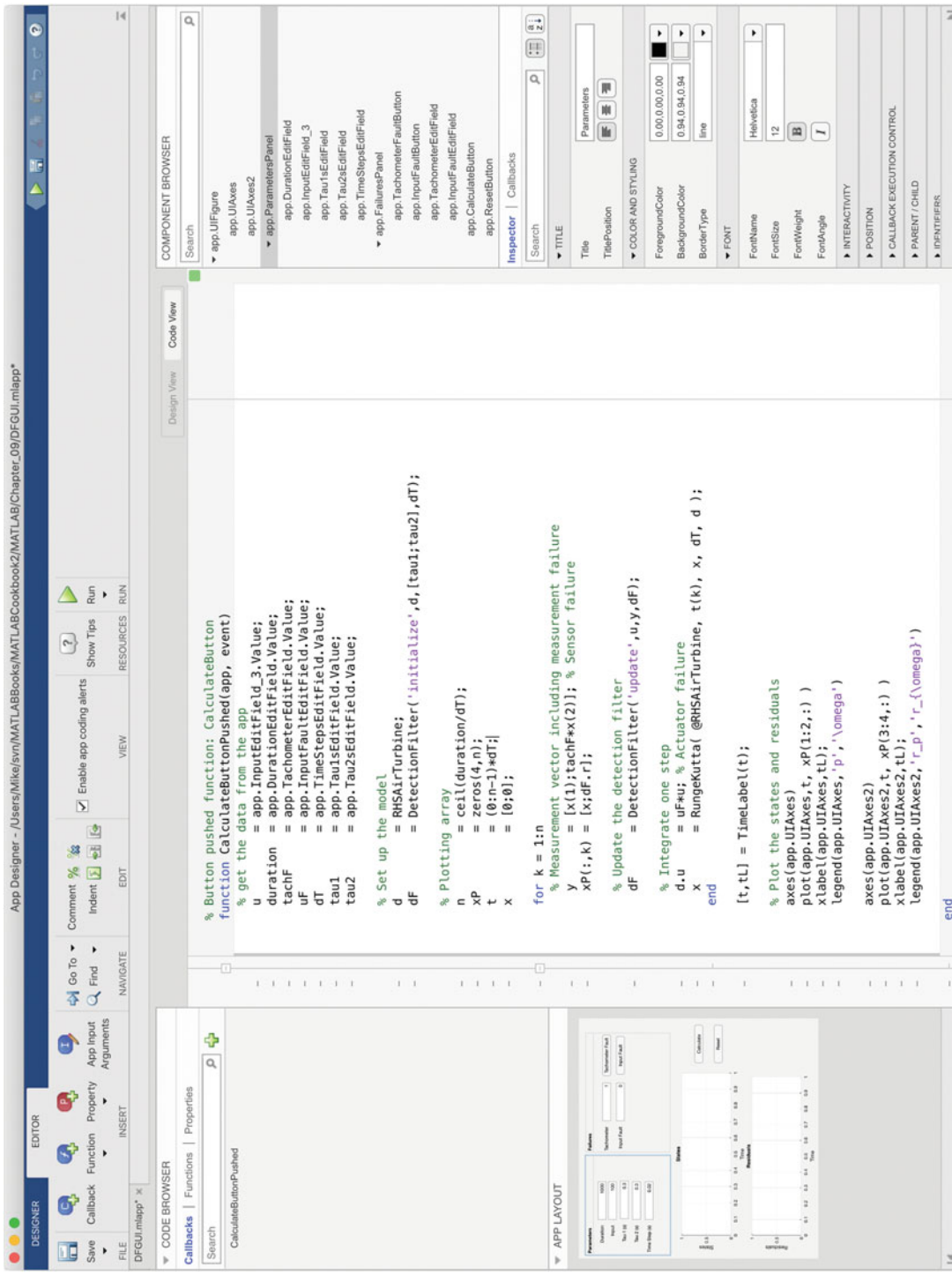

Figure 10.10: *The light area is where the code goes. The code is from the simulation script.*

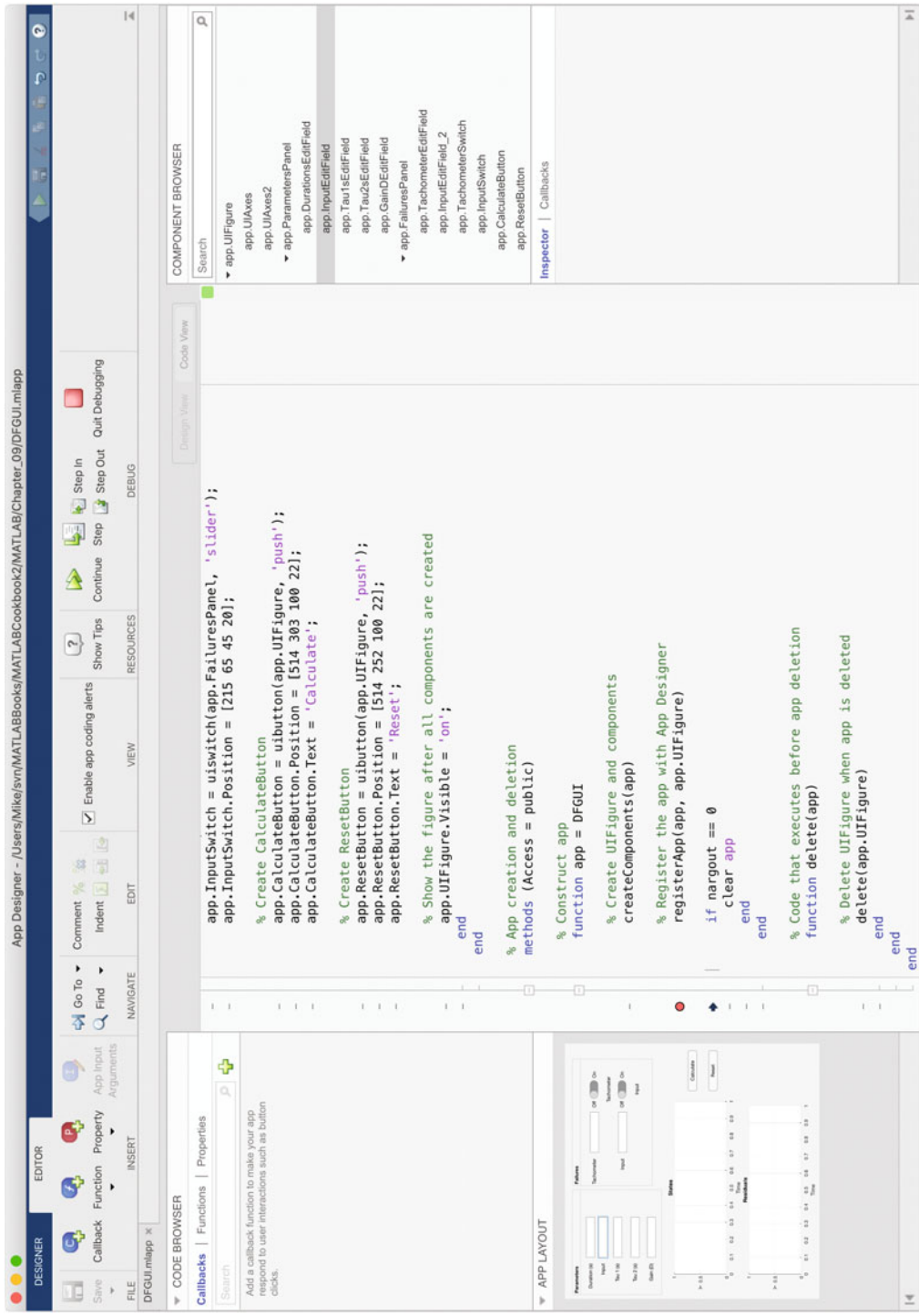

Figure 10.11: *You can use the debugger in the App Designer.*

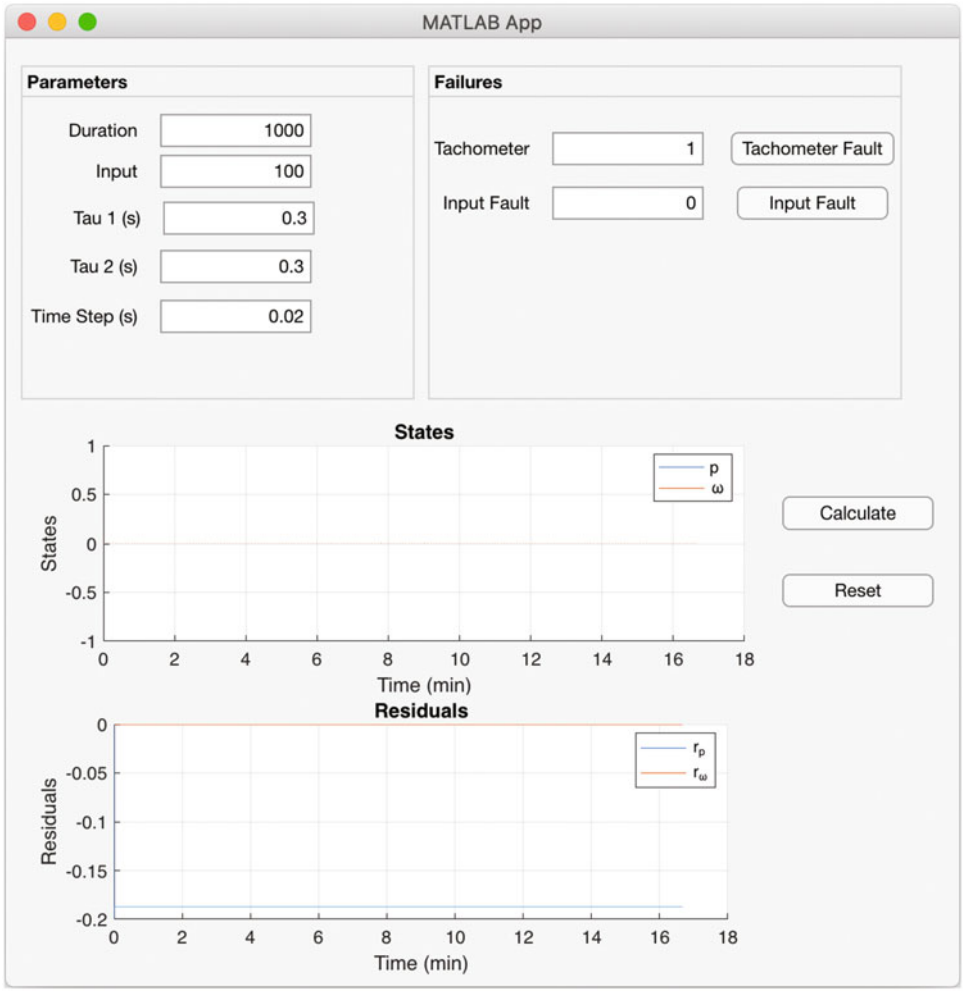

Figure 10.12: *The app after a run.*

Summary

This chapter has demonstrated how to design a detection filter for detecting faults in a dynamical system. The system is demonstrated with an air turbine that can experience a pressure regulator failure and a tachometer failure. In addition, we used App Designer to design a GUI to automate filter simulations. The GUI demonstrates real-time plotting and injecting failures into an ongoing simulation loop. Table 10.1 lists the code developed in the chapter.

Table 10.1: *Chapter Code Listing*

File	Description
`RHSAirTurbine`	Air turbine dynamical model in continuous state space form
`DetectionFilter`	Builds and updates a linear detection filter
`DetectionFilterSim`	Simulation of a detection filter
`DetectionFilterGUI`	Run the detection filter simulation from a GUI
DFGUI.m1App	App Designer app
DFGUI.mlappinstall	DFGUI app installer