CHAPTER 12

■ ■ ■

Orbit Determination

12.1 Introduction

Determining orbits from measurements has been done for hundreds of years. The general approach is to take a series of measurements of the object from the ground. This is a set of angles at different times. Given the location on the Earth, and this set of data, one can reconstruct the orbit. Ideal orbits, which make the assumption that the Earth's gravity is a point at the center of the Earth, are conic sections. Those that stay near the Earth are ellipses. These can be defined as a set of orbital elements. In this chapter, we will design a neural network to find the values for two of the elements. Our model will be simpler than that which astronomers must use. We will assume that all of our orbits are in the Earth's equatorial plane and that the observer is at the center of the Earth.

The purpose of this chapter is to show that a neural net can do orbit determination. For comparison with traditional methods, see the classic textbook from 1965 by Escobal [11].

12.2 Generating the Orbits

12.2.1 Problem

We want to create a set of orbits for testing and training a neural net.

12.2.2 Solution

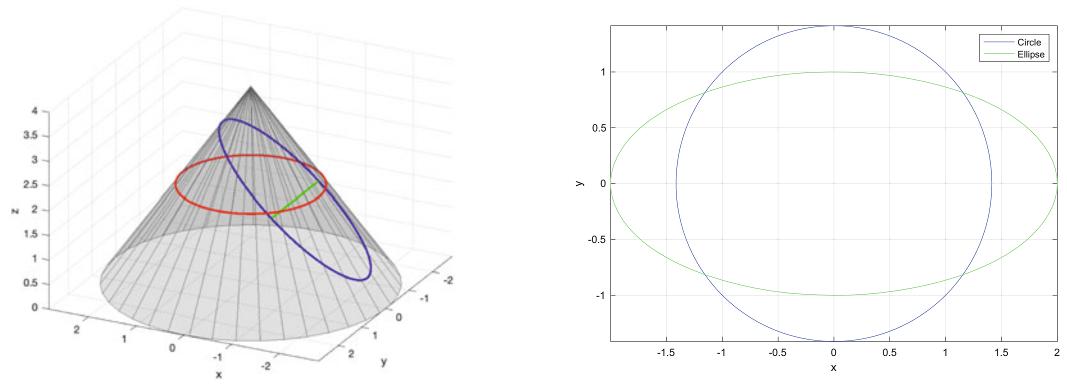
Implement a random orbit generator using Keplerian propagation of elements.

12.2.3 How It Works

An orbit involves at least two bodies, for example, a planet and a spacecraft. In the ideal two-body case, the two bodies rotate about the common center of mass, known as the barycenter. For all practical spacecraft cases, the spacecraft mass itself is negligible, and this means that the satellite follows a conic section path about the primary body's center of mass. A conic section is a curve that fits on a cone, as shown in Figure 12.1. Two conics, a circle and ellipse, are drawn. Hyperbolas and parabolas are also conic sections, but we will only look at elliptical orbits in this chapter.

M. Paluszek and S. Thomas, *Practical MATLAB Deep Learning*,
https://doi.org/10.1007/978-1-4842-5124-9_12

Figure 12.1: Ellipse and circle on a cone and viewed along their normal.



The code that draws this picture is in the following script. It calls two functions, `Cone` and `ConicSectionEllipse`. `r0` and `h` are only needed to draw the cone. The algorithm only cares about `theta`, the cone half angle.

ConicSection.m

```
theta = pi/4;
h     = 4;
r0    = h*sin(theta);

ang   = linspace(0,2*pi);
a     = 2;
b     = 1;
cA    = cos(ang);
sA    = sin(ang);
n     = length(cA);
c     = 0.5*h*sin(theta)*[cA;sA;ones(1,n)];
e     = [a*cA;b*sA;zeros(1,n)];

% Show a planar representation
NewFigure('Orbits');
plot(c(1,:),c(2,:),'b')
hold on
plot(e(1,:),e(2,:),'g')
grid
xlabel('x')
ylabel('y')
axis image
legend('Circle','Ellipse');

 [z,phi,x]    = ConicSectionEllipse(a,b,theta);
ang           = pi/2 + phi;
e             = [cos(ang) 0 sin(ang);0 1 0; -sin(ang) 0 cos(ang)]*e;
e(1,:)        = e(1,:) + x;
e(3,:)        = e(3,:) + h - z;

Cone(r0,h,40);
```

```
hold on
plot3(c(1,:),c(2,:),2*ones(1,n),'r','linewidth',2);
plot3(e(1,:),e(2,:),e(3,:),'b','linewidth',2);
line([x x],[-b b],[h-z h-z],'color','g','linewidth',2);
view([0 1 0])
```

The view is set to look along the y-axis which is the axis of rotation for the ellipse. The function `Cone` draws the cone. `line` draws the axis of rotation that is along the short axis.

The solution that is used to draw the conic sections is derived in the last section of this chapter. The orbit may be elliptical, with an eccentricity less than 1, parabolic with an eccentricity equal to 1, or hyperbolic with an eccentricity greater than 1. Figure 12.2 shows the geometry of an elliptical orbit. This is a planar orbit in which the orbital motion is two-dimensional. The semi-major axis a is

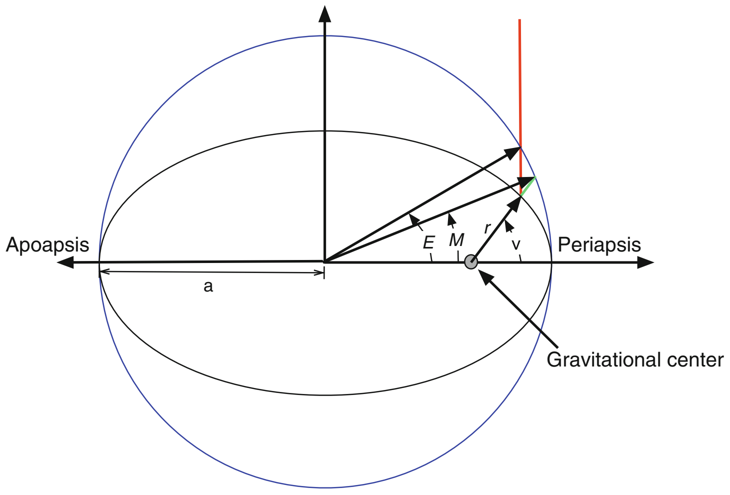
$$a = \frac{r_a + r_p}{2} \tag{12.1}$$

where r_a is the apoapsis (apogee for the Earth) radius, or point furthest from the central planet, and r_p is the periapsis radius (perigee for the Earth), or closest point to the planet. The eccentricity, e, of the orbit is

$$e = \frac{r_a - r_p}{r_a + r_p} \tag{12.2}$$

When $r_a = r_p$, the orbit is circular and $e = 0$. This formula is not meaningful for parabolic or hyperbolic orbits. Figure 12.2 shows three angular measurements, M mean anomaly, E eccentric anomaly, and ν true anomaly. All are measured from periapsis. The mean anomaly is

Figure 12.2: Elliptical orbit.



related to the mean orbit rate n through a simple function of time.

$$M = M_0 + n(t - t_0) \tag{12.3}$$

The eccentric anomaly is the angle to the current position as projected onto the ellipse's circumscribing circle, drawn in blue. It is related to the mean anomaly by Kepler's equation.

$$M = E - e \sin E \tag{12.4}$$

This equation needs to be solved numerically in general, but for small values of e, $e < 0.1$, this approximation can be used.

$$E \approx M + e \sin M + \frac{1}{2} e^2 \sin 2M \tag{12.5}$$

This is because apoapsis is not well defined for very small e. Higher-order formulas can also be found. The true anomaly is related to the eccentric anomaly through the equation

$$\tan \frac{\nu}{2} = \sqrt{\frac{1+e}{1-e}} \tan \frac{E}{2} \tag{12.6}$$

Finally, the orbit radius is

$$r = \frac{a(1-e)(1+e)}{1 + e \cos \nu} \tag{12.7}$$

If $e > 1$ in this equation, r will go to ∞, as is expected for parabolic or hyperbolic orbits.

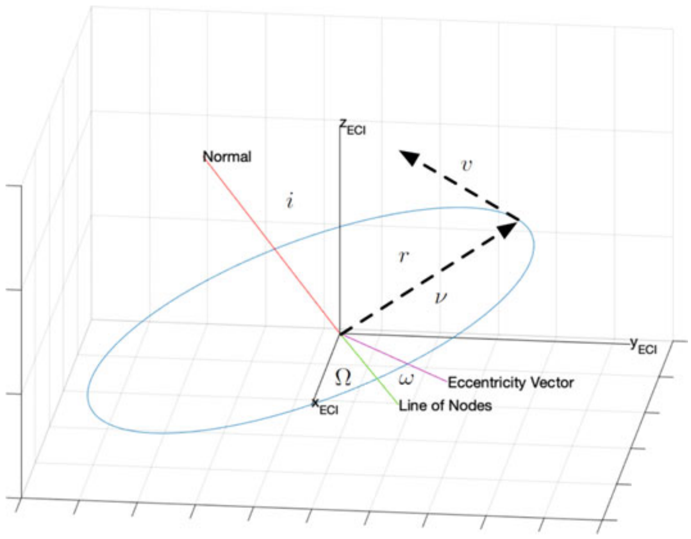
Seven parameters are necessary to define an orbit of a spacecraft about a spherically symmetric body. One is the gravitational parameter, generally denoted by the symbol μ. The gravitational parameter is

$$\mu = G(m_1 + m_2) \tag{12.8}$$

where m_1 is the mass central body and m_2 is the mass of the orbiting body. G is the gravitational constant with units of m^3/kg/s^2. For the Earth, $G = 6.6774 \times 10^{-11}$. μ for the Earth is 3.98600436 $\times 10^8$ m^3/s^2. There are many ways of representing the other six elements. The two most popular sets are position and velocity (r and v) vectors, and Keplerian orbital elements. Each representation uses six independent variables to describe the orbit, plus μ. Both are shown in Figure 12.3.

The Keplerian elements are defined as follows. Two elements define the elliptical orbit. The size of the orbit is determined by the semi-major axis a, which is the average of the perigee radius and apogee radius. The size and shape of the orbit are defined by the eccentricity, e. Two elements define the orbital plane. Ω is the longitude which is the right ascension of the ascending node, or the angle from the $+X$ axis of the reference frame to the line where the orbit plane intersects the xy-plane. i is the inclination and is the angle between the xy-plane and the orbit plane. ω is the argument of perigee and is the angle in the orbit plane between the ascending node line and perigee (where the orbit is closest to the center of the central body). ν is the true anomaly and is the angle between perigee and the spacecraft. The mean anomaly M may be used in the element set instead of ν. M or ν tells us where the spacecraft is in its orbit.

Figure 12.3: Orbital elements. The underlying plot was drawn using `DrawEllipticOrbit`.



To summarize, the Keplerian elements are

$$x = \begin{bmatrix} a \\ i \\ \Omega \\ \omega \\ e \\ M \end{bmatrix} \tag{12.9}$$

The orbit period, with units of seconds, is

$$P = 2\pi\sqrt{\frac{a^3}{\mu}} \tag{12.10}$$

The orbit parameter, with units of distance (conventionally km), is

$$p = a(1-e)(1+e) \tag{12.11}$$

The in-plane position and velocity are

$$r_p = \frac{p}{1+e\cos\nu}\begin{bmatrix} \cos\nu \\ \sin\nu \\ 0 \end{bmatrix} \tag{12.12}$$

$$v_p = \sqrt{\frac{\mu}{p}}\begin{bmatrix} -\sin\nu \\ e+\cos\nu \\ 0 \end{bmatrix} \tag{12.13}$$

The transformation matrix from planar to three-dimensional coordinates is

$$c = \left[\begin{array}{rrr} \cos\Omega\cos\omega - \sin\Omega\sin\omega\cos i & -\cos\Omega\sin\omega - \sin\Omega\cos\omega\cos i & \sin\Omega\sin i \\ \sin\Omega\cos\omega + \cos\Omega\sin\omega\cos i & -\sin\Omega\sin\omega + \cos\Omega\cos\omega\cos i & -\cos\Omega\sin i \\ \sin\omega\sin i & \cos\omega\sin i & \cos i \end{array}\right] \tag{12.14}$$

That is

$$r = cr_p \tag{12.15}$$

$$v = cv_p \tag{12.16}$$

For the purposes of creating the neural net, we will look at orbits with the inclination, $i = 0$, and the ascending node, $\Omega = 0$. The transformation matrix reduces to a rotation about z.

$$c = \left[\begin{array}{rrr} \cos\omega & -\sin\omega & 0 \\ \sin\omega & \cos\omega & 0 \\ 0 & 0 & 1 \end{array}\right] \tag{12.17}$$

We now want to propagate the orbit forward in time. There are two alternative approaches for doing so. One approach is to use Keplerian propagation, where we keep five of the elements constant, and simply march the mean anomaly forward in time at a constant rate of $n = \sqrt{\mu/a^3}$. At each point in time, we can convert the set of six orbital elements into a new position and velocity. This approach is limited though, in that it assumes the orbit follows a Keplerian orbit (the only external foci is the gravitation of a central body with uniform mass distribution). The second approach, which gives us more flexibility, for external forces like thrust and drag is to numerically integrate the dynamic equations of motion. The state equations for orbit propagation are

$$\dot{v} = -\mu\frac{r}{|r|^3} + a \tag{12.18}$$

$$\dot{r} = v \tag{12.19}$$

The terms on the right-hand side of the velocity derivative equation are the point mass gravity acceleration with additional acceleration a. This is implemented in `RHSOrbit`.

```
function xDot = RHSOrbit(~,x,d)

r     = x(1:2);
v     = x(3:4);
xDot  = [v;-d.mu*r/(r'*r)^1.5 + d.a];
```

We will create a script that simulates multiple orbits. The simulation will use `RHSOrbit`. The first part of the orbit generation script sets up the random orbital elements.

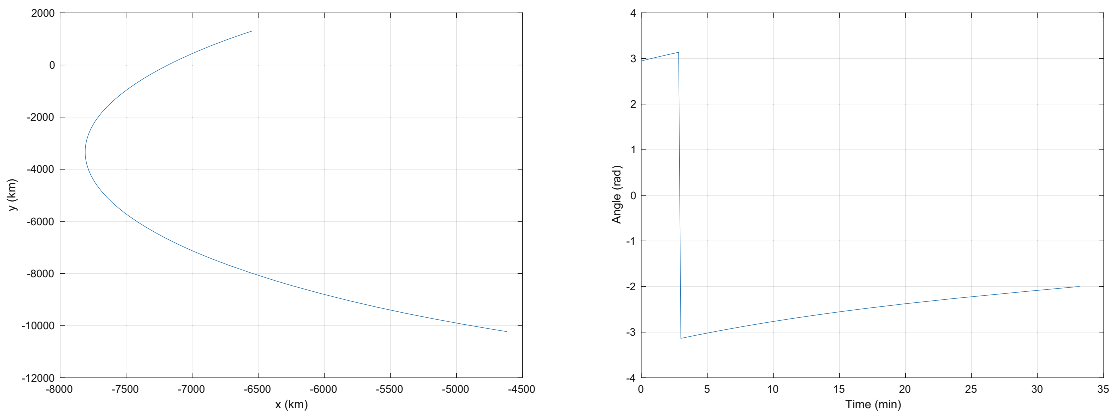
Orbits.m

```
%% Generate Orbits for angles-only element estimation
% Saves a mat-file called OrbitData.
%% See also
% El2RV, RungeKutta, RHSOrbit, TimeLabel, PlotSet

nEl    = 500;           % Number of sets of data
d      = struct;        % Initialize
d.mu   = 3.98600436e5; % Gravitational parameter, km^3/s^2
d.a    = [0;0];         % Perturbing acceleration

% Random elements
e      = 0.6*rand(1,nEl);           % Eccentricity
a      = 8000 + 1000*randn(1,nEl); % Semi-major axis
M      = 0.25*pi*rand(1,nEl);       % Mean anomaly
```

The next section runs the simulations and saves the angles. Each simulation has 2000 steps, and each step is 2 seconds. We are only using one in ten points for the orbit determination. We save the orbital elements for testing the neural network. We are not applying any external acceleration. We could have used Kepler propagation, but by simulating the orbit, we have the option for studying how well the neural network performs with disturbances.

```
% Set up the simulation
nSim   = 2000; % Number of simulation steps
dT     = 2; % Time step

% Only use some of the sim steps
jUse   = 1:10:nSim;

% Data for Deep Learning
data   = cell(nEl,1);

%% Simulate each of the orbits
x      = zeros(4,nSim);
t      = (0:(nSim-1))*dT;
el(nEl) = struct('a',7000,'e',0); % initialize struct array

for k = 1:nEl
  [r,v] = El2RV([a(k) 0 0 0 e(k) M(k)]);
  x     = [r(1:2);v(1:2)];
  xP    = zeros(4,nSim);
  for j = 1:nSim
    xP(:,j) = x;
    x       = RungeKutta( @RHSOrbit, 0, x, dT, d );
  end
  data{k}      = atan2(xP(2,jUse),xP(1,jUse));
  el(k).a      = a(k);
  el(k).e      = e(k);
end
```

Figure 12.4: The last test orbit. The measured angle is on the right. These are only showing the data used in orbit determination.



The final part plots the orbits and saves the data to a file.

```
%% Save for the Deep Learning algorithm
save('OrbitData','data','el');
```

The last orbit is shown in Figure 12.4. The jump in angle is due to angles being defined from $-\pi$ to $+\pi$. We could have used `unwrap` to get rid of this jump. We are only measuring for part of an orbit. We can set up the simulation to measure any part of an orbit, or even multiple orbits.

12.3 Training and Testing

12.3.1 Problem

We want to build a deep learning system to compute the eccentricity and semi-major axis for an orbit from angle measurements.

12.3.2 Solution

The orbit history is a time series of angles. We will take angles at uniform time intervals. We will use `fitnet` to fit the data.

12.3.3 How It Works

We load in the data from the mat file and separate it into training and testing sets.

OrbitNeuralNet.m

```
%% Train and test the Orbit Neural Net
%% See also:
% Orbits, fitnet, configure, train, sim, cascadeforwardnet,
    feedforwardnet

s       = load('OrbitData');
n       = length(s.data);
nTrain  = floor(0.9*n);

%% Set up the training and test sets
kTrain  = randperm(n,nTrain);
sTrain  = s.data(kTrain);
nSamp   = size(sTrain{1},2);
xTrain  = zeros(nSamp,nTrain);
aMean   = mean([s.el(:).a]);

for k = 1:nTrain
  xTrain(:,k) = sTrain{k}(1,:);
end

elTrain     = s.el(kTrain);
yTrain      = [elTrain.a;elTrain.e];
yTrain(1,:) = yTrain(1,:)/aMean;
% Normalize the data so it is the same magnetic as the eccentricity
kTest       = setdiff(1:n,kTrain);
sTest       = s.data(kTest);
nTest       = n-nTrain;
xTest       = zeros(nSamp,nTest);
for k = 1:nTest
  xTest(:,k) = sTest{k}(1,:);
end

elTest      = s.el(kTest);
yTest       = [elTest.a;elTest.e];
yTest(1,:)   = yTest(1,:)/aMean;
```

The neural network will use sequences of angles and their related times as the input. The output will be the two orbital elements: semi-major axis and eccentricity. In general, if we know the position and velocity at a point in the orbit, we can always compute the orbital elements. This is done in the function `El2RV`. Although we don't directly measure velocity, it can be estimated by differencing position measurements. With angle-only measurements, we don't have a measure of range. The question is, can the neural network infer the range from the time variation of the angles?

We train the network using `fitnet`. Note that we normalized the semi-major axis so that the magnitude is the same order as the eccentricity. This improves the fitting.

```
%% Train the network
net        = fitnet(10);

net        = configure(net, xTrain, yTrain);
net.name   = 'Orbit';
net        = train(net,xTrain,yTrain);
```

We use the test data to test the network.

```
%% Test the network
yPred        = sim(net,xTest);
yPred(1,:)   = yPred(1,:)*aMean;
yTest(1,:)   = yTest(1,:)*aMean;
yM           = mean(yPred-yTest,2);
yTM          = mean(yTest,2);
fprintf('\nFit Net\n');
fprintf('Mean semi-major axis error %12.4f (km) %12.2f %%\n',yM(1),100*
    abs(yM(1))/yTM(1));
fprintf('Mean eccentricity    error %12.4f      %12.2f %%\n',yM(2),100*
    abs(yM(2))/yTM(2));
%% Plot the results
yL  = {'a' 'e'};
yLeg = {'Predicted','True'};
PlotSet(1:nTest,[yPred;yTest],'x label','Test','y label',yL,...
'figure title','Predictions using Fitnet','plot set',{[1 3] [2 4]},...
'legend',{yLeg yLeg});
```

The results are best for `fitnet`. However, the results will vary with each run.

```
>> OrbitNeuralNet
>> OrbitNeuralNet

Fit Net
Mean semi-major axis error      31.9872 (km)          0.41 %
Mean eccentricity    error       0.0067               2.48 %

Cascade Forward Net
Mean semi-major axis error     -89.8603 (km)          1.15 %
Mean eccentricity    error      -0.0100               3.74 %

Feed Forward Net
Mean semi-major axis error      40.2986 (km)          0.52 %
Mean eccentricity    error       0.0001               0.03 %
```

Figures 12.5, 12.6, and 12.7 show the test results. Both semi-major axis and eccentricity results are reasonably good. You can experiment with different spans of data and different sampling intervals. The code is in the script `OrbitNeuralNet.m`.

Figure 12.5: Test results using `fitnet`.

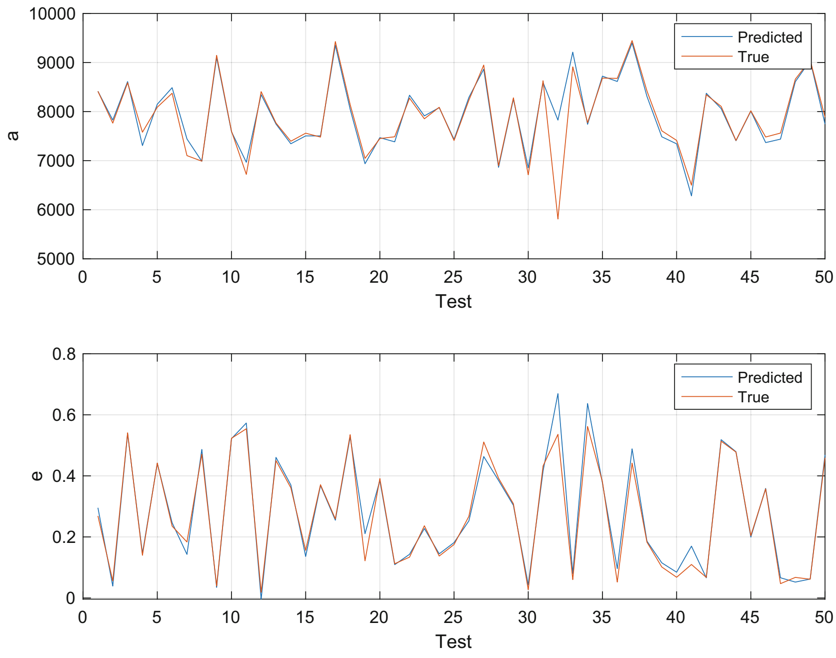


Figure 12.6: Test results using `cascadeforwardnet`.

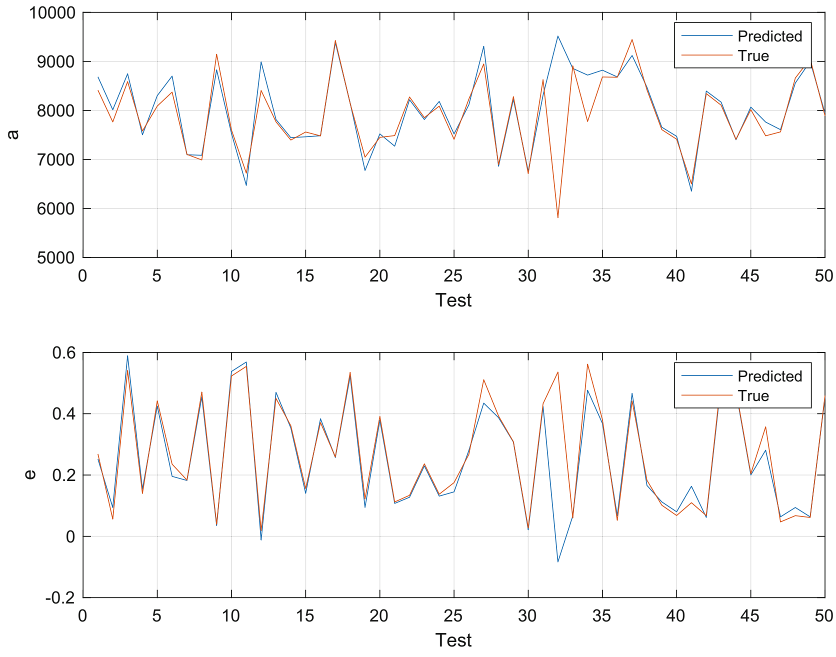
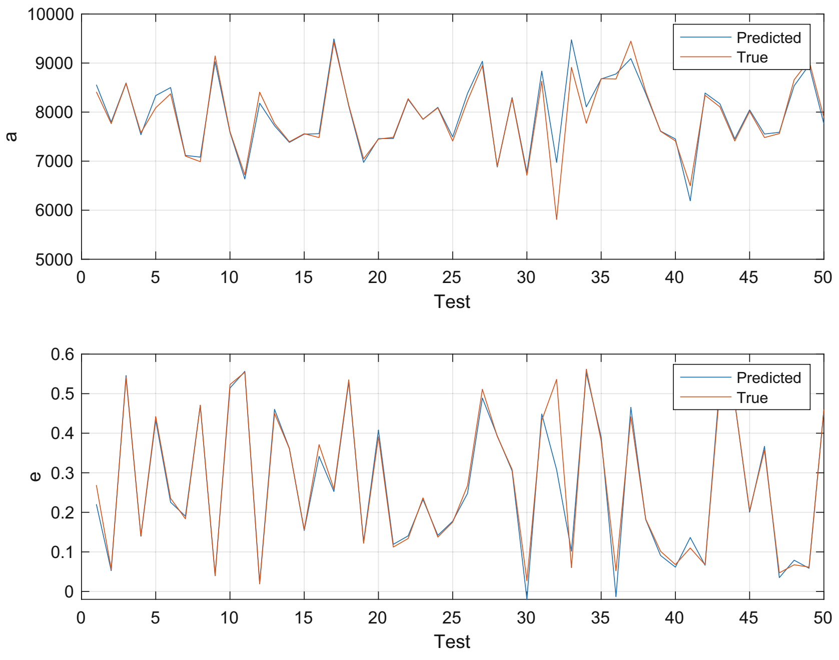


Figure 12.7: Test results using `feedforwardnet`.



We train the network using `cascadeforwardnet`. The code doesn't change except for the function name.

```
%% Train the cascade forward network
net          = cascadeforwardnet(10);

net          = configure(net, xTrain, yTrain);
net.name     = 'Orbit';
net          = train(net,xTrain,yTrain);
```

We finally train it using `feedforwardnet`.

```
%% Train the feed forward network
net          = feedforwardnet(10);

net          = configure(net, xTrain, yTrain);
net.name     = 'Orbit';
net          = train(net,xTrain,yTrain);
```

12.4 Implementing an LSTM

12.4.1 Problem

We want to build a long short-term memory neural net (LSTM) to estimate the orbital elements. LSTMs have been demonstrated in previous chapters. They are an alternative to the functions shown earlier.

12.4.2 Solution

The orbit history is a time series of angles. We will use a bidirectional LSTM to fit the data. We will take angles at uniform time intervals.

12.4.3 How It Works

We load in the data from the mat file and separate it into training and testing sets. The data format is different from the feedforward networks. `xTrain` is a cell array, but `yTrain` is a matrix with a row for each cell array in `xTrain`.

OrbitLSTM.m

```
%% Script to train and test the Orbit LSTM
% It will estimate the orbit semi-major axis and eccentricity from a time
% sequence of angle measurements.
%% See also
% Orbits, sequenceInputLayer, bilstmLayer, dropoutLayer,
    fullyConnectedLayer,
% regressionLayer, trainingOptions, trainNetwork, predict

s            = load('OrbitData');
n            = length(s.data);
nTrain       = floor(0.9*n);

%% Set up the training and test sets
kTrain       = randperm(n,nTrain);
aMean        = mean([s.el(:).a]);
xTrain       = s.data(kTrain);
nTest        = n-nTrain;

elTrain      = s.el(kTrain);
yTrain       = [elTrain.a;elTrain.e]';
yTrain(:,1)  = yTrain(:,1)/aMean;
kTest        = setdiff(1:n,kTrain);
xTest        = s.data(kTest);

elTest       = s.el(kTest);
yTest        = [elTest.a;elTest.e]';
yTest(:,1)       = yTest(:,1)/aMean;
```

We train the network using `trainNetwork`.

```
%% Train the network with validation
numFeatures       = 1;
numHiddenUnits1   = 100;
numHiddenUnits2   = 100;
numClasses        = 2;

layers = [ ...
    sequenceInputLayer(numFeatures)
    bilstmLayer(numHiddenUnits1,'OutputMode','sequence')
    dropoutLayer(0.2)
    bilstmLayer(numHiddenUnits2,'OutputMode','last')
    fullyConnectedLayer(numClasses)
    regressionLayer]

maxEpochs = 20;

options = trainingOptions('adam', ...
    'ExecutionEnvironment','cpu', ...
    'GradientThreshold',1, ...
    'MaxEpochs',maxEpochs, ...
    'Shuffle','every-epoch', ...
    'ValidationData',{xTest,yTest}, ...
    'ValidationFrequency',5, ...
    'Verbose',0, ...
    'Plots','training-progress');

net = trainNetwork(xTrain,yTrain,layers,options);
```

`options` is given validation data. Note the cell array that is required for the validation data.

```
     'ValidationData',{xTest,yTest}, ...
```

We shuffle the data. This generally improves the results since the learning algorithm sees the data in a different order on each epoch. We use the test data to test the network. `predict` produces results based on the test data. This is the same data used for validation during learning.

The results are given as follows.

```
>> OrbitLSTM
layers =
  6x1 Layer array with layers:

     1   ''   Sequence Input      Sequence input with 1 dimensions
     2   ''   BiLSTM              BiLSTM with 100 hidden units
     3   ''   Dropout             20% dropout
     4   ''   BiLSTM              BiLSTM with 100 hidden units
     5   ''   Fully Connected     2 fully connected layer
     6   ''   Regression Output   mean-squared-error

biLSTM
Mean semi-major axis error     -63.4780 (km)
Mean eccentricity    error       0.0024
```

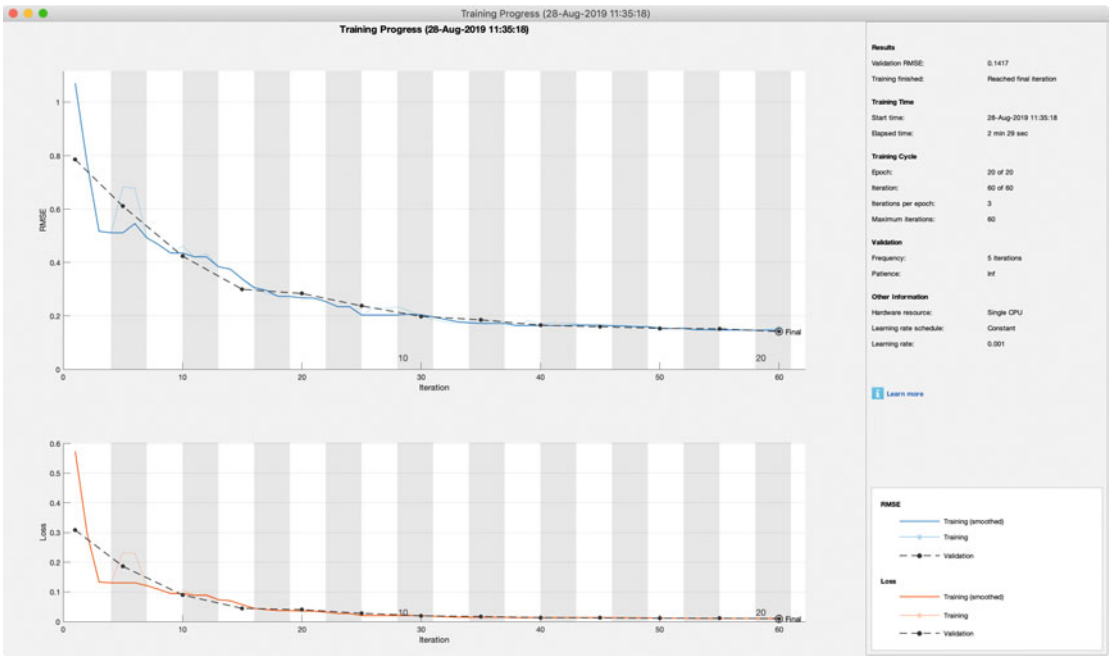
We use two BiLSTM layers with a 20% dropout between layers. Dropout removes neurons and helps prevent overfitting. Overfitting is when the results correspond too closely toward a particular set of data. This makes it hard for the trained network to identify patterns in new data. The first BiLSTM layer produces a sequence as its output. The second BiLSTM layer's `'OutputMode'` is set to `'last'`. The `numClasses` is 2 because we are estimating two parameters. The fully connected layer connects the two BiLSTM outputs to the two parameters we want to identify in the regression layer. The training window is shown in Figure 12.8. We could have continued the training for more epochs as the root-mean-square error (RMSE) is still improving.

This particular set of layers is to show you how to build a neural network. It is by no means the ‘‘best” architecture for this problem. We did try a single LSTM layer and a single BiLSTM layer worked better.

Figure 12.9 shows the test results. The results are not quite as good as the feedforward nets given earlier. We've only used two layers. From Chapter 11, you see that ‘‘professional” networks can have dozens if not hundreds of layers. The difference is due to the smaller number of neurons in the LSTM. You can experiment with this network to improve the results.

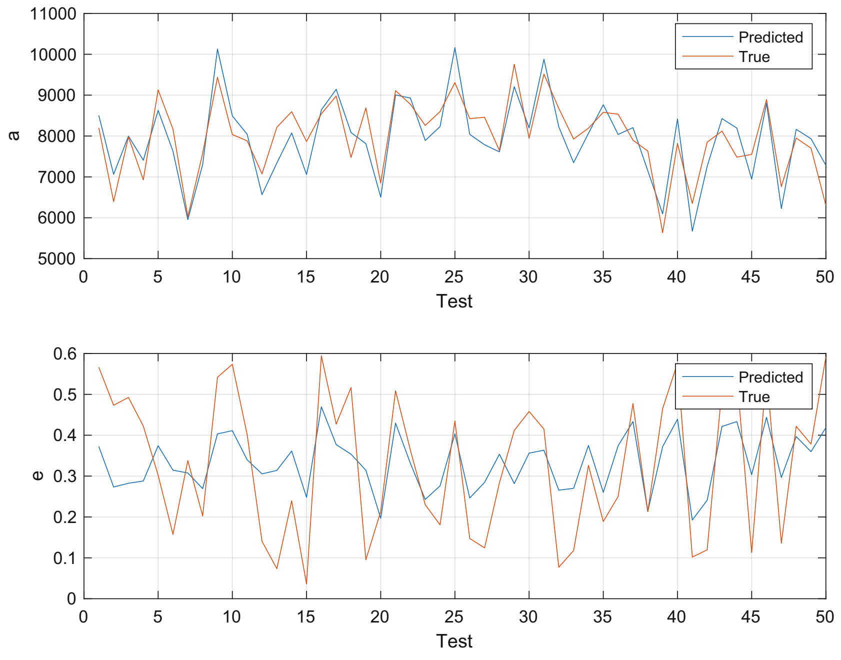
In this chapter, we have compared two approaches, in MATLAB, to solving the orbit determination problem. Using the MATLAB functions worked a bit better than the LSTM we implemented. We made the argument of perigee constant to make the problem easier. The next step would be to try and find the full set of orbital elements and then try to design a system that works from a fixed point on the Earth. In the latter case, we would need to account for the rotation of the Earth. Another improvement would be to take the measurements at different time steps. For an elliptical orbit, taking many measurements at perigee is more productive than at apogee because the spacecraft is moving faster. One could write a preprocessor to select inputs to our neural network based on the angular change with respect to time. Orbit determination systems, using algorithmic approaches, can also compute errors in the observer's location. You could also try other measurements, such as range and range rate. These measurements are used for deep space and geosynchronous spacecraft.

Figure 12.8: Training window.



Results	
Validation RMSE:	0.1417
Training finished:	Reached final iteration
Training Time	
Start time:	28-Aug-2019 11:35:18
Elapsed time:	2 min 29 sec
Training Cycle	
Epoch:	20 of 20
Iteration:	60 of 60
Iterations per epoch:	3
Maximum iterations:	60
Validation	
Frequency:	5 iterations
Patience:	Inf
Other Information	
Hardware resource:	Single CPU
Learning rate schedule:	Constant
Learning rate:	0.001

Figure 12.9: Test results using the bidirectional LSTM.



12.5 Conic Sections

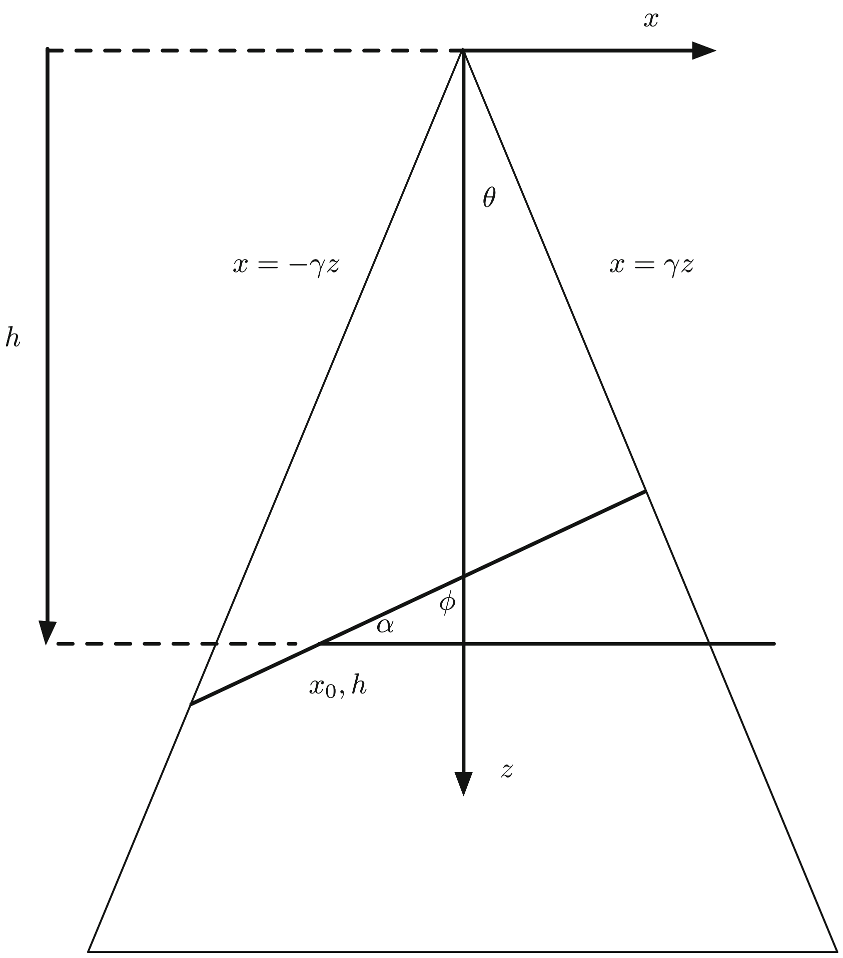
For a given ellipse and cone, we need to solve for the location of the center of the plane that cuts the cone and its angle. The problem can be solved by working in the zy-plane. This is shown in Figure 12.10. The equation for an ellipse is

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1 \tag{12.20}$$

The equation for a right circular cone is

$$x^2 + y^2 = \gamma^2 z^2 \tag{12.21}$$

Figure 12.10: Conic section.



where $\gamma = \cos\theta$ and θ is the cone half angle. This is just saying that the radius of the cone at position z is γz. The equation for a plane is

$$z = h \tag{12.22}$$

That is, z is constant for all x and y. If we rotate this about the y-axis by α and translate it by x_0, we get the equation for the cone. The ellipse comes from the intersection of the plane with the cone. b is along the y-axis. The angle between the plane that cuts the cone and the xy-plane is α and is a function of a, b, and the cone half angle θ. The following equations are for $\theta = \pi/4$.

$$\tan^2\alpha = 1 - \frac{b^2}{a^2} \tag{12.23}$$

Noting that

$$\phi = \frac{\pi}{2} - \alpha \tag{12.24}$$

The relationship between the plane and the vertical for a cone with a half angle of $\frac{\pi}{4}$ is then

$$\phi = \frac{\pi}{2} - \text{atan}\sqrt{1 - \frac{b^2}{a^2}} \tag{12.25}$$

The cone can be viewed in the plane. On the right side, the equation is

$$x = \gamma x \tag{12.26}$$

where $\gamma = \cos\theta$, where θ is the cone half angle. On the left side

$$x = -\gamma x \tag{12.27}$$

We then write the equations for the line along the major axis of the ellipse on each side of the triangle. On the right

$$x = x_0 + a\cos\alpha \tag{12.28}$$
$$z = h - a\sin\alpha \tag{12.29}$$

where $\alpha = \pi/2 - \phi$. On the left

$$x = x_0 - a\cos\alpha \tag{12.30}$$
$$z = h + a\sin\alpha \tag{12.31}$$

Substituting into the equations for the cone, we get

$$\begin{bmatrix} 1 & -\gamma \\ 1 & \gamma \end{bmatrix} \begin{bmatrix} x_0 \\ h \end{bmatrix} = a \begin{bmatrix} -\gamma\sin\alpha - \cos\alpha \\ \cos\alpha - \gamma\sin\alpha \end{bmatrix} \tag{12.32}$$

The code that solves the equations follows. We could have solved the inverse analytically since it is 2 by 2.

ConicSectionEllipse.m

```
function [h,phi,x] = ConicSectionEllipse(a,b,theta)

if( nargin < 1 )
  [h, phi, y] = ConicSectionEllipse(2,1,pi/4);
  fprintf('h   = %12.4f\n',h);
  fprintf('phi = %12.4f (rad)\n',phi);
  fprintf('x   = %12.4f\n',y);
  clear h
  return
end

phi   = pi/2 - atan(sqrt(1-b^2/a^2));

alpha = pi/2 - phi;
c     = cos(alpha);
s     = sin(alpha);
gamma = cos(theta);
f     = a*[-gamma*s - c;c - gamma*s];
q     = [1 -gamma;1 gamma]\f;
x     = q(1);
h     = q(2);
```