MATLAB/Octave Modeling of PV Tracking System

Jonathan West

January 21, 2014

1 Introduction

MATLAB is a commercial software package that is used for mathematical computations. Octave is an open source, free program similar to, and mostly compatible with MatLAB. Although Octave lacks some significant features of MatLAB, its free price tag makes it an attractive alternative/teaching tool for MatLAB. Although your school and possibly your company may have MATLAB available, its high price tag makes it a bit out of reach for small businesses or individuals, so we will write programs that will run on both MATLAB and Octave. I have found that Octave does almost everything I have ever needed, but since MATLAB is the accepted academic and industrial standard we will also call this a MATLAB course, with the understanding that almost everything can be run just as well on Octave.

There are already a number of good MATLAB tutorials and resources on the web so search for MATLAB tutorial and you will have more reading material than you need. For this class I am going to skip all the theory and move straight into an application.

1.1 Octave Functions

Octave can be used at the command line to do immediate calculations, which is handy for quick analysis but clumsy for repeated work since you must type everything in every time you want to do it. You can save your commands in a script file, which simply runs whatever is in the file as if you typed it in at the command line, but the real power of Octave and any other programming language is in fuctions, so we will start there.

A function is a set of instructions that perform a task and modify or return a value. Best programming practice is to have a function do one operation and do it well.

In Octave a function can be defined on the command line, but more often it is defined in a seperate file so that it can be reused and distributed to others. An Octave function file is named the same as the function itself with a .m extension. So a function called $myFunction$ would be saved in a file named $myFunction.m$.

1.2 Problem

For our project we want to model the sun's movement and create a mathematical model for a PV tracking system that will calculate the sun's movement and radiation and also simulate a tracking system.

1.3 Our First Function: Solar Radiation

The first step in our model will be to model and simulate the movement of the sun. To calculate how much power our PV panels will produce, we need to know:

1. The intensity of the solar radiation (also known as irradiance ($E_e = W/m^2$)
2. The direction from which the radiation is coming. This is measured in azimuth (θ), the horizontal angle from north, and elevation (ρ), or the angle up from horizontal.

So we will write a function that we will call $solarIrradiance$ that will calculate these values. These will be the outputs of our function and so our function will look something like $(E_e, \theta, \rho) = solarIrradiance()$ which means our function will return (or produce) those values when it is called.

So our next question is what information do we need to know in order to calculate those values? The position and intensity of the sun at a particular location is influenced by the rotation of the earth, the tilt of the earth's axis relative to the sun, the orbit of the earth around the sun and your location on the earth and the transmission properties of the atmosphere. So these parameters will be the inputs to our model.

1. The angle of the earth's axis with respect to the sun (ϕ)
2. The position of the earth in its orbit ($dayOfYear$)
3. The rotation angle of the earth on its axis ($timeOfDay$)
4. Your position on the earth ($long, lat, elevation$)

So theses will be the inputs to our function and now our function looks like $(E_e, \theta, \rho) = solarIrradiance(\phi, day, time, long, lat, elevation)$

So let's analyze what we will need to calculate:

1. Using the day of the year, calculate the angle of the sun with respect to the equator.
2. Using the time of day, calculate the position of the sun in the sky
3. Using the sun's position and your elevation, calculate the effective thickness of the atmosphere
4. Using the atmosphere thickness and the irradiance in space, calculate the irradiance at your location.

As is often the case, our function looks fairly complicated so we will probably want ot break it down into smaller functions. This is a good general programming approach because it allows you to solve large, complex problems by solving several smaller, easier ones. In addition to being smaller and therefore easier to write, small functions are also easier to test so that once you have written and tested it, you can use it with confidence in future calculations and projects.

So let's start with the first function: calculate the angle of the sun from the day of the year. This will give us a function like: $(\phi) = solarLatitude(dayOfYear)$ which means if we give the function the current day of the year, it will return the angle of the sun from the equator. So how will we calculate that?

We know that the earth's axis is tilted 23.4° from the plane of its orbit and that the days at which the sun is directly over teh equator are the spring and fall equinoxes which occur on March 20 and Sep 22. The solstices are the peaks of the curve when the sun is farthest north and south and these fall on June 21 (172) and December 21 (355). We need to decide how to tell Octave what day it is so let's use a simple system of naming the day by it number 1-365 with 1 being Jan 1 and 365 being Dec 31. This puts the equinoxes at days 79 and 265. Since the motion is roughly circular (if we wanted to get carried away we could model the fact the orbit is actually elliptical, but that is not likely to be significant) we can assume the shape of the curve will be sinusoidal so we start with the sine wave shown in Figure 1. We know that our amplitude should be 23.4°, that it completes one cycle every 365 days and that it crosses zero at 79 and 265 so we can write the sine function as:

$$23.4 * sin(\frac{day - 79}{365} * 360) \tag{1}$$

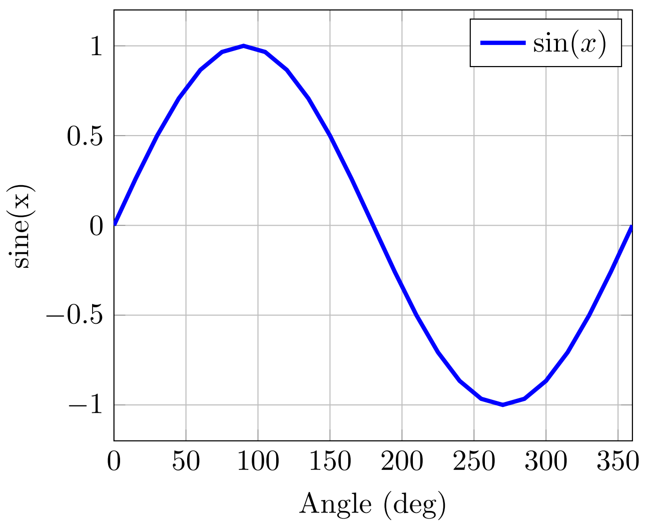

Figure 1: Sine Wave

This is plotted in Figure 2. So now we know the formula we want to use and we need to get Octave to calculate it for us. Here is what the file *solarLatitude.m* looks like:

```
function lat = solarLatitude(day)
    % solarLatitude Calculates the solar latitude angle
```

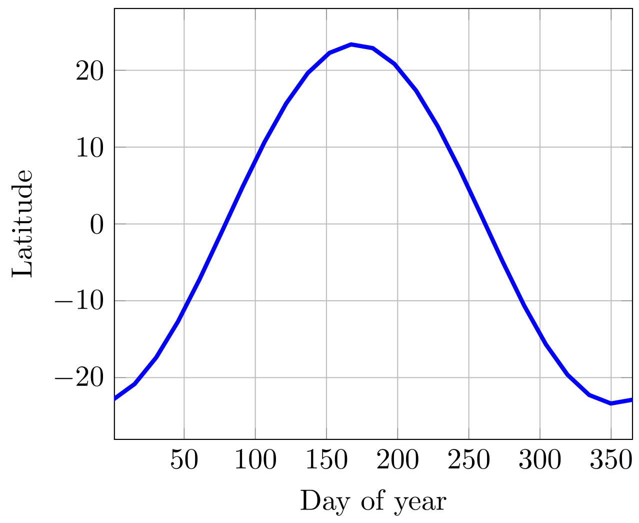

Figure 2: Solar Latitude

```
 3  %   based on the day of the year.
 4  %
 5  %   day is the day of the year (1-365)
 6  %   lat is the solar latitude angle in degrees
 7
 8  % I put these values into variables just to make the formula
 9  % easier to read, but they could go directly in the formula.
10   tilt = 23.4;
11   equinox = 79;
12
13  % note that in Octave, sin() is in radians, thus
14  % the 2*pi instead of 360
15   lat = tilt * sin( (day-equinox)/365 * 2 * pi);
16 end
```

Line 1 defines the function. *lat* is the returned value, *solarLatitude* is the name of the function and *day* is the value you give the function which is also known as the *argument*. Lines 2-6 are comments that will display on the Octave command line if you type *help solarLatitude*. In Octave comment always start with a % sign and go to the end of the line. It is vital to always comment your code well. Uncommented code is difficult or impossible to reuse since it can be time consuming to go through the code to try to figure out what it does. The comments don't need to be long or wordy, they just need to let a programmer know enough to be able to use your function correctly.

So now we have our first function. Now we need to test it in Octave to make sure it gives a reasonable result. In Octave there are three basic types of numbers:

1. Scalars: These are just a single number like x =12;

2. Vectors: These are an indexed list of numbers like x = [0, 1, 2, 3, 4]
3. Matrix: These are multiple dimensioned lists of numbers like x = [1, 2, 3 : 4, 5, 6]

To test our function we can feed it a vector of input values and compare its outputs to what we know are the correct answers. So for our function lets put in the equinoxes and solsitices and make sure the results are correct. The vector of inputs is $days = [79, 172, 265, 355]$

So in Octave type:

```
days = [79, 172, 265, 355];
solarLatitude(days)
  0.0   23.4   -1.4   -23.4
```

Note that if you put a ; at the end of the line, Octave does not print the result, but if you leave it off, Octave prints the result after your command.

So looking at the results, they seem reasonable at these points.

Now lets plot the function in Octave. To plot a function you need to define the inputs. You could type every day from 1 to 365 into a vector but that would be tedious. So lets use Octave's ability to fill in a vector for us. The syntax is simple: $x = [start : step : stop]$. We want days from 1 to 365 in steps of 1 so we type

```
days=[1 : 1 : 365];
```

We don't want Octave to print 365 answers to the screen so we will save the result in a variable called *angles* with a command like:

```
angles = solarLatitude(days);
```

Now the vector *angles* had our values in it so we can plot it using the plot command:

```
plot(days,angles);
```

This produces a plot using days as the x axis and angles as the y axis as shown in Figure 3. The plot can be decorated with legends and grids, but how to do that can be found in other tutorials.

1.4 Octave Times and Dates

In many of our simulations we will need to work with times and dates. Octave stores times and dates as a DateNumber which is a floating point number representing the number of days since Jan 0, 0000. In general this number is not useful to the human reader, but there are many functions to convert to and from DateNumbers. For example, the function *now()* returns the DateNumber of the current time. *datenum(2014,1,21,12,35)* will return the DateNumber for January 21, 2014 at 12:35. A custom file *doy(t)* will give you the day of the year (1-366) from a given DateNumber. So by creating and using DateNumbers, we can store and manipulate time information. In our function, we need the day of the year as an input so you could calculate the present latitude of the sun by calling *solarLatitude(doy(now()))*;

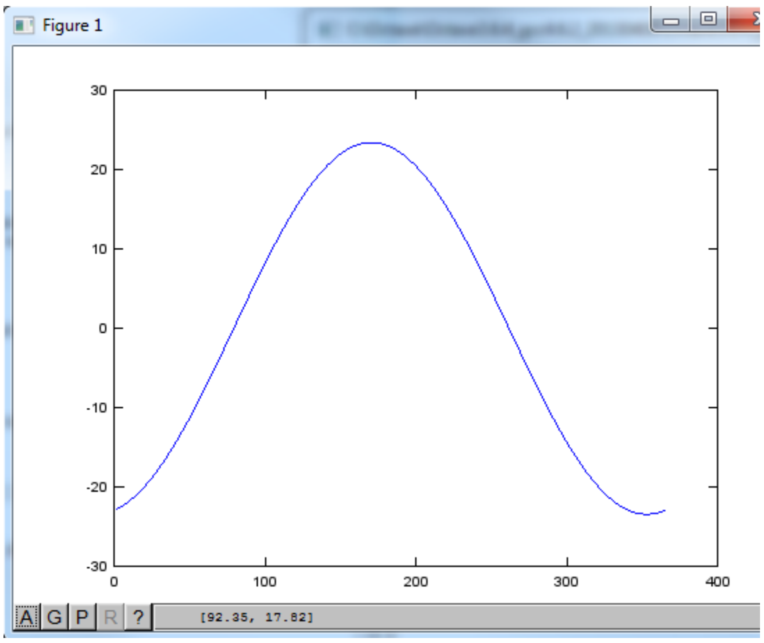

Figure 3: Output of solarLatitude Function