Wind Energy Toolkit: User Manual

Alec Koumjian

April 18, 2010

Contents

1 Introduction

The Wind Energy Toolkit is a compilation of software tools aimed towards students learning about preliminary wind turbine design and analysis. It features six core libraries that cover topics from wind data analysis to rotor aerodynamics. The code is designed to be used in conjunction with a course using the text Wind Energy Explained by James Manwell, et. al. This software is a Python port and hopefully advancement of the original Wind Energy Engineering Toolkit that was written in Visual Basic.
More information can be found at http://umass.edu/windenergy in the research tools section.
The code can be obtained at http://code.google.com/p/windenergytk.

This document summarizes most of the functions that the software can currently perform. For the core libraries, most of these examples show how to call the function in a Python shell with valid inputs, and the type of output to be expected. For more detail on valid inputs and outputs, see the API documentation.

2 Installation

Dependencies for the Core Libraries:

1. Numpy = 1.3
2. Scipy = 0.7.0
3. Scipy.special
4. Scikits.timeseries

Dependencies for the graphic interface:

1. Matplotlib = .98
2. wxPython = 2.8
3. wxmpl

Installation instructions:

1. Once the dependencies are sucessfully installed, untar the windenergytk code.
2. Use the command "python setup.py install" and the module should be added to your Python path.

3 Status of the Project

The project is divided into three major sections: the core libraries, the graphical interface, and additional utilities. These sections are all at various levels of completion, outlined below. There are also future prospects that include running the core libraries as a server side utility and web application. Project Status:

1. Analysis 100%
2. Synthesis 60%
3. Aerodynamics 70%
4. Mechanics 80%
5. Electrical 0%
6. System Performance 0%
7. GUI 40%
8. File Utilities 30%

4 Using Core Library Functions

4.1 Working with Wind Data Time Series

The toolkit has functions for analyzing wind data and generating artificial data using different models. These functions are designed to work with scikits.timeseries objects. Many of them will work on numpy.ndarray objects as well.

4.1.1 Loading Wind Data Files

The toolkit can load data files and store them as time series objects to be analyzed. Examples of the data format can be found in the "examples/" directory in the distribution package (they end in a ".dat" extension).

To parse one of these data files, the file_ops module is imported. Use the parse_file() function to separate the time series data and store it in a dictionary.

Listing 1: Reading in a .dat file

```
>>> import file_ops
>>> some_file = open("/home/aleck/Code/minicodes/windenergytk/examples/barnstable.dat","rb")
>>> some_file
<open file '/home/aleck/Code/minicodes/windenergytk/examples/barnstable.dat', mode 'rb' at 0
    x9623cf0>
>>> ts_dict = file_ops.parse_file(some_file)
```

For each time series, the dictionary contains another dictionary with meta information such as location and the name of the sensor in addition to the time data itself and can be accessed accordingly:

Listing 2: Looking at the data read in

```
>>> ts_dict.keys()
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18]
>>> ts_dict[1]
{'elevation': 21, 'name': 'etmp2degc', 'designation': 'primary', 'collector': 'rerl @ univ.
    of massachusetts', 'filters': {'dissimilar sensors': -988, 'out of range': -989, '
    missing data': -999, 'icing or wet snow event': -991}, 'comments': '', '
    meters_above_ground': 2, 'site_name': 'barnstable', 'coords': {'latitude':
    41.664830000000002, 'longitude': -70.304569999999998}, 'location': 'barnstable, ma', '
    timeseries': timeseries([ 3.2  3.3  3.4  3.5  3.6  3.6  3.6  3.6  3.6  3.6  3.6  3.6
    3.5  3.4  3.2
  2.9  2.9  2.6  2.3  2.3  2.2  2.2  2.3  2.3  2.4  2.4  2.6  2.6  2.7  2.7
  2.7  2.7],
   dates = [01-Jan-2006 00:00 01-Jan-2006 00:10 01-Jan-2006 00:20 01-Jan-2006 00:30
 01-Jan-2006 00:40 01-Jan-2006 00:50 01-Jan-2006 01:00 01-Jan-2006 01:10
 01-Jan-2006 01:20 01-Jan-2006 01:30 01-Jan-2006 01:40 01-Jan-2006 01:50
 01-Jan-2006 02:00 01-Jan-2006 02:10 01-Jan-2006 02:20 01-Jan-2006 02:30
 01-Jan-2006 02:40 01-Jan-2006 02:50 01-Jan-2006 03:00 01-Jan-2006 03:10
 01-Jan-2006 03:20 01-Jan-2006 03:30 01-Jan-2006 03:40 01-Jan-2006 03:50
 01-Jan-2006 04:00 01-Jan-2006 04:10 01-Jan-2006 04:20 01-Jan-2006 04:30
 01-Jan-2006 04:40 01-Jan-2006 04:50 01-Jan-2006 05:00 01-Jan-2006 05:10],
   freq  = T)
, 'time_step': 600, 'units': 'units', 'timezone': -5, 'time_period': '2006-01-01 to
    2006-04-03 09:50:00', 'logger_sampling': 2, 'type': 'external temperature', '
    report_created': '2006-07-14 16:08:28'}
```

The individual pieces can be accessed by their keys. For example, retriving the actual timeseries data is done using ts_dict[1]['timeseries'].

4.1.2 Analyzing Timeseries

Once a timeseries has been loaded uses the file_ops package, or made by some other method, it can be looked at with the tools in the analysis module. The following functions assume that the analysis module has been imported:

```
>>> from windenergytk import analysis
```

Whenever the variable "a_series" is used, it refers to an already loaded or created timeseries.

4.1.3 get_statistics()

The get_statistics() function returns the minimum, maximum, mean, standard deviation, and length of the timeseries array. The default behavior returns the values in a dictionary. Optionally the values can be returned as a list.

Listing 3: Getting basic statistics

```

>>> analysis.get_statistics(a_series)
{'std': 0.52035707391655595, 'max': 3.6000000000000001, 'min': 2.2000000000000002, 'size':
    32, 'mean': 2.9718750000000007}
>>> analysis.get_statistics(a_series, "list")
[2.9718750000000007, 0.52035707391655595, 3.6000000000000001, 2.2000000000000002, 32]
```

4.1.4 get_histogram_data()

The get_histogram_data() function returns the bin divisions and integral value histogram data that can later be graphed. The options let the user specify the number of bins to divide the data into, and whether or not to normalize the results to integrate to one.

```
>>> analysis.get_histogram_data(a_series, 20, normalized=False)
(array([2, 4, 2, 0, 0, 3, 0, 4, 0, 2, 0, 0, 0, 0, 2, 1, 0, 2, 2, 8]), array([ 2.2 ,  2.27,
    2.34,  2.41,  2.48,  2.55,  2.62,  2.69,  2.76,
        2.83,  2.9 ,  2.97,  3.04,  3.11,  3.18,  3.25,  3.32,  3.39,
        3.46,  3.53,  3.6 ]))

>>> analysis.get_histogram_data(a_series, 20)
(array([ 0.89285714,  1.78571429,  0.89285714,  0.        ,  0.        ,
        1.33928571,  0.        ,  1.78571429,  0.        ,  0.89285714,
        0.        ,  0.        ,  0.        ,  0.        ,  0.89285714,
        0.44642857,  0.        ,  0.89285714,  0.89285714,  3.57142857]), array([ 2.2 ,
            2.27,  2.34,  2.41,  2.48,  2.55,  2.62,  2.69,  2.76,
        2.83,  2.9 ,  2.97,  3.04,  3.11,  3.18,  3.25,  3.32,  3.39,
        3.46,  3.53,  3.6 ]))
```

The first results simply returns the count for each bin. Note that the second array describes the bin edges, so it has a length of 21 in this example.

4.1.5 crosscorrelate(), autocorrelate()

The analysis module also has the ability to correlate two timeseries, or autocorrelate a single one. The autocorrelate method is just wrapper for the crosscorrelate function using the same timeseries for both inputs.

```
>>> analysis.crosscorrelate(a_series, b_series, 10)

([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
 [-0.36521092103439157, -0.36160609254553211, -0.37158119282774843, -0.32505698690735946,
    -0.26040221198872837, -0.21543155504558056, -0.11467735043907809,
    0.00035630333759444592, 0.17680203494273927, 0.32690479145901119, 0.43411586400721747])
```

The correlation functions take an optional argument that specifies the max lag to calculate. The returned values are arrays of the lag index and the normalized amount that the timeseries correlated to.

4.1.6 get_weibull_params()

The analysis.get_weibull_params() function will take a mean and standard deviation and return the parameters c and k that describe a Weibull distribution with those statistical characteristics. This can be useful later for generating wind data with similar characteristics to another timeseries.

```
>>> analysis.get_weibull_params(5., 1.4)
(5.5175214395121817, 3.984615114175146)
```

4.1.7 block_average()

It may sometimes be better to reduce the size of the timeseries array by averaging blocks of the timeseries and labeling them with a new frequency. The block average function does this task. The inputs include the timeseries to be block averaged and the new frequency to conver it to. The available frequencies are those used by the scikits.timeseries package and can be found at .

```
>>> new_series = analysis.block_average(a_series, "hourly")
>>> new_series
timeseries([ 3.43333333  3.6         3.08333333  2.26666667  2.56666667  2.7        ],
   dates = [01-Jan-2006 00:00 ... 01-Jan-2006 05:00],
```

```
    freq   = H)

>>> len(a_series)
32
>>> len(new_series)
6
```

4.1.8 power_spectral_density()

This is a wrapper function for the power spectral density functions offered through the matplotlib library. It allows for the ability to window the timeseries data using a Hanning window or square window. The segment size specifies the length of unit to be used for the fast fourier transform procedure.

```
>>> analysis.power_spectral_density(a_series.data, .1)
(array([ 3.44258296,  3.35495142,  3.10480255,  2.72769245,  2.27463464,
        1.80194415,  1.36078285,  0.98883648,  0.70582479,  0.51341346,
        ...
        0.00534055,  0.00508732,  0.00525327,  0.00579735,  0.00660582,
        0.0075136 ,  0.00833654,  0.00890797,  0.00911224]),
 array([ 0.        ,  0.00039063,  0.00078125,  0.00117188,  0.0015625 ,
        0.00195312,  0.00234375,  0.00273438,  0.003125  ,  0.00351563,
        ...
        0.046875  ,  0.04726563,  0.04765625,  0.04804688,  0.0484375 ,
        0.04882812,  0.04921875,  0.04960938,  0.05      ]))
```

It should be noted that this function does not take timeseries objects as their argument. This will be changed in future revisions, once timeseries objects support subsecond frequencies.

4.1.9 Generating Timeseries

The synthesis module allows for a variety of theoretical models to build timeseries wind data. The following examples assume that the synthesis function has been imported.

```
>>> from windenergytk import synthesis
```

4.1.10 gen_arma()

This function employs an autoregressive moving average technique to generate a timeseries. The user may specify the mean, standard deviation, autocorrelation factor, and the number of point they want to generate. For now, the timeseries is given a frequency of one data point per minute. This will change in future revisions.

```
>>> synthesis.gen_arma(6., 2., .9, 20)
timeseries([ 6.          4.93769887  5.0143947   5.86851525  5.46211784  5.13990151
  4.33108298  4.70299863  3.6767802   5.04753763  5.35064961  4.92116187
  3.98690985  4.51543033  4.47688702  5.59489155  4.39838848  4.13372571
  3.29544382  3.05370454],
    dates = [01-Jan-2001 00:00 ... 01-Jan-2001 00:19],
    freq  = T)
```

The more data points that are created, the better the function meets the average and standard deviation requirements.

4.1.11 gen_ts_from_tpm()

This function makes use of a Markov process whereby a timeseries is generated from a previously developed transition probability matrix (for now represented by the variable tpm).

```
>>> new_ts = synthesis.gen_ts_from_tpm(tpm, 2., 20)
>>> new_ts
timeseries([  4.03174413  11.16332847  10.31461996  10.83023286  11.92281933
  11.59823027  10.30378196  11.34647608  10.01147421  10.02408444
  11.08574492  10.4952402   10.6724465   10.02908196   9.37970177
   0.89870268   0.25366088   3.93376901   4.39539457  10.37376654],
   dates = [01-Jan-2001 00:00 ... 01-Jan-2001 00:19],
   freq  = T)
```

The function also requires that the bin width be specified (here it is 2) in whatever units the user wants to measure the timeseries data in. The last input is the length of the timeseries array.

4.1.12 gen_markov_tpm()

If you want to model a new timeseries from a currently existing one, it's possible to generate a transition probability matrix first from the existing one and feed it into the previous function.

```
synthesis.gen_markov_tpm(a_series, 5)
array([[ 0.875      ,  0.125      ,  0.         ,  0.         ,  0.         ],
       [ 0.         ,  0.83333331,  0.16666667,  0.         ,  0.         ],
       [ 0.         ,  0.         ,  0.5        ,  0.         ,  0.5        ],
       [ 0.33333334,  0.         ,  0.         ,  0.33333334,  0.33333334],
       [ 0.         ,  0.08333334,  0.         ,  0.08333334,  0.83333331]], dtype=float32)
>>>
```

4.2 Rotor Aerodynamics

The aerodynamics module provides functions for testing the performance of certain rotor blade designs. It also allows the user to produce optimum rotor design given a certain range of parameters using the techniques employed in the Wind Energy Explained book. This section assumes that the "aerodyn" module has been imported.

```
>>> from windenergytk import aerodyn
```

4.2.1 optimum_rotor()

This function takes a list of given parameters, such as the desired lift coefficient, angle of attack, tip speed ratio, and blade dimensions. It then returns a matrix of the chord and twist of each section of blade that best emulates those parameters.

```
>>> lift_coef = 4.
>>> AoA = .3
>>> tsr = [4., 5., 6., 6., 6., 6., 7., 8., 9., 10]
>>> radius = 10
>>> hradius = 1
>>> blades = 3
>>> sections = 10
>>> aerodyn.optimum_rotor(lift_coef, AoA, tsr, radius, hradius, blades, sections)

[[0, 0.16514867741462683, 0.0],
[1, 0.13255153229667402, 0.018453536538179546],
[2, 0.11065722117389563, 0.025698583040107224],
[3, 0.11065722117389563, 0.038547874560160837],
[4, 0.11065722117389563, 0.051397166080214447],
[5, 0.11065722117389563, 0.064246457600268064],
[6, 0.09495170634275632, 0.056733628360690423],
[7, 0.083141231888441219, 0.050729596832123344],
[8, 0.07393903765794034, 0.045841924844088687],
[9, 0.066568163775823808, 0.041795126957110006]]
```

```
>>>
```

This kind of matrix can also be fed into the linear or nonlinear rotor analysis functions. It states the station, chord, and twist at each section.

4.2.2 rotor_analysis()

The rotor analysis function can be used one of two ways. One method involves using a linear approximation of a lift coefficient vs. angle of attack, in which case the slope and intercept are fed into the function. The other involves feeding a set of points inside of an array as a more detailed approximation of a lift coefficient versus angle of attack curve. In this case, the program interpolates values between the points given.

```
>>> tip_speed_ratio = 10
>>> number_blades = 3
>>> pitch_0 = .1
>>> blade_radius = 10
>>> hub_radius = 1
>>> lift_curve = [2, 1]
>>> drag_curve = [.2, .5]
>>> method = "linear"
>>> aerodyn.rotor_analysis(rct_matrix, tip_speed_ratio, number_blades, pitch_0, blade_radius
    , hub_radius, lift_curve, drag_curve, method)
[[1.0, 1.0, −0.72396523891638109, −0.62396523891638112, −0.44793047783276219,
    0.35520695221672377, −0.02144358657715318, 0.015437499991458428, −0], [2.0, 1.0,
    −0.75083587340192903, −0.63238233686374945, −0.50167174680385807, 0.34983282531961418,
    −0.0092480161116521731, 0.0033883338838999657, 0.0], ...
>>>
```

Alternatively, separate arrays can be used to describe points on the lift coefficient an drag coefficient curves:

Listing 4: Using nonlinear curve

```
>>> aerodyn.rotor_analysis
    ([[.2,2.,.3],[.4,2.,.4],[.6,2.,.5],[.8,2.,.6],[.9,2,.6,[.9,2.,.6]],
10.,
3,
.1,
10.,
1.,
[[0.,0.],[1.,30],[1.5,40]],
[[0.,0.],[1.,30],[1.5,40]],
"nonlinear")
[[2.0, 1.5706112057342645, 0.057708874366294849, 0.458028101867056, 1.7408430560116797,
    1.7312662309888456, 0.37769080854769471, 0.093099882389402228, −0.0046811935852177477],
    ...
>>>
```

4.3 Predicting Blade Movement

The *mechanics* module provides functions for describing blade movements and blade fatigue while in operation. The following functions assume that the mechanics module is imported.

4.3.1 Natural Frequencies

The natural frequencies of beams can be calculated using the Euler or Myklestaad methods. These frequencies are then used in functions that predict blade movement.

The euler_beam_vibrations() function is used for blades which are assumed to have uniform dimensions and mass distribution.

```
>>> beam_length = 10
>>> area_moment = 4
>>> mass_per_length = 2
>>> elastic_modulus = 1.3
>>> mode = 2
>>> mechanics.euler_beam_vibrations(beam_length, area_moment, mass_per_length,
    elastic_modulus, mode)
(0.35529550069413179, 0.46940911329743357)
```

The myklestad_beam_vibrations() require much more detailed input than the euler beam method.

```
>>> sec_lengths = [1., 1., 1., 1.]
>>> sec_masses = [1., 1., 1., 1.]
>>> e_i = [1.5, 1.5, 1.5, 1.5]
>>> density = 1.
>>> rot_velocity = 20
>>> freq_start = 1
>>> freq_step = 0.5
>>> freq_final = 30
>>> mechanics.myklestad_beam_vibrations(sec_lengths, sec_masses, e_i, density, rot_velocity,
     freq_start, freq_final, freq_step)
[2.5, 3.0, 3.5, 6.5]
```

The results are a list of natural frequencies in radians per second.

4.3.2 Flapping Blade Motion

The myklestad and euler method modes of oscillation can be used as part of the input for solving the flapping blade motion. The hinge_spring_flapping() function returns results that describe the coning, vertical, and lateral tilting terms that describing the motion of the plane of rotation.

```
>>> num_blades = 3
>>> blade_radius = 10
>>> blade_chord = .5
>>> blade_mass = 100
>>> lift_curve_slope = 2
>>> blade_pitch_angle = .2
>>> rot_nat_freq = 2.5
>>> non_nat_freq = 2.
>>> yaw_to_blade = 1
>>> yaw_rate = .2
>>> cross_flow = 1.5
>>> linear_shear = 3
>>> air_density = .01
>>> rot_velocity = 2
>>> tip_speed_ratio = 10
mechanics.hinge_spring_flapping(num_blades, blade_radius, blade_chord, blade_mass,
    lift_curve_slope, blade_pitch_angle, rot_nat_freq, non_nat_freq, yaw_to_blade, yaw_rate,
     cross_flow, linear_shear, air_density, rot_velocity, tip_speed_ratio)
(0.013276940189759386, -0.18395789165942605, -0.0024270777670343615)
```

5 Using the Graphical Interface

The graphical interface only has minimal functionality at the moment. However, enough is completed that it can demonstrate that way it will be used and the overall layout.

The start the GUI, go to the directory with the “gwindtk.py” file and execute it. From a *nix environment, that will mean going to a shell and typing:

```
# ./gwintk &
```

This opens the main window.

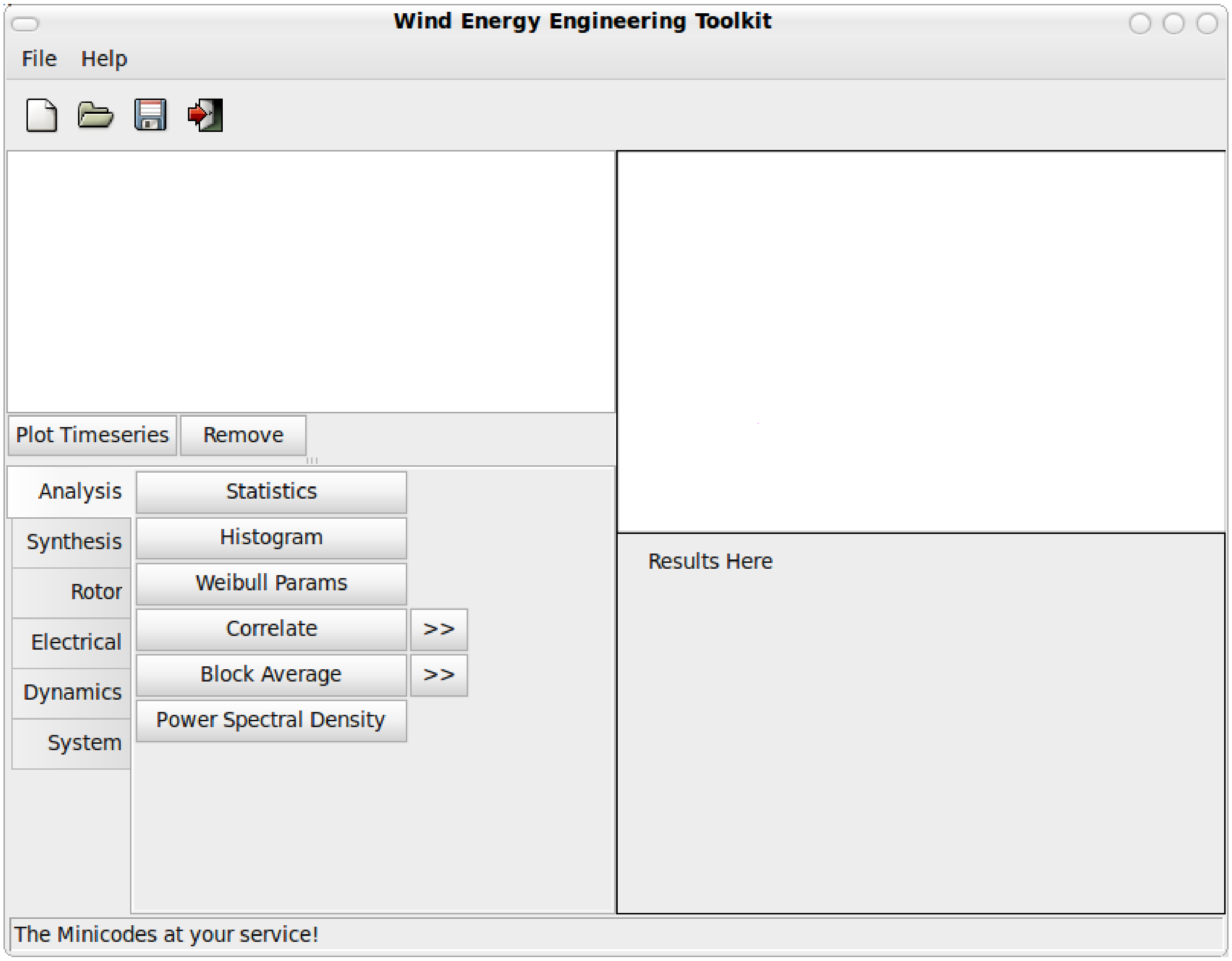
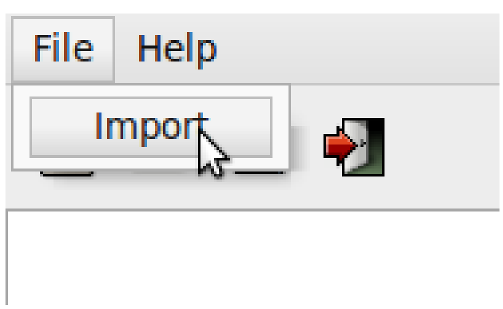


Figure 1: Main Page

5.1 Loading Files and Plotting Timeseries

Currently the graphical interface loads *.dat files that can be downloaded from the UMass website. By using the menu options File, Import, a data file can be loaded from a local directory. Sample files are included in the examples directory.



Once any number of timeseries are loaded (shown in the upper left corner), they can be plotted, analyzed, or manipulated. Plotting a timeseries will display a graph of it in the upper right corner.

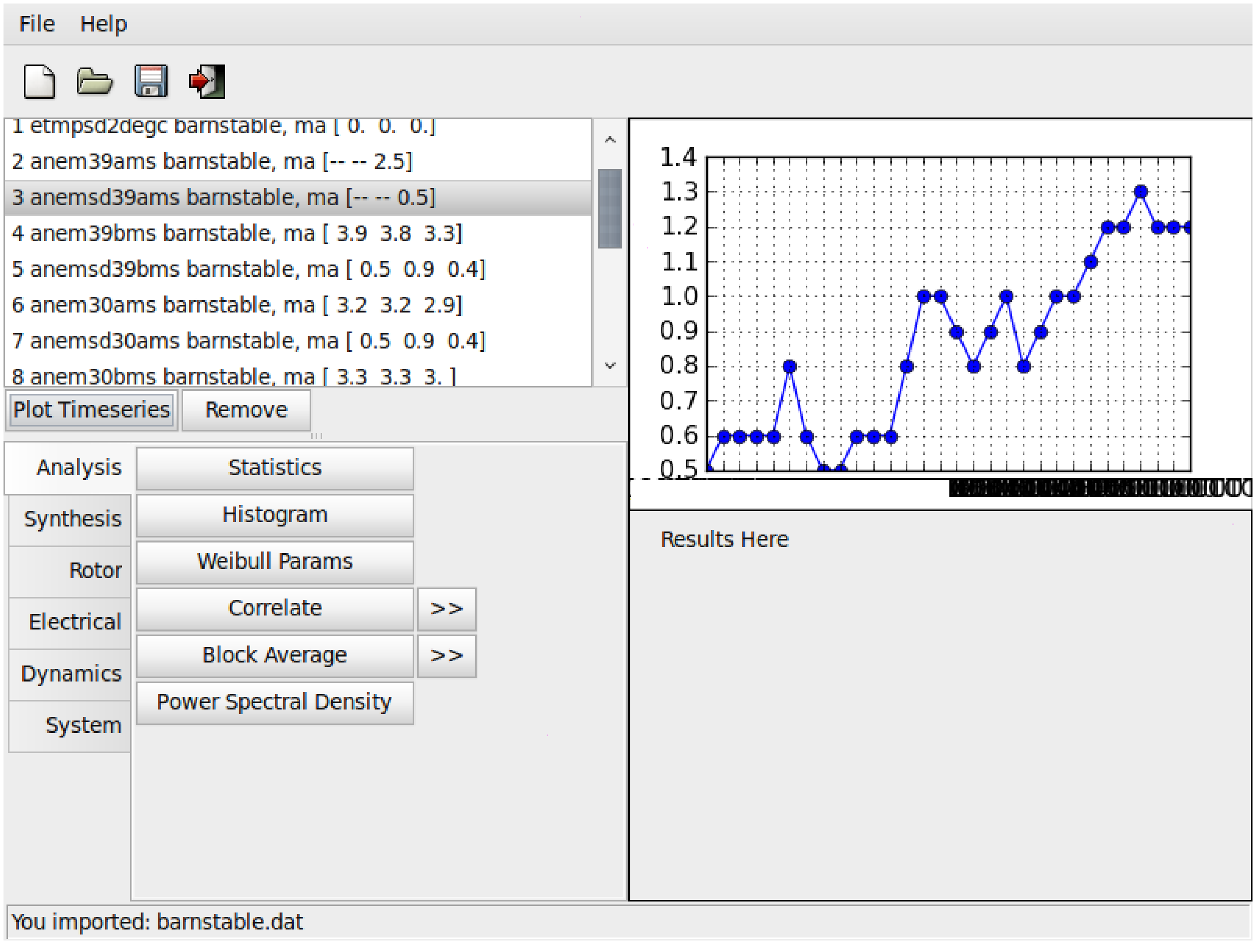


Figure 2: Plotting a Timeseries

5.2 Other Kinds of Functions

Functions provided by each of the different core librations are found by selected the tab in the bottom right corner. Once there, options such as retrieving histogram data or crosscorrelations can be performed:

Most of these functions require selecting one or more timeseries to perform actions on them.

When generating new wind timeseries data, the new arrays should appear at the bottom of the list of available data to work with. Since they are artificial, they have no location of origin.

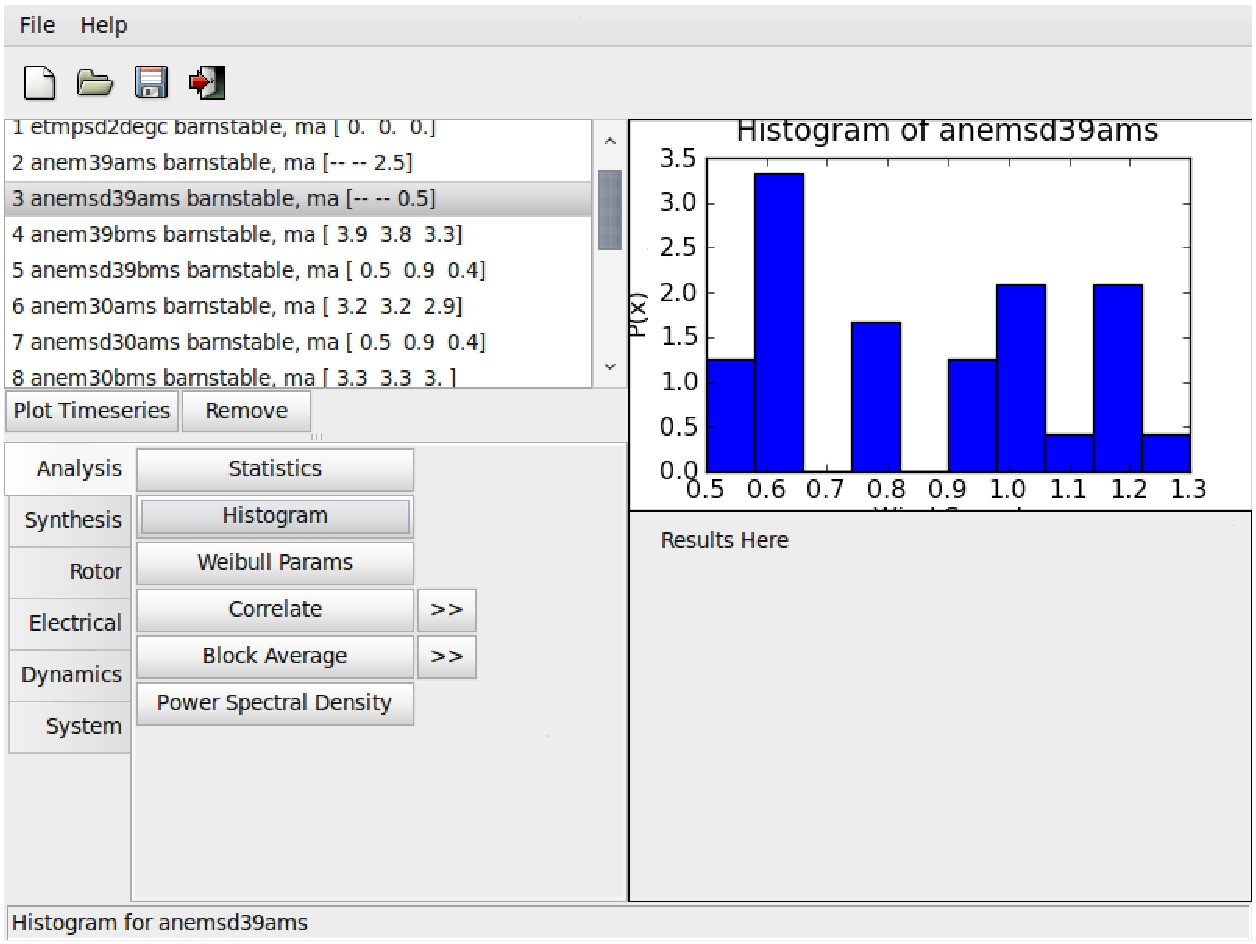


Figure 3: Histogram data

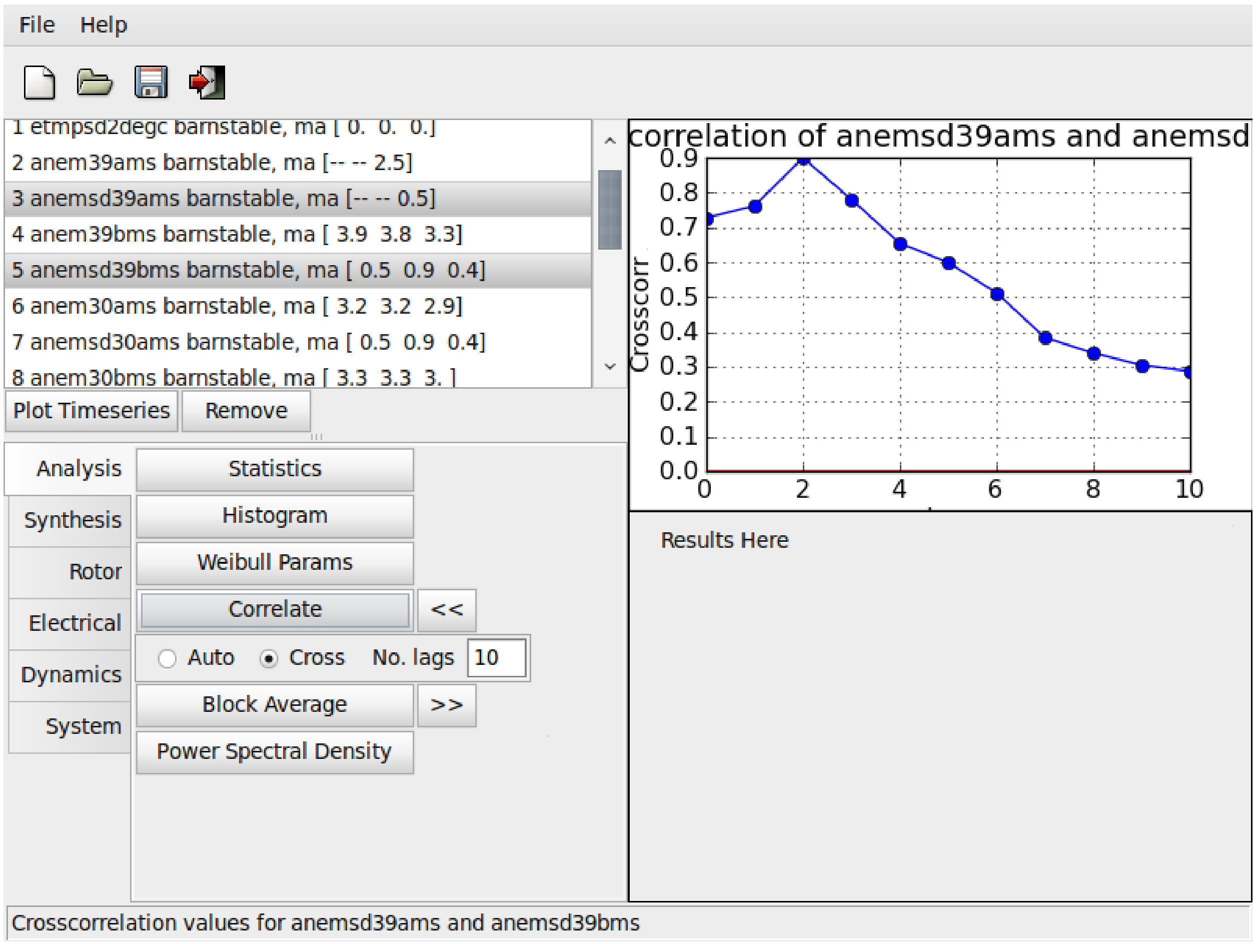


Figure 4: Crosscorrelation

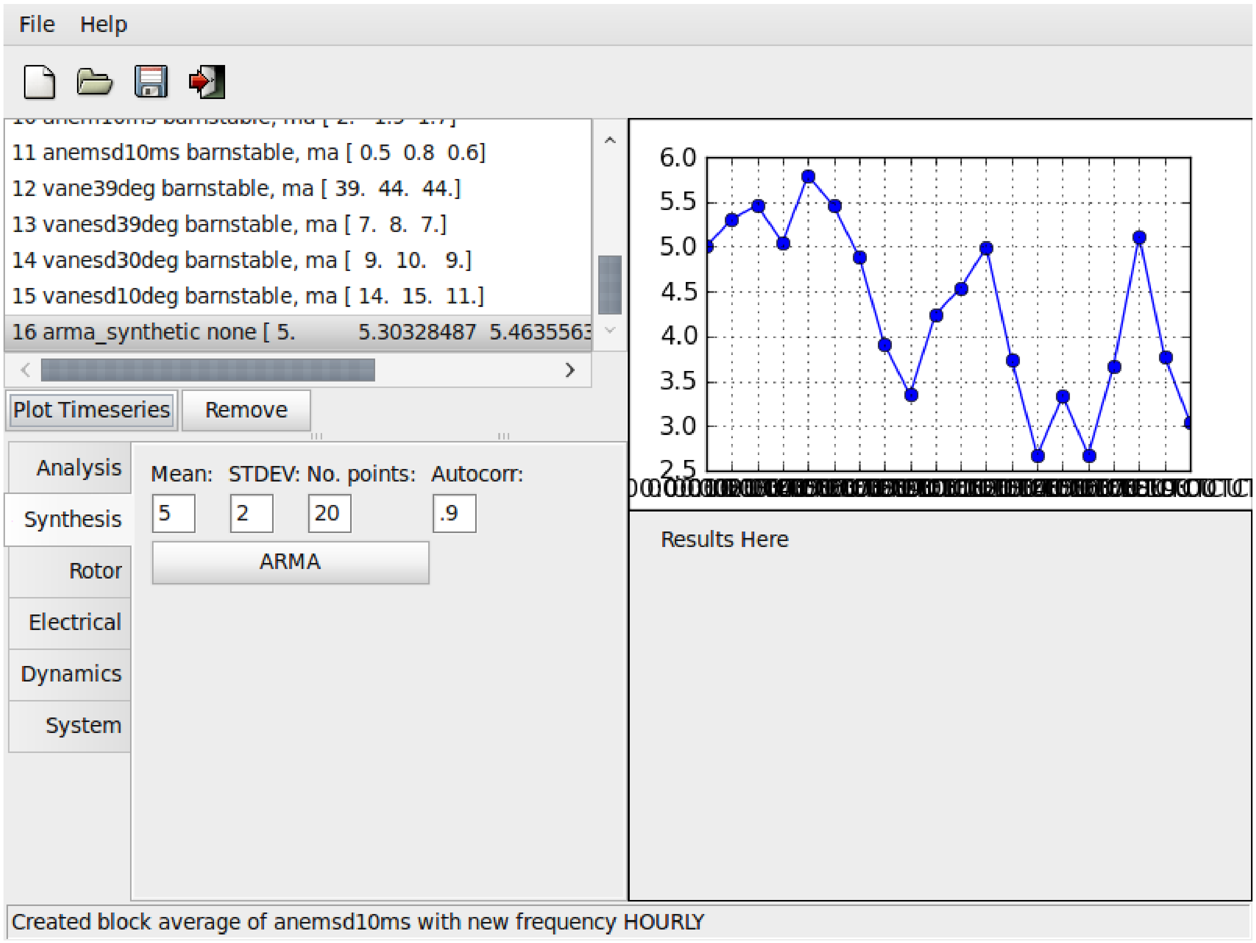


Figure 5: A new timeseries is added using ARMA